R215-01

VOS Transaction Processing Facility Guide

Stratus Computer, Inc.

Notice

Manual Name: *VOS Transaction Processing Facility Guide*

Part Number: R215
Revision Number: 01
VOS Release Number: 14.0.0
Printing Date: December 1998

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

Contents

Figures

Tables

Preface

The *VOS Transaction Processing Facility Guide (R215)* documents the tasking, queue I/O, and transaction protection capabilities offered by the VOS operating system. This manual prepares the reader for the VOS Transaction Processing Facility Reference Manuals, which are language-specific manuals documenting the TPF commands, monitor requests, and subroutines.

This manual is intended for programmers of transaction processing systems and other applications that use the tasking, queue I/O, or transaction protection capabilities of the VOS operating system.

Before working with the *VOS Transaction Processing Facility Guide (R215)*, you should be familiar with the Stratus manual documenting the high-level language in which your application will be written. In addition, you should be familiar with the VOS Subroutines manual for that high-level language.

Manual Version

This manual is a revision. For information on which release of the software this manual documents, see the Notice page.

Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual. Note, however, that change bars are not used in new chapters or appendixes.

This revision incorporates information about the following changes.

- Open StrataLINK on Continuum-series modules
- ORACLE relational database system
- C language `signal` function in relation to a task
- Process switching vs. task switching
- Initializing `break_char` bit
- Reading a transaction file opened in `dirty_input` mode
- File positioning and `s$seq_read`
- A process writing to a queue using `s$msg_send`
- Single-event-notify message response
- Starting transactions in the server with two-way server queues
- A remote requester port connection is detached

Related Manuals

Refer to the following Stratus manuals for related documentation.

- VOS Transaction Processing Facility Reference manuals

  VOS BASIC Transaction Processing Facility Reference Manual (R032)
  VOS C Transaction Processing Facility Reference Manual (R069)
  VOS COBOL Transaction Processing Facility Reference Manual (R034)
  VOS FORTRAN Transaction Processing Facility Reference Manual (R036)
  VOS Pascal Transaction Processing Facility Reference Manual (R038)
  VOS PL/I Transaction Processing Facility Reference Manual (R015)

- VOS Subroutines manuals

  VOS BASIC Subroutines Manual (R018)
  VOS C Subroutines Manual (R068)
  VOS COBOL Subroutines Manual (R019)
  VOS FORTRAN Subroutines Manual (R020)
  VOS Pascal Subroutines Manual (R021)
  VOS PL/I Subroutines Manual (R005)

- VOS Language manuals

  VOS BASIC Language Manual (R011)
  VOS Standard C Reference Manual (R363)
  VOS Standard C User's Guide (R364)
  VOS COBOL Language Manual (R010)
  VOS FORTRAN Language Manual (R013)
  VOS Pascal Language Manual (R014)
  VOS Pascal User's Guide (R144)
  VOS PL/I Language Manual (R009)
  VOS PL/I User's Guide (R145)
  VOS Symbolic Debugger User's Guide (R308)

- VOS Forms Management System manuals

  VOS BASIC Forms Management System (R033)
  VOS C Forms Management System (R070)
  VOS COBOL Forms Management System (R035)
  VOS FORTRAN Forms Management System (R037)
  VOS Pascal Forms Management System (R039)
  VOS PL/I Forms Management System (R016)

Notation Conventions

This manual uses the following notation conventions.

- *Monospace* represents text that would appear on your display screen (such as commands, subroutines, code fragments, and names of files and directories).
  For example:

  ```
  /= task_id 2
  ```

- *Monospace italic* represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  ```
  /= task_id task_id
  ```

- *Italics* introduces or defines new terms. For example:

  A *two-way queue* handles messages and replies to those messages.

- *Boldface* emphasizes words in text. For example:

  The event count is incremented by 1 for **each** notification.

Key Mappings for VOS Functions

VOS provides several command-line and display-form functions. Each function is mapped to a particular key or combination of keys on the terminal keyboard. To perform a function, you press the appropriate key(s) from the command line or display form. For an explanation of the command-line and display-form functions, see the *Introduction to VOS (R001)*.

The keys that perform specific VOS functions vary depending on the terminal. For example, on a V103 ASCII terminal, you press the Shift and F20 keys simultaneously to perform the `INTERRUPT` function; on a V105 PC/+ 106 terminal, you press the 1 key on the numeric keypad to perform the `INTERRUPT` function.

> **Note:** Certain applications may define these keys differently. Refer to the documentation for the application for the specific key mappings.

The following table lists several VOS functions and the keys to which they are mapped on commonly used Stratus terminals and on an IBM PC® or compatible PC that is running the Stratus PC/Connect-2 software. (If your PC is running another type of software to connect to a Stratus host computer, the key mappings may be different.) For information about the key mappings for a terminal that is not listed in this table, refer to the documentation for that terminal.

VOS Function	V103 ASCII	V103 EPC	IBM PC or Compatible PC	V105 PC/+ 106	V105 ANSI
`CANCEL`	F18	* †	* †	5 † or * †	F18
`CYCLE`	F17	F12	Alt-C	4 †	F17
`CYCLE BACK`	Shift-F17	Shift-F12	Alt-B	7 †	Shift-F17
`DISPLAY FORM`	F19	- †	- †	6 † or - †	F19 or Shift-Help
`HELP`	Shift-F8	Shift-F2	Shift-F2	Shift-F8	Help
`INSERT DEFAULT`	Shift-F11	Shift-F10	Shift-F10	Shift-F11	F11
`INSERT SAVED`	F11	F10	F10	F11	Insert_Here
`INTERRUPT`	Shift-F20	Shift-Delete	Alt-I	1 †	Shift-F20
`NO PAUSE`	Shift-F18	Shift-* †	Alt-P	8 †	Shift-F18

† Numeric-keypad key

Online Documentation

You can find additional information by viewing the system's online documentation in `>system>doc`. The online documentation contains the latest information available, including updates and corrections to Stratus manuals and a master glossary of terms.

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.

- Customers in North America can call the Stratus Customer Assistance Center (CAC) at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see the file `cac_phones.doc` in the directory `>system>doc` for CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

How to Comment on This Manual

You can comment on this manual by using the command `comment_on_manual` or by completing the customer survey that appears at the end of this manual. To use the `comment_on_manual` command, your system must be connected to the RSN. If your system is **not** connected to the RSN, you must use the customer survey to comment on this manual.

The `comment_on_manual` command is documented in the manual *VOS System Administration: Administering and Customizing a System (R281)* and *VOS Commands Reference Manual (R098)*. There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press Enter or Return, and complete the data-entry form that appears on your screen. When you have completed the form, press Enter.
- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press Enter or Return. Enter this manual's part number, `R215`, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press Enter.

  Note: If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail` request of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Chapter 1:
Transaction Processing Facility Overview

This chapter provides information on the following topics:

Online Transaction Processing Systems

An online transaction processing (OLTP) system is an interactive computer system capable of handling many user requests and successfully processing those requests while sharing system resources and data. OLTP application environments typically consist of a large number of users and a large volume of data. They require minimal response time and maximum system availability. Familiar OLTP systems include automatic teller machine (ATM) networks, airline reservation systems, and manufacturing on-floor scheduling systems.

The following are some specific operating characteristics of OLTP systems.

- Multitasking and task scheduling. OLTP systems must handle hundreds of user requests concurrently (or seemingly concurrently) without exhausting system resources. Multiple user requests must be able to execute the same section of code and access the same data files. Also, there must be ways to prioritize and schedule user requests so that they are processed according to importance and order of submission.

- Online database integrity. There must be a way to ensure that user-entered requests are either completely and successfully accomplished or appear to have never occurred. The system must never allow partial file updates.

- Concurrent I/O facility. Multiple users must be able to read the same file, and even the same record. However, a record that is being updated must be exclusively locked.

- Centralized management from a monitor terminal. Some types of privileged manual intervention are usually desirable, such as canceling tasks when terminals malfunction and setting or resetting parameters. Many applications send messages to a monitor terminal, requesting the monitor operator to perform specific actions.

- Limited user capabilities. The set of functions performed from end-user terminals must be well-defined and limited to preserve data integrity and system security.

- User-oriented system. A user-oriented system displays forms and menus on the user terminals for easy data entry. Response time is short, and system down-time is negligible.

- Logging. A detailed log of file updates ensures that a complete recovery is possible if the system fails or if files are corrupted.

The Transaction Processing Facility

The Transaction Processing Facility (TPF) offers the applications programmer the means to satisfy all requirements of an OLTP system. TPF consists of several capabilities that are extensions to the VOS operating system. To develop an OLTP system, you combine the features described in this manual with the programming capabilities described in other manuals. Manuals containing related information are listed in the Preface of this manual. Your final OLTP application programs might contain subroutines described in the VOS Transaction Processing Facility Reference manuals, as well as subroutines described in the VOS Subroutines manuals.

As shown in Figure 1-1, the Transaction Processing Facility provides the following capabilities:

- Queue I/O
- Transaction protection, including transaction logging
- Tasking, including an optional monitor task for each tasking process in an application.

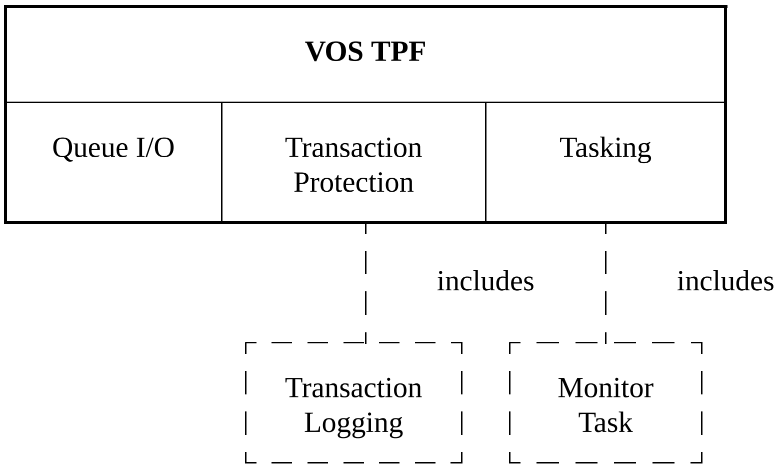


Figure 1-1. TPF Capabilities

These capabilities can be used collectively or individually within an application. For example, your application can use tasking without using transaction protection or queue I/O.

Online Transaction Processing on VOS

There are several processing goals that any OLTP system must be able to achieve. A VOS application can satisfy these goals because of the standard Stratus system architecture and the special features offered through TPF.

One of these goals is maximum system availability. Any application, whether or not it uses TPF capabilities, benefits from the fault-tolerant operating features included within the architecture of Stratus modules. *Fault-tolerance* refers to duplexed hardware components that operate in lock-step simultaneously. If a component fails, processing continues. Meanwhile, the system automatically dials into a local Customer Assistance Center (CAC) to report its defective component, and a replacement component is shipped to your installation.

Another OLTP processing goal is a communications system that does not restrict the location or number of terminals. Applications should be easily expandable over multiple modules and networked systems. This goal can be satisfied with the Open StrataLINK™, StrataLINK, and StrataNET™ communication networks. All TPF capabilities operate across these networks. Open StrataLINK is supported on all Continuum-series modules and on XA/R-series modules running VOS Release 12.2 or later.

A third OLTP processing goal is the ability to handle many user requests efficiently. The tasking feature does this for you.

Other goals involve the system's ability to accomplish the following I/O requirements:

- shared data-file reads
- exclusive locks during updates
- minimal lock contention and wait time for data-file access
- guaranteed data-file integrity
- immediate data-file updates
- high volume of input and output

TPF accomplishes these goals with its transaction protection, record locking, and queue I/O facilities.

Tasking

Tasking allows many terminals and users to share resources. Individual tasks are allocated some resources, but many of the overhead requirements can be shared. Tasking significantly increases the number of user requests processed within a given time period by minimizing both CPU competition and virtual memory usage.

Many tasks can execute the same section of code, or individual tasks can execute different code. Tasks can be assigned priorities, and those priorities can be changed using various scheduling techniques. Task switching is controlled by the programmer or by operating system defaults.

Terminals assigned to tasks are not login terminals. This means that the application program controls the terminal. The end-user cannot enter VOS commands, interrupt program execution, or perform any function not specifically permitted by the application program.

Transactions and Transaction Protection

Transaction protection guarantees consistency and integrity to data files that have been converted to *transaction files*. Any VOS files, except for stream files and some queues, can be converted to transaction files.

A *transaction* is a unit of work consisting of one or more I/O requests on transaction files. The I/O requests contained in a transaction are considered as a unit, and either all updates requested in the transaction are completed, or none are made. Transaction protection guarantees that a transaction file or a set of transaction files never contains partially completed transactions after any type of program, system, module, or file storage malfunction.

Transactions are defined directly in the program code by subroutine calls that indicate the beginning and end of a transaction. The I/O requests made within the bounds of the transaction define the work that the transaction must accomplish. If all defined work cannot be completed, then all files are returned to the state they were in before the transaction started.

Although a transaction should be defined as a concise set of I/O calls, there are no limitations on what is considered a transaction, or on what types of actions a transaction can perform. A single transaction might involve reads and writes to one or many transaction files.

A transaction ends in one of the following ways.

- The transaction is *committed*, which guarantees that all updates on transaction files requested in the transaction are made or will be made as soon as possible.
- The transaction is *aborted*, which guarantees that no updates on transaction files requested in the transaction were made or will be made.

Queue I/O

Because most OLTP systems spend more time performing I/O than performing computations, system efficiency depends on the efficiency of I/O operations. Queues are effective intermediaries in systems with high-volume input/output requirements.

The operating system supports five different queue types. All queues can be used on multimodule systems connected by the Open StrataLINK, StrataLINK, or StrataNET communication network. Each type of queue has specific advantages over the others. The application's requirements determine which type to use.

TPF Application Management

Two application management features are available in TPF — the monitor task and the transaction processing roll-forward facility. The monitor task provides the application with an administrative environment, from which an operator issues commands and requests affecting the currently running tasks. The roll-forward facility provides a way to reconstruct a corrupted data file or set of data files. The roll forward starts with a backup version of your data file and reapplies transactions using a set of transaction logs. The transaction logs are maintained for you by the background process that manages transaction processing.

The Monitor Task

The monitor task is an optional TPF capability associated with tasking. It is started from within the application program module. Each tasking process in an application may include a monitor task. The various monitor tasks in an application can perform the same functions, or they can each perform different functions.

The applications programmer controls which administrative functions can be performed from the monitor task. The programmer includes only the commands that are useful to the application, choosing from among the VOS commands and approximately 30 TPF commands related to tasking and queues. In addition, the programmer can write application-specific administrative commands and include them.

Other features related to monitor task programming are as follows:

- The monitor task can receive messages from other tasks running in the process.
- The application can run the monitor task without a monitor terminal. In this situation, the monitor task receives instructions from a queue.
- The application can start the monitor task with an initial request for the monitor to perform. Another option causes the monitor task to return control to the calling program immediately after the single initial request is performed.

The Roll-Forward Facility

The roll-forward facility is associated with transaction protection. If a transaction-protected data file is corrupted, or if a set of transaction-protected data files is no longer synchronized, the `tp_restore` command can help recover the files. The roll forward starts with a backup version of the data files, and then reapplies transactions using a set of transaction logs.

The transaction logs are created and maintained by the TPOverseer process. The *TPOverseer* is a background process that manages transaction processing and, as a result, maintains a log of transactions in their various states. The logs contain all of the information needed to re-create all transactions, in the same order as they were first performed. On multimodule systems, the logs from all modules can be used to accurately simulate the running application.

The transaction logs are always created whenever transaction protection is used. If you expect to use the roll-forward facility, you must specify that you want the logs **saved** on all of the modules involved.

Related User Interfaces

This discussion of related user interfaces provides information on the following topics:

- ‘‘Terminal Configuration”
- ‘‘Forms Development”
- ‘‘The Data File System”

Terminal Configuration

An OLTP system interfaces with users through terminals. In tasking, the terminal interface is accomplished by associating a task with a terminal. This can be done through a *task configuration table*, which assigns a predetermined set of terminals to an application every time the application is started. Tasks and terminals can also be associated dynamically while the application is executing, either by calling a subroutine within the program or through a command issued by the monitor task.

Forms Development

An integral part of the user interface is the format of information passed between the system and the user. Most OLTP systems display forms and/or menus to aid the user in entering requests. The *Forms Management System (FMS)* is the Stratus product that creates forms and menus. FMS also manages the form displays, accepts the data entered by the user, performs some data validation, and redisplays values and messages back to the user after validation and processing. For complete information about the FMS capabilities, see the VOS Forms Management System manuals.

The following steps briefly define one method for integrating a form into your application program.

1. Define, name, and store the form using FMS commands.
2. Declare the input variables in your application program by copying or including the field-values include file (`name_incl.lang`) produced by FMS as a result of step 1.
3. Within your application program, use FMS statements to display the form on user terminals and to read and perform limited validations on the data entered by the user. Your program can perform more extensive validations.
4. Also within your application program, use FMS statements to send changed or updated variables back to the user terminal.

Your application can also implement direct terminal I/O using VOS subroutines. See the ‘‘Default Terminal Ports and Terminal I/O” section in Chapter 2, ‘‘Tasking.”

The Data File System

The data files for your TPF application can be VOS files defined with VOS subroutines or commands, or you can use an ORACLE™ relational database system supported by Stratus.

ORACLE relational database systems include their own implementation of transaction protection. To protect data in those database file systems, use the specialized transaction protection capabilities of ORACLE. To protect data in VOS files, use the transaction protection facility described in this manual.

Programming a TPF Application

This discussion of programming a TPF application provides information on the following topics:

- ‘‘TPF Commands, Requests, and Subroutines”
- ‘‘The Source Code”
- ‘‘Error Messages”
- ‘‘Debugging”

TPF Commands, Requests, and Subroutines

The TPF product contains commands, requests, and subroutines, defined as follows:

- Commands. The commands are issued at command level like the standard VOS commands, but they relate to TPF capabilities. An OLTP administrator might use these commands for troubleshooting or normal monitoring of the application. The commands can also be executed from the monitor task (as can all other VOS commands). Examples of TPF commands are `display_lock_wait_time` and `set_transaction_file`.

- Monitor Requests. These requests can be included as functions of the monitor task. Unlike commands, requests cannot be issued at command level. They can be issued only from within the monitor task. Examples of TPF monitor requests are `start_task` and `stop_task`.

- Subroutines. The subroutines are used within the application program. TPF subroutines support tasking, queue I/O, and transaction protection capabilities. Examples of TPF subroutines are `s$start_task` and `s$commit_transaction`.

The following sections provide information on TPF commands, requests, and subroutines:

- Appendix A, ‘‘List of Transaction Processing Facility Commands and Requests” lists the TPF commands and monitor requests.

- ‘‘Task-Related Subroutines” in Chapter 2 lists TPF subroutines related to tasking.

- ‘‘Monitor-Related Subroutines” in Chapter 3 lists TPF subroutines related to the monitor task.

- ‘‘Details of `s$commit_transaction`” and ‘‘Details of `s$abort_transaction`” in Chapter 4 provide information about subroutines related to transaction protection.

- ‘‘Queue-Related Subroutines” in Chapter 5 lists TPF subroutines related to queue I/O.

- The VOS Transaction Processing Facility Reference manuals include complete descriptions of each TPF command, monitor request, and subroutine.

The Source Code

Figure 1-2 shows the Stratus products typically used to develop an OLTP application. A TPF application program might include VOS language subroutines, TPF subroutines, Forms Management System language statements, and database I/O queries.

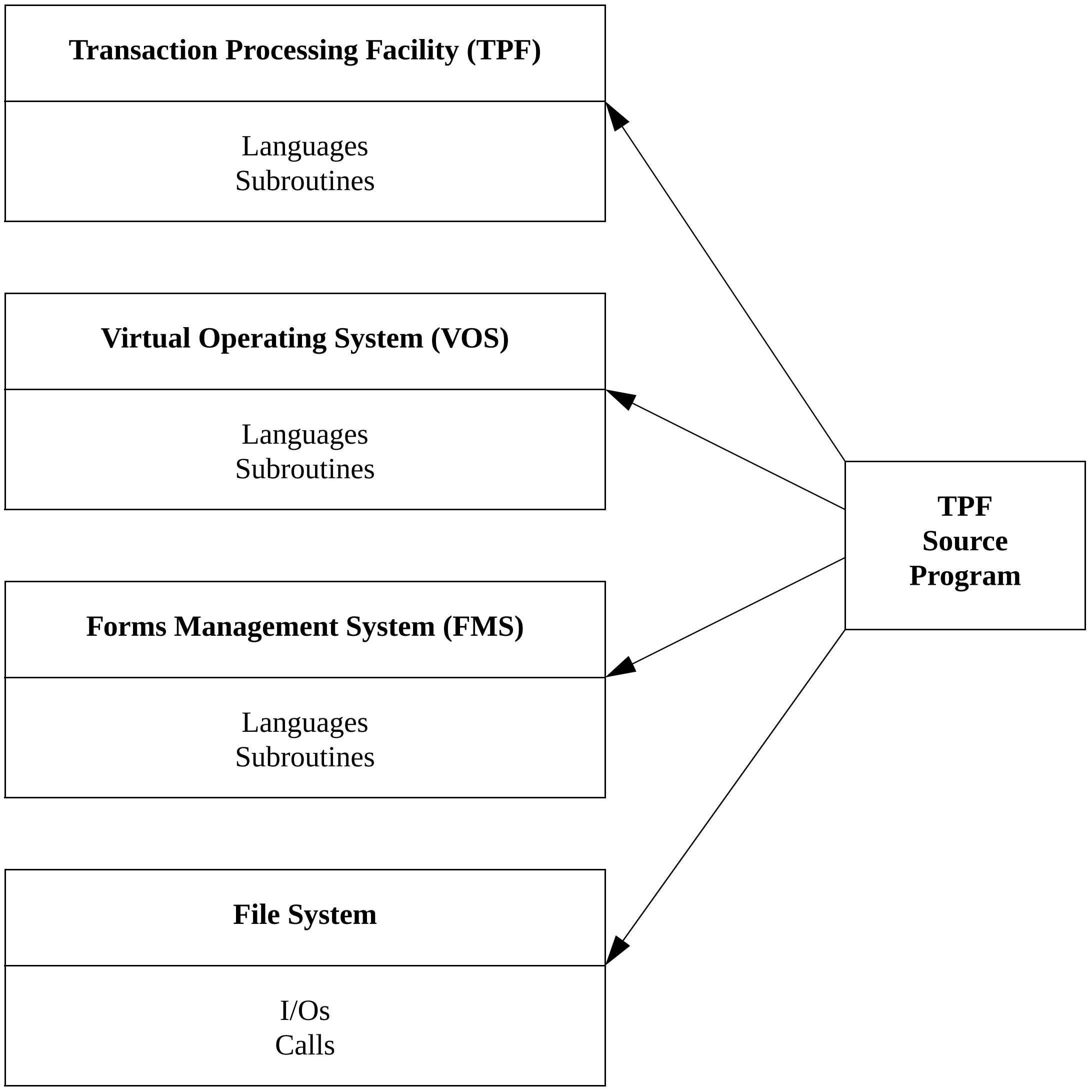

Transaction Processing Facility (TPF)
Languages
Subroutines
Virtual Operating System (VOS)
Languages
Subroutines
Forms Management System (FMS)
Languages
Subroutines
File System
I/Os
Calls
TPF
Source
Program

Figure 1-2. Typical Components of an OLTP Source Program

Error Messages

Application error codes are handled using the VOS error facilities. *VOS System Administration: Administering and Customizing a System (R281)* describes how to add new error codes and message text to a system's error table. The VOS Subroutines manuals document the subroutines available for checking error codes (status codes) and for displaying associated messages on terminals.

In addition to error messages, you can provide help messages and text for your application users. The Forms Management System allows explanatory text for each field on a displayed screen. You can also provide a menu system for longer descriptions. The *VOS Commands User's Guide (R089)* explains how to write new help system files.

All VOS subroutines, including the TPF subroutines, return a status code of zero when they execute successfully. A status code other than zero indicates an error that should be handled in your program.

Debugging

Several VOS debugging tools may be useful to the TPF programmer.

- The `debug` command. This command allows the programmer to follow program execution line-by-line and view the results at any point in the program. The command is documented in the *VOS Commands Reference Manual (R098)*. Two of its features that deal specifically with debugging tasking programs are:

  – the `env` argument with the `-task` attribute.

    The initial current environment is task 1. To debug a task other than task 1, use the `env` request with the `-task` attribute. For example, to debug task 3, enter:

    ```
    env -task 3
    ```

  – the `task_status` argument.

    This argument displays status information about either the current task or a specified task.

- The `mp_debug` command. This command is similar to `debug`, except that it debugs multiple processes. Most OLTP applications consist of several program modules, all executing at the same time in different processes. The `mp_debug` command is documented in the *VOS Commands Reference Manual (R098)*.

- The `analyze_system` command. This command provides many debugging and research capabilities, in the form of requests. The `list_transactions` request shows information about running transactions. The `list_transaction_trace` request provides information on committed or aborted transactions. The `dump_tdr` request shows task information. Other requests might also prove useful in debugging tasking programs and queue I/O. The `analyze_system` command is documented in the *VOS System Analysis Manual (R073)*.

The Requester/Server Design Model

TPF is especially suited for an application design model known as the requester/server model. This model is referred to frequently throughout this manual in examples illustrating various TPF features.

The requester/server model usually uses at least two program modules to run the online application. One program, called the *requester*, accepts user requests submitted through terminals and handles all interaction with the user. The requester sends user requests to a second program known as the *server*. The server program performs all data-file I/O. The server may or may not send messages back to the requester program. Usually, a queue file is the intermediary handling the messages between the requester and the server.

Figure 1-3 illustrates this requester/server design. The upper portion of the figure shows the basic model described above — one requester, one server, and one queue. The model can be expanded to multiple requesters, multiple servers, and multiple queues, or any combination of single and multiple components. The multiple requesters and servers can be the same program module executing many times in different processes, or each requester and server can perform different functions. The lower portion of Figure 1-3 shows a design that includes multiple server programs.

You do not have to use the requester/server design model for your application; the TPF capabilities work well with other designs. However, the requester/server design should be seriously considered, since it permits you to take full advantage of all the features offered by TPF.

The requester/server model is advantageous for the following reasons.

- The tasking capability is easily implemented in requester/server applications. Tasking saves memory, which is especially desirable on smaller modules.
- The server, which is performing only file I/O, is able to take advantage of the cache much more frequently than a program that is also performing terminal I/O. Since the server is not delayed by waiting for terminal I/O, it is more likely to find the data it needs in the cache, thus speeding up file I/O. (The *cache* is a place in memory where data is put as it is read from disk. Data remains in cache until the user no longer needs it. Even at that time, the data may remain in cache until another process needs the page. Cache pages are reused based on a least-recently-used algorithm.)
- On multimodule applications, it is usually faster to perform queue I/O across the network than it is to perform data-file I/O across the network. If the server programs and the data files reside on the same module, the server programs’ file I/O is local. Remote requester programs, which usually do not perform file I/O, can communicate efficiently with the queue across the network.
- Application development is simplified because the requester program functionality is separated from the server program functionality. Modifications are easier because of the modularity enforced by this division. For example, requester programs usually communicate with terminals and server programs usually perform the data file interface.
- On machines with multiple CPUs, true simultaneous processing may occur. Multiple CPUs servicing requester and server processes at the same time are faster than one CPU attempting to perform both kinds of functions within the same program. (Simultaneous processing refers to simultaneous servicing of processes, not to simultaneous servicing of tasks within a process. Only a single task within a tasking process executes at any given time. However, tasks in two different processes can execute simultaneously.)

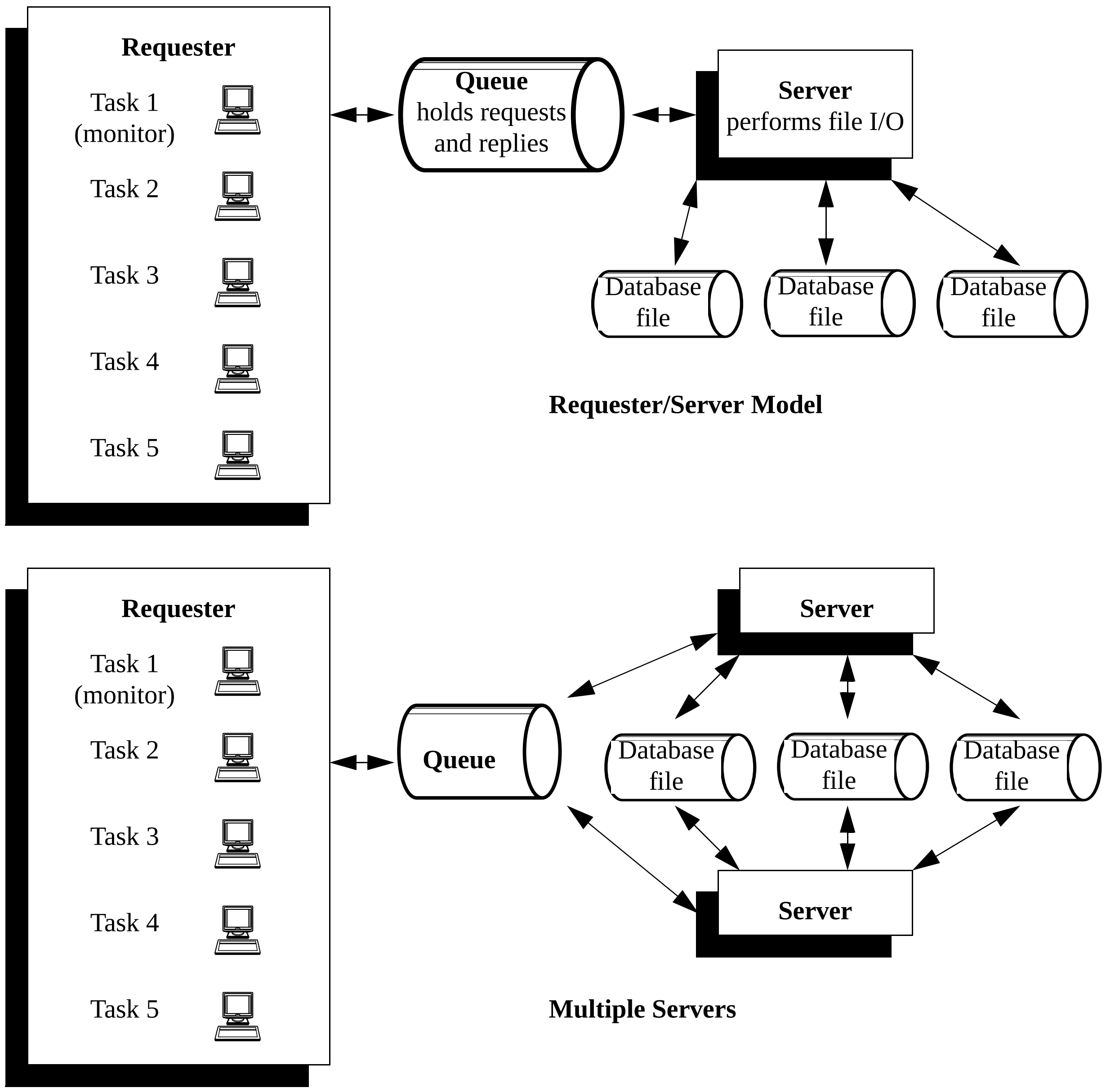


Figure 1-3. The Requester/Server Design Model

A Sample TPF Application

This section introduces a sample OLTP application that will be further developed throughout this guide, using TPF capabilities. The purpose of the sample application is to illustrate TPF features; it is not intended to illustrate a complete application. Appendix B, ‘‘Sample Application” contains program code, written in C, COBOL, and PL/I, implementing the system.

The sample application is an automated auction system. It permits buyers to submit bids for auction items from locations remote from the auction display site. The remote terminals let the users view the current highest bid submitted on auction items and submit their own bids for items.

The application uses the requester/server design model. It uses tasking to manage the end-user terminals, with a monitor task to administer the system. A two-way server queue transmits the I/O requests from the requester program to the server program. Requests for information and bid submissions from users are transactions requiring transaction protection.

Chapter 2: Tasking

This chapter describes multitasking capabilities. It begins with a brief discussion on the VOS process, followed by more detailed discussions of tasking, task states, task priorities and scheduling, and interprocess and intertask communication. A list of task-related subroutines is provided, and a sample tasking application is discussed. This chapter contains the following sections:

- ‘‘The VOS Process”
- ‘‘Tasking Overview”
- ‘‘Implementing Tasks”
- ‘‘Changing Task States after Startup”
- ‘‘Task Scheduling, Task Priorities, and Task Switching”
- ‘‘Interprocess and Intertask Communication”
- ‘‘Login vs. Slave Terminals”
- ‘‘Task-Related Subroutines”
- ‘‘Programming Considerations”
- ‘‘Sample Application”

For more information on the task-related subroutines, refer to the VOS Transaction Processing Facility Reference manuals.

The VOS Process

A VOS *process* is a collection of system resources that enables a program, macro, or VOS command to execute successfully. Some of the resources involved in a process are memory, default I/O attachments, CPU access, and access to the operating system. A process can be interactive or noninteractive.

A VOS process is started in any of the following ways.

- A user logs in. An interactive process is created whenever a user successfully logs in. The process is suspended (inactivated) if a subprocess is created from it, but it still exists.

- An interactive user creates a subprocess. A *subprocess* is an additional interactive process for the user. When a subprocess is created, it is active, and the process that created the subprocess is suspended (but still exists). A single main process can create many subprocesses, but only the most recently created subprocess is active.

- An interactive user requests a noninteractive process using the `start_process` command. This command creates a process that exists only long enough to execute the program, macro, or commands listed in the `start_process` command. The process executes independently of the interactive process that started it.

- An interactive user requests a noninteractive process using the `batch` command. When a user issues the `batch` command, the request is placed in a queue and executes in its turn. A VOS program called the *batch processor* handles batch requests and creates the processes that execute them.

System resources are assigned to a process when the process is created. The memory allocated to a process is a valuable system resource — a resource that can be exhausted if too many active processes are running on the module. Figure 2-1 illustrates the memory resources assigned to a nontasking process when the process is executing a program.

In Figure 2-1, the numbers in parentheses represent pages of virtual memory. One page of memory consists of 4,096 bytes. When two numbers are shown in parentheses, the larger number represents the number of pages in a G200 CPU board. The smaller number represents pages in boards offered before the G200 board was available. The total virtual address space available is either 8 MB or 64 MB, depending on the CPU type.

The map is not to scale. The user heap and the stack/heap expansion areas are both quite large compared with other areas shown.

In the figure, the term *wired* refers to pages of virtual memory that have physical pages associated with them. Unlike other virtual memory pages, wired pages are not swapped out of memory. A *fence* is virtual memory that is not addressable. A fence is positioned before a stack. If the stack expands enough to write in the fence, an error occurs. (Remember that a stack expands from a high address to lower addresses.) The purpose of the fence is to protect other addressable memory from being overwritten by an expanding stack.

The `logout` command terminates an interactive process. A subprocess is terminated when the `logout` command is issued from the current (active) subprocess. For example, if five subprocesses were created, you would have to issue five `logout` commands to return to the main process, and a sixth `logout` command would be necessary to completely exit from the system.

Noninteractive processes terminate when the program, command, or macro executing in the process stops running.

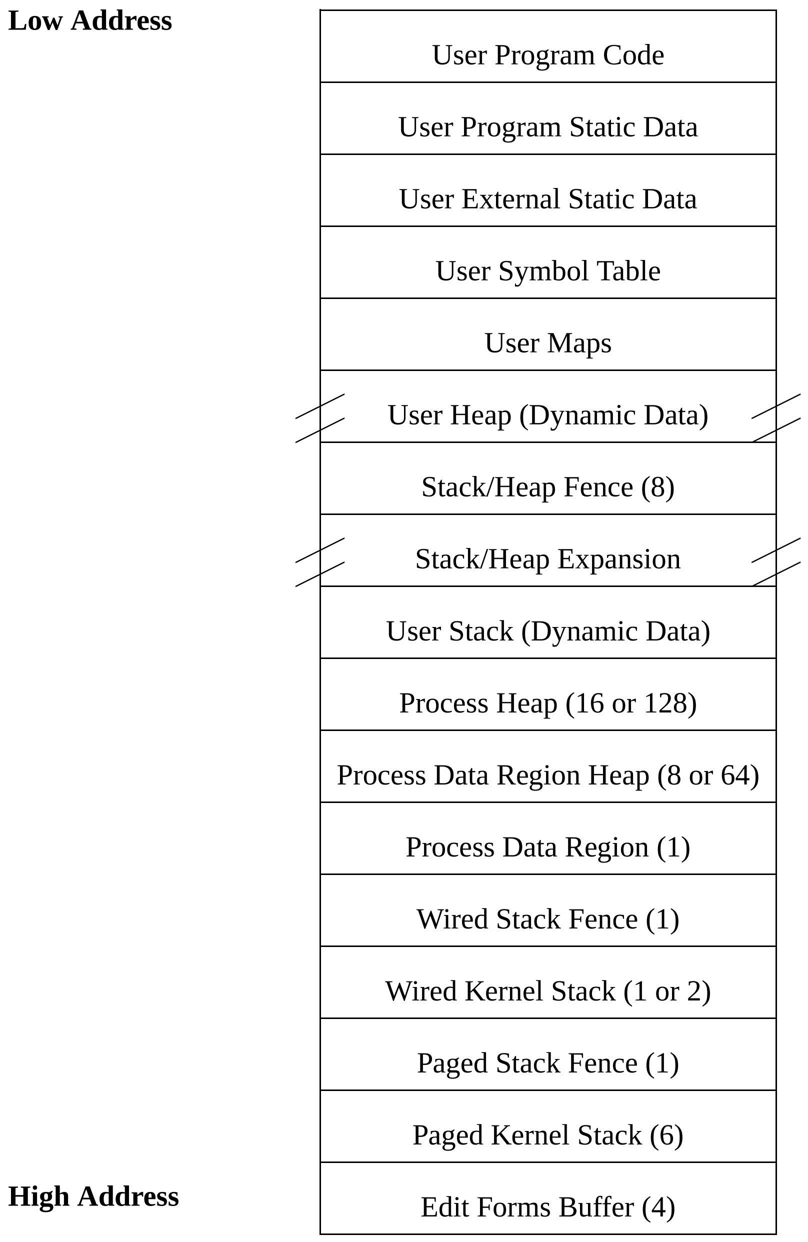


Figure 2-1. Virtual Memory Map for a Process Executing a Nontasking Program

Tasking Overview

This section provides an overview of tasking. It includes the following topics.

- ‘‘Definition of Tasking”
- ‘‘Memory Resources in a Tasking Environment”
- ‘‘I/O Ports”
- ‘‘Default Terminal Ports and Terminal I/O”

Definition of Tasking

The tasking facility allows a single process to manage multiple executing “threads” that proceed independently through the same program code. Each thread is a task, with its own data area. All tasks share the same program module, although they may or may not be executing the same section of code in the program module. In most applications, each task is associated with a separate terminal; however, tasks can exist without terminals, and tasks can share terminals.

A tasking program can be started from an interactive process or from a noninteractive process using the `start_process` or `batch` command. Any process can support multiple tasks.

For OLTP applications, tasking has several advantages over the one-process-per-terminal method. Tasking is desirable for the following reasons.

- Saved system resources. Many of the resources assigned to a process can be shared by all the tasks running in the process.

- Controlled entry and exit of users. The tasking terminals exist in a no-login, or *slave*, state. This means that the tasking terminal user cannot break out of the program, terminate the program, or do anything not explicitly allowed by the program. Most OLTP applications require the program, not the user, to be in control of the terminal. (The monitor task, if used, is an exception.)

- Effective use of queue I/O. Tasking is an efficient implementation of the requester side of a requester/server design model.

Memory Resources in a Tasking Environment

Memory Map

When a task executes, it uses some shared resources and some resources established only for it. Figure 2-2 shows how the memory resources are allocated for a process executing a tasking program. Compare this figure with Figure 2-1, which shows memory for a nontasking process. Note that most of the user memory areas are shared by all tasks in the process. Also, all of the process memory areas, the wired areas, the kernel, and the FMS forms buffer are shared by all tasks in the process.

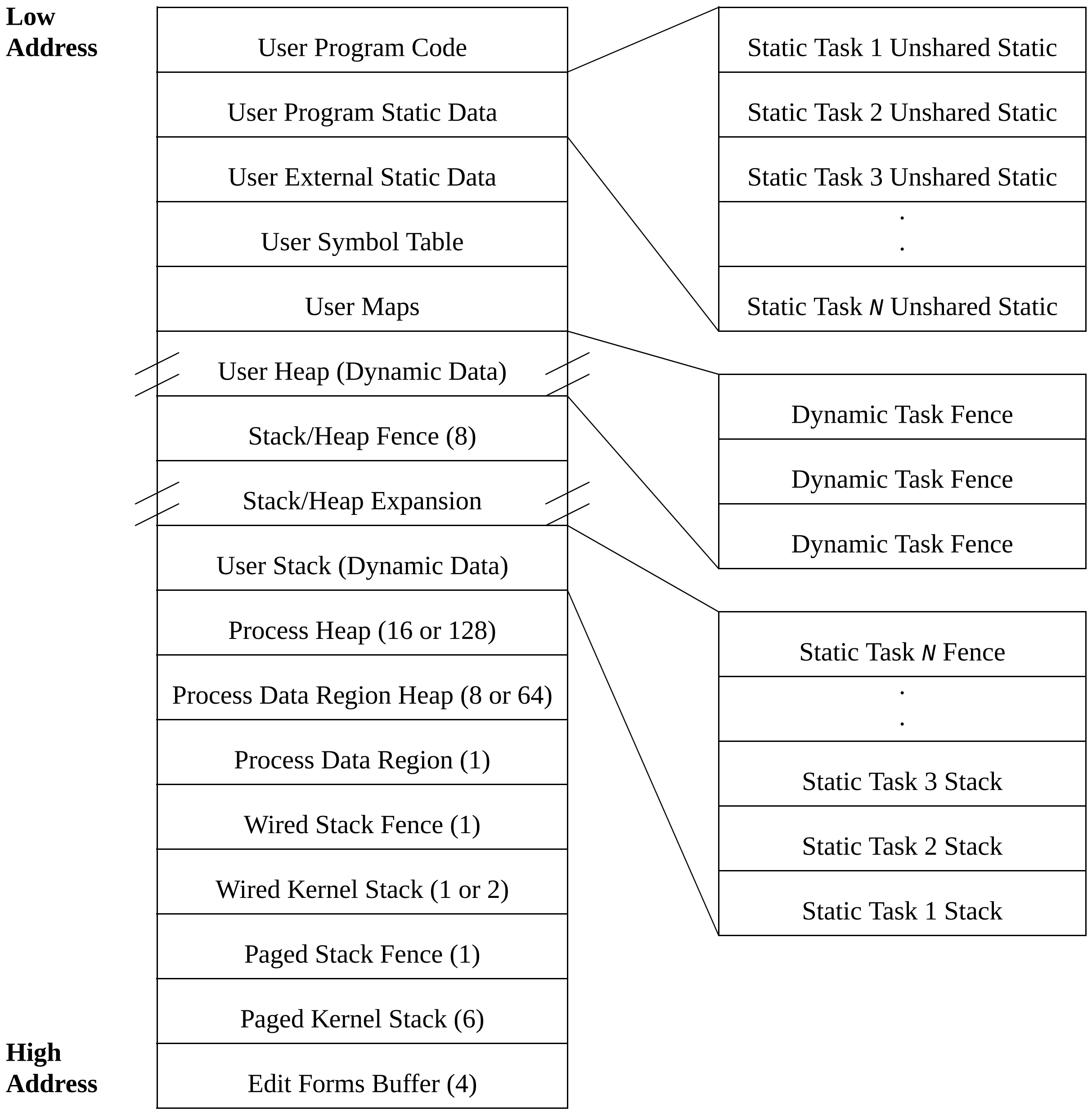


Figure 2-2. Virtual Memory Map for a Process Executing a Tasking Program

The following memory resources are not shared among tasks.

- Task stack. A task stack contains the list of return addresses generated by subroutine calls in the code executed by a single task. The task stacks maintain proper program flow for individual tasks. A task stack also contains *automatic variables* for the task. Automatic variables are variables that exist only while a called subroutine is active.

- Task static-data area. The task static-data area contains all *static variables* for a task (except for shared variables). Static variables are variables whose value is either initialized or retained across subroutine calls. Even when all tasks are executing the same code, each task maintains its own version of each variable in its static data area.

A program can specify that some static variables be *shared* among tasks in the the process. With a shared variable, all tasks access the same memory address to obtain the current value of the variable. Only one version of a shared variable is maintained in the program's external static data area.

There are two types of tasks — static and dynamic. The location of the stack and static data areas is different for the two types of tasks. *Static tasks* are created at bind time. Their stack and static data addresses are assigned as shown in Figure 2-2 and are explained below.

- The stacks for static tasks are assigned so that the lowest task number has the highest starting memory address. All static tasks have the same stack size. Therefore, the size must be large enough to accommodate the task with the largest stack requirements. You can specify the stack size in the binder control file, or you can let the stack size default to 32,768 bytes.

- A fence region is created between the last stack and the heap.

- The static data areas for static tasks are assigned so that the lowest task number has the lowest starting memory address.

The other type of task, called the *dynamic task*, is created during program execution. Dynamic tasks are assigned a static data area and a stack area located within the user heap area when the task is created. The stack size for the dynamic task can be specified when the task is created, or you can let the stack size default to 32,768 bytes.

A fence is allocated in front of each dynamic task's stack. Refer again to Figure 2-2. Note that static task stacks are adjacent to each other, and are not separated by individual fences. In effect, only the last stack (task *n*'s stack) has the protection of a fence when static tasks are used. In some situations, the benefit of one fence per task is a valid reason for creating dynamic tasks rather than static tasks.

The Task Data Region (TDR)

The *task data region* (TDR) is a data structure maintained only for tasking programs. It is located in the user heap, and contains information such as task state, task priority, task terminal's port ID, CPU time spent on the task, and task stack length. The information contained in the TDR is available to the application program through the use of the `s$get_task_info` subroutine. Likewise, this information is available to the monitor task with the `display_task_info` request.

I/O Ports

A *port* is a data structure that serves as the connection between the process and the input/output devices and files. Ports serve database files, queues, terminals, printers, and so on. A process can have up to 255 attached ports. The total number of ports for all the tasks running in a process is also limited to 255.

All ports, including ports to I/O devices, files, and queues, can be shared among the tasks in a process. To share a port, the 16-bit variable that contains the port ID must be declared as a shared variable in all of the source code modules that reference it. The ‘‘Programming Considerations” section at the end of this chapter includes a description of how to declare shared variables.

Default Terminal Ports and Terminal I/O

When the operating system creates a process, it attaches four default ports to the process. In a tasking process, these four ports are shared by all tasks in the process. The ports are used as input and output for VOS commands entered on the terminal and for related system error messages. Also, two of the ports are used for default terminal I/O during program execution. (Your program performs terminal I/O using those ports. You reference the ports through language-specific methods described below.)

The following list shows the port names of the four process default ports and briefly describes the purpose of the ports.

- `default_input`. Input data required by VOS commands is transmitted on this port. Also, this port is the default port for terminal input.
- `default_output`. The operating system writes output data from commands to the terminal on this port. Also, this port is the default port for terminal output.
- `terminal_output`. The operating system transmits system error messages to the terminal on this port.
- `command_input`. The operating system commands are transmitted on this port.

Process terminal I/O is redirected from these four ports to another port as follows:

- In nontasking interactive processes, terminal I/O is redirected to port 5, which is the port attached to the process terminal.
- In nontasking noninteractive processes, terminal input is an error. Terminal output is redirected to a file that is usually attached to port 6. If port 6 is already in use, another port is attached to the file. The default name for this output file normally has a suffix of `.out`.
- In tasking processes, task terminal I/O is redirected to the port attached to the task’s terminal. The task’s terminal is the one assigned to the task during task initialization. (Sometimes, the task terminal is a terminal reassigned to the task by a call to the `s$set_process_terminal` subroutine.) When the task’s terminal specification is a null string, there is no terminal to which the default I/O can be redirected. In that case, an error occurs if default I/O is attempted by the task.

  For example, in a process with five tasks, there is one port named `default_input`, and all five tasks can reference `default_input`. For each task, data requests on `default_input` are redirected to the port attached to the task’s terminal.

  I/O is **not** redirected through a terminal that was attached from within the task using the `s$attach_port` subroutine. Those ports are direct, and are independent of the default I/O capabilities discussed here. The task handles I/O through its direct ports by naming the port in I/O calls. The task cannot use the language statements described below to perform default terminal I/O through direct ports. Direct ports can be shared among tasks.

The following paragraphs describe how to achieve terminal I/O using the `default_input` and `default_output` ports in each of the VOS languages.

BASIC

Channel 0 refers by default to input and output to and from the terminal. The `print` and `print_using` statements output text to a terminal. The `input` and `linput` statements input text to a terminal. Channel 0 is the default if no channel number is used with these statements.

C

Many I/O functions, such as `fprintf`, write to and read from `stdout` and `stdin`. `stdout` is attached to the `default_output` port, and `stdin` is attached to the `default_input` port.

COBOL

To receive input from the terminal, use either a `read` statement with a file name of `default_input` or the `accept` statement. To send data to the terminal, use either a `write` statement with a file name of `default_output` or the `display` statement.

FORTRAN

Units 5 and 6 are, by default, attached to the `default_input` and `default_output` ports, respectively.

A `read` statement with no `unit` specifier or `unit` equal to `*` receives input through unit 5. A `write` statement with `unit` equal to `*` sends output through unit 6. The `print` statement has no unit specifier. Connections with `print` are always to unit 6.

Pascal

The Pascal default `input` file is attached to the `default_input` port. The Pascal default `output` file is attached to the `default_output` port.

PL/I

The PL/I `sysin` file is attached to the `default_input` port. The PL/I `sysprint` file is attached to the `default_output` port. Therefore, use the `get sysin` and `put sysprint` statements to perform terminal I/O.

Implementing Tasks

This section describes how tasks are implemented. It includes the folowing topics.

Task States

A task exists in any one of eight states. The current state is identified by a one-digit code, as explained below. The source program can determine the current state of a task by calling the `s$get_task_info` subroutine. The monitor task user can obtain the current state of a task with the `display_task_info` request. The task states are as follows:

0 - uninitialized (created)

All tasks begin existence in the uninitialized (created) state. An uninitialized task has been allocated memory, but it has not been assigned an execution entry point in the program module, and it has not been attached to an I/O device (such as a terminal).

1 - initialized

The variables in the task’s static area are initialized according to the programming language’s initialization conventions. The contents of the automatic data area in the task’s stack is undefined. The task might have a terminal or other I/O device associated with its task port. If so, the device is open and ready for I/O.

2 - ready

The task is waiting for the VOS task scheduler to give it CPU time. An execution entry point is associated with the task.

3 - running

The task is executing code. Only one task can run at a time in a process.

4 - waiting

The task was running, but has been forced to wait for something. For example, it might be waiting for input from a user. When a task state is changed from running to waiting, a task switch occurs. The task switch allows another task in the process to use the CPU.

5 - paused-waiting

The task was waiting and was intentionally paused either by a call to `s$control_task` or by the monitor task user invoking the request `control_task`.

6 - paused-ready

The task was either running and intentionally paused as in state 5, or it was in the paused-waiting state and would become a ready task if it was not paused.

7 - stopped

The task was started and then stopped because it was no longer needed. It can be started again at a valid entry point, or it can be uninitialized.

Tasks move from state to state as a result of subroutine calls, monitor task requests, or VOS actions. For example, a task changes from an initialized state to a ready state as a result of a call to the `s$start_task` subroutine or the `start_task` monitor request. An example of a VOS-controlled change in a task state is the change from running to waiting. Various sections later in this chapter describe how to create, initialize, start, and stop tasks.

Figure 2-3 shows the task states and how they are changed. The figure implies a sequence of states that must be maintained. For example, a task cannot change from an uninitialized state to a running state without first being in the initialized and ready states.

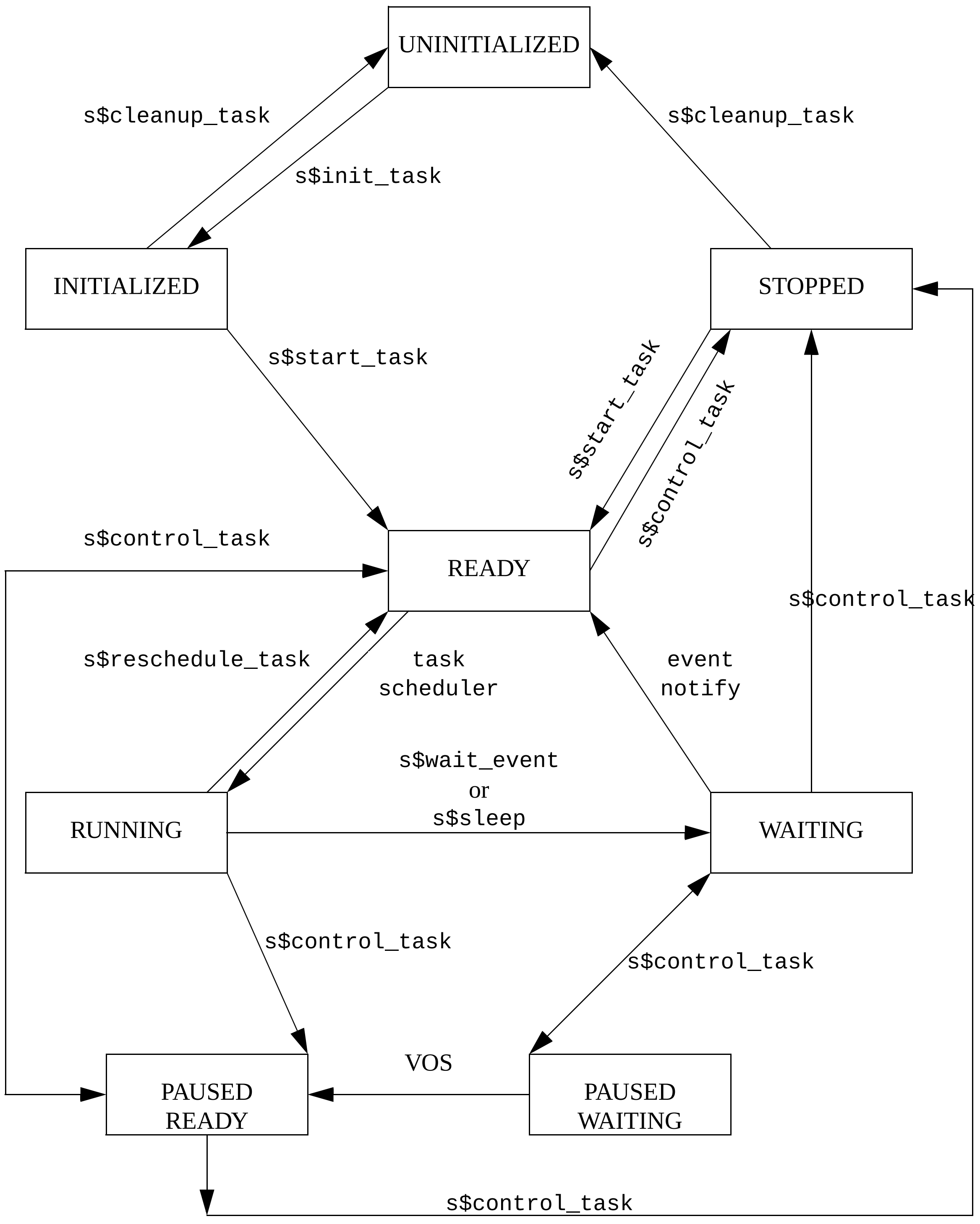


Figure 2-3. Changing Task States

The Task ID

Tasks are identified by a task ID. Except for task 1, you specify the task ID when you create the task. See ‘‘The Primary Task and the Process Terminal” below for a discussion about task 1.

The task ID must be an integer, unique within the process.

The task ID is a required input variable for many tasking subroutines. However, most subroutines requiring the task ID as an input variable accept the value of 0 to indicate the current (calling) task. This feature is useful in sections of common code executed by multiple tasks.

Another convention acceptable to many subroutines is the value -1 for task ID, which indicates all tasks in the process. This value means that all tasks in the process, not just the calling task, are affected by the subroutine call.

If the actual task ID value is required, it can be obtained for the current (calling) task through the `s$get_task_id` subroutine.

The Primary Task and the Process Terminal

When a tasking program begins execution, it starts in task 1 by default. There is no way to alter this default. This first task is called the *primary task*. It has a task ID of 1.

The primary task is attached to a terminal called the *process terminal*, on a port called the `process_terminal` port. The process terminal is the terminal from which the command that started the program was issued. The `process_terminal` port is a unique port in the process.

Other tasks can also use the `process_terminal` port. When other tasks are initialized, the `process_terminal` can be specified as the terminal port for the new tasks. Tasks that are attached to other terminals can temporarily use the `process_terminal` port by calling the `s$set_process_terminal` subroutine.

There are no other differences between the primary task and other tasks. Any task can be the monitor task, and any task can initialize, start, and stop other tasks in the process.

Program Flow

When a tasking program begins execution, it executes in task 1 by default. Execution begins at the entry point specified in the `entry:` directive in the binder control file. Additional tasks are implemented by subroutine calls made from task 1 or by monitor requests issued from the monitor task.

To implement additional tasks, the sequence of events is:

- create the task
- initialize the task
- start the task

An execution entry point is assigned to a task when the task is started. Valid entry points into the tasking program are identified by various directives in the binder control file. A valid task entry point is defined in either of the following ways:

- the name of any of the separately compiled object modules listed in the `modules:` directive
- the name of any entry point or subroutine contained within one of the object modules and listed in the `retain:` directive.

A task is usually associated with a terminal, and, through the terminal, to a user. The application can assign functions to tasks in the following ways.

- All tasks (that is, all users) perform the same function. All tasks are assigned the same entry point in the program module, and all tasks execute the same section of code.
- From a menu screen, users choose the function they want to perform. All tasks are assigned the same entry point in the program module, but they can execute different code based on the choices the users make from the menu.
- Each user is limited to a specific, but different, function or set of functions. Each task is assigned a different entry point in the program module.
- Any combination of the above methods is also possible.

Creating Static and Dynamic Tasks

To *create* a task means to allocate memory for the task. After creation, a task exists in an uninitialized state.

There are two types of tasks — static tasks and dynamic tasks. The two types of tasks are created differently. Once created, there is no difference in how they execute.

A *static task* is created by the `bind` command when the program module is bound. This means that memory resources are established in the program module at bind time. Directives in the binder control file establish the number of tasks to be created and the size of the task stacks. When program execution begins, static tasks exist in the uninitialized state.

Static tasks cannot be deleted. The memory resources allocated for them at bind time remain assigned until the program stops executing.

A *dynamic task* is created at any time during program execution. The `s$create_task` subroutine and the `create_task` monitor request create dynamic tasks. The new task exists in the uninitialized state.

The operating system creates a static area for a dynamic task by copying the primary task's static area into the program module's user heap and appropriately adjusting address references. Therefore, dynamic task creation requires CPU time that is not required by static tasks. However, dynamic tasks require less disk space than static tasks. Refer back to the memory map in Figure 2-2 to compare memory space used by dynamic and static tasks.

A dynamic task may be created in a process that was not using tasks previously. The operating system takes care of the overhead involved in changing the process from a nontasking process to a tasking one.

When dynamic tasks are no longer needed, they can be deleted during program execution with the `s$delete_task` subroutine or the `delete_task` monitor request. The task must be in the uninitialized state to be deleted. When a dynamic task is deleted, the heap allocated for the task's stack, fence, and internal static data area is released. However, heap storage allocated by the task is **not** freed, and ports attached or opened by the task are **not** closed and detached. A task epilogue handler is recommended for accomplishing these cleanup activities. See the ''Stopping Tasks and Invoking Epilogue Handlers" section later in this chapter.

Initializing Tasks

Task initialization attaches a task to a terminal by attaching a terminal port. Tasks need not be attached to terminals. However, tasks that are not attached to terminals must still be initialized using the null string (the literal value `''`) for the terminal name. Note that this is **not** the #null pseudodevice.

Task initialization also initializes the variables in the task's static data area in accordance with the programming language's initialization conventions. Each time a task is initialized, the variables in the task's static data area contain the same values that they would contain when execution begins in a nontasking program. As with nontasking programs, you can assign specific initial values to variables, either in the source program or in the binder control file. The binder control file assignment overrides any assignment in the source program.

The automatic data areas in the task's stack are undefined after each initialization.

There are various ways to achieve task initialization.

- List the task/terminal associations in a task configuration table, and call the `s$init_task_config` subroutine from the primary task. All tasks listed in the task configuration table are initialized by the subroutine.
- List the task/terminal associations in a task configuration table, and depend on the user of the monitor task to submit the `initialize_configuration` request. This leaves the monitor task user in charge of task startup.
- From the primary task, call the `s$init_task` subroutine individually for each additional task.
- Depend on the user of the monitor task to submit the `initialize_task` request for each task.
- A started task may start other tasks.
- A combination of the above is possible, where some tasks are initialized using the task configuration table, and others are initialized individually later in the program.

Configurations are easily changed during program execution by using the `s$init_task` subroutine or the `initialize_task` monitor request. Changes and additions might be required to create specialized tasks or to replace malfunctioning terminals.

When a tasking program module begins execution, the primary task (task 1) is executing by default. Therefore, the primary task does not have to be initialized and started. The primary task can then call `s$init_task_config` to initialize and start other tasks; however, any task can call `s$init_task_config`.

> **Note:** If the task ID of the task that calls `s$init_task_config` is listed in the task configuration table, then the call to `s$init_task_config` reinitializes and restarts the calling task at the entry point listed in the task configuration table. In that situation, control never returns to the point where `s$init_task_config` was called. As long as the task configuration table does not include the calling task's task ID, control returns to the point where `s$init_task_config` was called, and the task continues execution. This concept is illustrated in Figure 2-5, which appears later in this chapter.

If you reinitialize task 1, you must reinitialize it using one of the following values for *`terminal_name`*:

- The literal value `(process_terminal)`. The process terminal is the terminal from which the command to start the tasking program was issued. When this value is used, task 1 is attached to the terminal that started the tasking program.
- The literal value `(task_terminal)`. The task terminal is the terminal attached to the task that is performing the initialization. When this argument is used, task 1 is attached to the calling task's terminal.

  Note: Do not initialize (or reinitialize) task 1 using a specific terminal path name, even if the path name correctly identifies the same terminal attached on the `process_terminal` port. By naming a specific terminal path name, you attach a new terminal I/O port. (The original port that was attached to task 1 when the program started, the `process_terminal` port, is not detached.) If task 1 has two terminal I/O ports, results might be unpredictable.

Attaching Terminals to Tasks

All the task initialization methods described above include a *`terminal_name`* argument. This argument specifies which terminal should be attached to each task when the task is initialized. Valid values for the *`terminal_name`* argument are as follows:

- a full terminal path name in the form *`%system_name#terminal_name`*.
- null. Use the null string (`''`) in the task configuration table. Use blanks in the `initialize_task` monitor task request. Null means that no terminal is attached to the task. A task without a terminal cannot perform terminal I/O.
- the literal values `(process_terminal)` and `(task_terminal)` are valid in the task configuration table for task 1.

The Task Configuration Table

You must create the task configuration table for the application if you want to use the `s$init_task_config` subroutine or the `initialize_configuration` monitor request. There are three steps to creating this table.

1. Ensure that a data description (`.dd`) file for task configuration exists on your module. Create one, if necessary.

   This file, which must have a suffix of `.dd`, contains the format of the table that is created in step 3. The file's contents are fixed and must be written exactly as shown below. (The operating system expects the data types in the `.dd` file to look like PL/I data types. The starting position of each line is not important.)

   ```
   fields:
             task_id              bin (15),
             entry_name           char (32) varying,
             terminal_name        char (66) varying;
   end;
   ```

2. Create a table input (`.tin`) file.

   This file, which must have a suffix of `.tin`, contains the data that is loaded into the table created in step 3. There must be an entry in the `.tin` file for each task to be initialized. Figure 2-4 shows the format of the task configuration `.tin` file and a sample file for three tasks.

3. Create the actual table by issuing the `create_table` command from command level.

   This command builds the task configuration table, which is a structured record file. To change its contents, make the change in the `.tin` file, and then reissue the `create_table` command.

   The `create_table` command requires the path names of the `.tin` file as input. The file that it creates has a suffix of `.table`. This `.table` file is the input used in the `s$init_task_config` subroutine.

   If you do not specify otherwise, the `create_table` command assumes that the `.dd`, `.tin`, and `.table` files all have the same prefix. For example, your files are named:

   `sample.dd` and `sample.tin`

   You issue the command:

   ```
   create_table sample
   ```

   The output file will be named:

   ```
   sample.table
   ```

   The full version of the `create_table` command for the above example is:

   ```
   create_table sample.tin sample.table -data_description sample.dd
   ```

Format:

```
/ =task_id       number
=entry_name      entry_point
=terminal_name   terminal_name
```

where:

`number`	is the number that identifies this task.
`entry_point`	is a valid entry point in the program module where this task begins execution; this entry point must be listed under either the `modules:` or the `retain:` directive in the binder control file used to create the program module.
`terminal_name`	is usually the full path name of the terminal to be attached to this task, in the form `%system_name#terminal_name`. Use the null string if no terminal is to be attached. For task 1, two other values are acceptable: the value `(process_terminal)`, and the value `(task_terminal)`.

In the example below, tasks 2 and 3 are initialized to specific terminals. Both tasks start execution at the same entry point. Task 15 is not attached to a terminal; presumably, its output (if any) is written to a file. Task 25 is attached to the terminal that started the tasking program (the process terminal). Since task 1 is not reinitialized in the example, task 1 is attached to the process terminal by default.

Example:

```
/ =task_id        2
=entry_name       common_entry
=terminal_name    %Riverside#t2.7

/ =task_id        3
=entry_name       common_entry
=terminal_name    %Riverside#t2.8

/ =task_id        15
=entry_name       batch_metering
=terminal_name    ''
```

Figure 2-4. Task Configuration Table Input (`.tin`) File

Starting Tasks

A task that has been started waits for CPU time to begin execution. Execution begins at the entry point supplied when the task was started. There are various ways to start a task.

- The task is already started if the `s$init_task_config` subroutine was called or if the `initialize_configuration` monitor request was used. The subroutine and the monitor request each initialize and start tasks listed in the task configuration table. The task configuration table contains the entry point for each task.
- A task can be started by a call to the `s$start_task` or `s$start_task_full` subroutine. The task's entry point is provided in the subroutine call. The `s$start_task_full` subroutine permits the calling task to pass arguments to the initial routine of the starting task.
- A task can be started by the monitor task with the `start_task` monitor request. The entry point is an argument of this request.

Static tasks are started by any of the methods described. Dynamic tasks must be started by the second and third methods.

> **Note:** The `s$init_task_config` subroutine puts tasks in the ready state. The started tasks cannot go from ready to running until the calling task does something to allow a task switch. A call to `s$reschedule_task` or `s$sleep` would cause a task switch.

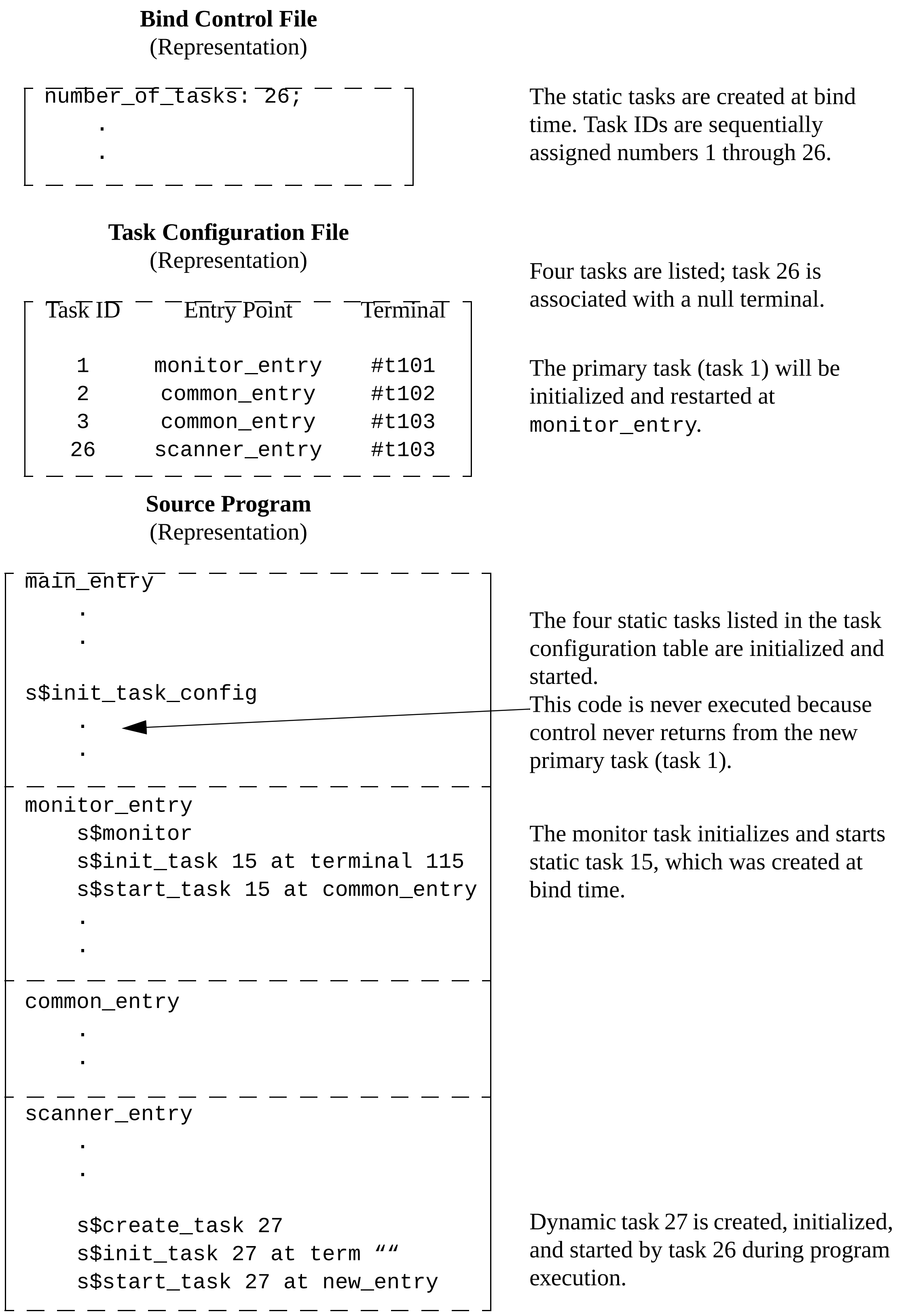
The `s$start_task` subroutine and the `start_task` monitor request use an argument with an alphanumeric data type to specify the starting point for the new task. The value used for the *`entry_point_name`* argument must exist in the entry map built by the `bind` command. This means that the *`entry_point_name`* value must be listed in either the `modules:` or `retain:` directive of the binder control file. The operating system gets the beginning execution statement address for the new task from the entry map.

See the ‘‘Binding Tasking Programs’’ section later in this chapter for information about the binder control file directives.

The `s$start_task_full` subroutine uses an argument with an entry data type to specify the starting point for the new task. Since entry variables contain addresses that point to the calling task's static data area, they must not be declared as shared. If multiple tasks call `s$start_task_full` to indicate the same entry point, each task must obtain its own value for the entry variable. See the ‘‘Entry Values’’ section later in this chapter for a description of the entry data type.

Example for Creating, Initializing, and Starting Tasks

Figure 2-5 illustrates a scenario for creating, initializing, and starting dynamic and static tasks.



Bind Control File
(Representation)

```
number_of_tasks: 26;
    .
    .
```

The static tasks are created at bind time. Task IDs are sequentially assigned numbers 1 through 26.

Task Configuration File
(Representation)

Task ID	Entry Point	Terminal
1	monitor_entry	#t101
2	common_entry	#t102
3	common_entry	#t103
26	scanner_entry	#t103

Four tasks are listed; task 26 is associated with a null terminal.

The primary task (task 1) will be initialized and restarted at `monitor_entry`.

Source Program
(Representation)

```
main_entry
    .
    .

s$init_task_config
    .
    .
```

The four static tasks listed in the task configuration table are initialized and started.
This code is never executed because control never returns from the new primary task (task 1).

```
monitor_entry
    s$monitor
    s$init_task 15 at terminal 115
    s$start_task 15 at common_entry
    .
    .
```

The monitor task initializes and starts static task 15, which was created at bind time.

```
common_entry
    .
    .
```

```
scanner_entry
    .
    .

    s$create_task 27
    s$init_task 27 at term ""
    s$start_task 27 at new_entry
```

Dynamic task 27 is created, initialized, and started by task 26 during program execution.

Figure 2-5. Creating, Initializing, and Starting Tasks

Changing Task States after Startup

This section describes how to change task states after a task is running. It includes the following topics.

- ‘‘The `s$control_task` Subroutine”
- ‘‘Stopping Tasks and Invoking Epilogue Handlers”
- ‘‘Reinitializing and Restarting Tasks”

The `s$control_task` Subroutine

The usual sequencing of task states is: uninitialized, initialized, ready, and running. Once a task is running, it may change to the ready, the waiting, the paused ready, the paused waiting, or the stopped state. As implied by Figure 2-3, these states are a result of either an operating system action or a program call to the `s$control_task` subroutine.

Refer to the detailed explanation of `s$control_task` in the VOS Transaction Processing Facility Reference manuals for more information on controlling task states.

Stopping Tasks and Invoking Epilogue Handlers

Although a stopped task is no longer executing, it still exists. A task is stopped by calling the `s$control_task` subroutine with the `stop_task` action. The task stops as a result of this call, but it may not stop immediately. It first executes task epilogue handlers, if any exist. It then detaches its terminal. Both of these actions may require the task to wait. Therefore, the task’s state may change several times between ready, running, and waiting before it is changed to stopped.

> **Note:** Usually (and preferably), a stopped task does not regain control of program flow or execution. The `stop_task_return` action in `s$control_task`, which would allow the task to return control to itself, is not recommended for new applications because of its incompatibility with other new TPF features.

An implicit task stop occurs when the first user entry in the task stack returns to its caller. The first user entry is `start_user_program`. It is placed in the stack when the task is started.

A task should not call the C language `signal` function because the `signal` function depends on a stack frame associated with a call to `s$start_c_program.` In a task’s stack, the entry `start_user_program` replaces `s$start_c_program`. To handle signals in a task, use the VOS subroutine `s$enable_condition` instead.

A stopped task may be cleaned up, started, or initialized and started from another task. A start with initialization reinitializes the task’s static area instead of using old values.

By calling the `s$set_process_terminal` subroutine, a task can switch its terminal association back and forth between the task’s terminal and the process terminal without stopping. The task’s static area is not affected by this subroutine. (The process terminal is the terminal from which the tasking program was started.)

Two followup actions that are often appropriate on stopped tasks are:

- task cleanup
- task epilogue handlers.

Task cleanup closes and detaches the task's terminal port and puts the task in the uninitialized state. It also frees all storage allocated by the operating system for the task. Task cleanup is accomplished with the `s$cleanup_task` subroutine, the `cleanup_task` monitor request, or the `stop_task_cleanup` action in the `s$control_task` subroutine.

Task cleanup does not free space that the task explicitly allocated for itself during execution. That type of allocation uses space in the user heap, and the user heap is allocated to the process, not to individual tasks. If a task allocates storage in the user heap, then the task should free up that space when the task ends. A task epilogue handler is recommended to perform the cleanup.

Task epilogue handlers are programmer-defined routines that are performed immediately after a task is stopped. Explicit calls are not required in your program code to invoke epilogue handlers; the `stop_task` action always passes control to the task's list of epilogue handlers.

> **Note:** On abnormal terminations, the task epilogue handlers cannot execute if the task stack was destroyed. Neither task nor program epilogue handlers are executed when the `stop_process` command or `s$stop_process` subroutine is used.

The `s$add_task_epilogue_handler` and `s$delete_task_epilogue_handler` subroutines add to, and delete from, the list of task epilogue handlers. Each task can have up to 10 epilogue routines associated with it.

Task epilogue handlers are different and separate from program epilogue handlers. Program epilogue handlers are also programmer-defined routines, but they apply to the entire program module instead of to just one task. The program module may have up to 10 program epilogue routines that execute when the program is stopped with the `s$stop_program` subroutine. If the program stops while tasks are still active, the following occurs.

- All task epilogue handlers execute, and then all tasks are stopped.

- Task cleanup is performed for all tasks. This closes and detaches all task terminal ports.

- Task 1 is started again to run the program epilogue handlers. The terminal output ports for the program epilogue handlers are the ones in effect when the program started.

The following are some actions that you may want task epilogue handlers to perform:

- freeing storage that the task explicitly allocated
- closing files
- deleting temporary files
- detaching unshared ports to unshared files and queues, when those files and queues were opened with the `hold_open` and `hold_attached` options

Reinitializing and Restarting Tasks

A task is reinitialized by initializing it again. A task is restarted by starting it again.

A task must be uninitialized before it can be initialized (or reinitialized). Therefore, to reinitialize a started task, you must first stop the task and clean it up. Then you can reinitialize it using any of the methods described previously for initialization. The reinitialization clears the task stack and sets all the variables in the task's static data region to undefined, except for the variables that are assigned initial values.

A stopped task may be restarted without being reinitialized. A task is restarted by any of the methods described previously for starting a task. The task restarts at the named entry point.

Task Scheduling, Task Priorities, and Task Switching

Task scheduling refers to the order in which tasks are selected for execution when more than one task is in the ready state. *Task priority* refers to a scheme that selects tasks to run based on their urgency. *Task switching* refers to the situation that occurs when a running task stops running and another task begins to run in its place.

You do not have to handle task scheduling in your program. An operating system program called the task scheduler performs the scheduling according to default rules. However, you can override the default scheduling by assigning priorities to the tasks, and by rescheduling tasks that are already running.

Various subroutines allow the programmer to directly or indirectly control task switching. Otherwise, task switching is controlled by the task scheduler.

This section contains the following topics.

- ‘‘The Task Scheduler”
- ‘‘Assigning Priorities”
- ‘‘Rescheduling Already-Running Tasks”
- ‘‘Task Switching”
- ‘‘Preventing Task Switching”

The Task Scheduler

The task scheduler takes tasks that are in the ready state and puts them in the running state, according to the following rules.

- The task with the highest priority value is selected first.

- If multiple tasks have the same priority value, those tasks are scheduled based on a first in, first out order. Among tasks of the same priority, the one that has been ready to run the longest is selected first.

- If an application does not assign priorities to its tasks, then all tasks have priorities of 0, and selection is based on the order in which the tasks were put in the ready state.

The application must take into account the possibility that a low-priority task may wait a long time before being scheduled if there are many higher priority tasks continuously in the ready state.

Figure 2-6 illustrates this scheduling scheme.

If the ready queue looks like this:

	Task ID	**Priority**
first in	5	0
	8	0
	2	8
	6	19
	4	19
last in	1	0

Order of selection for execution is:

Task 6
Task 4
Task 2
Task 5
Task 8
Task 1

Figure 2-6. The Task Scheduling Scheme

Assigning Priorities

A task priority is maintained for each task. Its value is between 0 and 255, inclusive. Higher numbers indicate greater urgency. The default priority on a newly created static or dynamic task is 0. This value is changed by a call to the subroutine `s$set_task_priority` or by the `set_task_priority` monitor request issued from the monitor task.

A task priority can be changed while the task is in any state. The only requirement for assigning or changing a priority is that the task must exist (that is, it must have been created). A priority change on a running task may cause a task switch. The running task is preempted by a higher priority task in the ready state.

The current priority of any task is obtained with the `s$get_task_info` subroutine. The monitor task views the priorities of tasks with the `display_task_info` monitor request.

Rescheduling Already-Running Tasks

The `s$control_task` and `s$reschedule_task` subroutines allow customized task scheduling.

The `s$control_task` subroutine allows the application program to pause, to continue tasks, and to stop tasks completely.

The `s$reschedule_task` subroutine permits the application program to request a task switch. When this subroutine is called, the task scheduler puts the calling task in the ready state with the same priority it had previously. The task is rescheduled for continued execution according to the scheduling rules described earlier. If no other task is in the ready state when the `s$reschedule_task` subroutine is called, then the calling task continues execution without interruption.

Rescheduling is useful for breaking up long-running tasks so that other tasks can execute.

> **Note:** The `s$reschedule_task` subroutine does **not** allow other lower priority processes to gain control of the CPU. The subroutine causes only a task switch, not a process switch.

Task Switching

A task switch usually occurs when a running task is put in a wait state. This frees the CPU, allowing another ready task to begin running. When the condition for the wait is satisfied and the previously running task is put back in the ready state, it is placed at the end of the ready queue. The task keeps the same priority value that it had before the wait, and it is chosen for execution according to the rules described previously. If no task is ready, the process relinquishes control of the CPU. Note that process switching is faster than task switching.

A running task is put into the wait state, thereby causing a task switch if any of the following are true.

- It must wait for a lock related to an I/O operation. The task is put in the wait state until it receives the lock; then it is put in the ready state. File I/O not requiring a wait does not cause a task switch.
- It calls the `s$sleep` or `s$wait_event` subroutine. These subroutines put the task in the wait state.
- It is waiting for terminal or file I/O.

The following actions might cause a task switch:

- the `s$control_task` subroutine
- the `control_task` monitor request
- the `s$reschedule_task` subroutine

Preventing Task Switching

The application program can prevent tasking, and, therefore, task switching, with the `s$enable_tasking` subroutine. This subroutine allows a task to disable task switching in its process, so that the task can exclusively use a device that otherwise would be shared among a set of tasks. For example, the `s$enable_tasking` subroutine can prevent task switching while the calling task waits for device I/O.

The same subroutine is used to reinstitute tasking. Paired calls to `s$enable_tasking` are essential.

The `s$set_process_terminal` subroutine also disables and re-enables tasking.

Interprocess and Intertask Communication

Processes and tasks must be able to communicate with each other. Communication is required in an application where a process or task must wait for a particular event to happen in another process or task before it can proceed. Communication is also required in a requester/server application in which one process/task might wait for a prompt or message from another process/task to begin or continue its processing.

The operating system provides the following methods for interprocess and intertask communication.

- Events. Events notify processes or tasks that something has occurred. No data, except for a status code, is transmitted as the result of an event. In tasking programs, events facilitate synchronization between tasks or between a task and another process.

- Pipe files. Pipe files allow a process to pass data to another process, but there are limitations on the use of pipe files in tasking programs. Usually, it is preferable to use queues instead of pipes for intertask communication.

- Queues. Queues offer an efficient, flexible method for processes and tasks to communicate sizable messages.

- Shared memory and shared files. Some aspects of shared memory and shared files are available to processes and tasks.

This section describes these communication methods in the following topics.

- ‘‘Events”
- ‘‘Pipe Files”
- ‘‘Queues”
- ‘‘Shared Memory and Shared Files”

Events

An *event* is a data structure maintained by the operating system. An event transmits the fact that some processing or I/O function has occurred. For example, a task may be programmed to wait until another process or task updates a file. The updating process notifies the operating system when the event occurs, and the operating system notifies the waiting task. No data is transmitted; a counter is incremented to indicate that an event has occurred.

The event data structure includes an event ID, an event count, and a status code. The event ID identifies the event. It is unique within the process. It should not be passed outside of the process, whether in a file, queue, or shared memory. The event count is initialized to zero when the event is created, and it is incremented whenever the event occurs. The status code represents information about the results of the event.

Conceptually, an event represents an *occurrence* — any statement or section of code that has executed with an expected result.

There are two types of events: system events and user events. For *user events*, the application program creates the event, determines when the event has occurred or performs the event, and issues the notification that the event has occurred. For *system events*, the operating system creates the event, determines when the event has occurred, and issues the notification.

All event capabilities are available to individual tasks. A task can wait on a list of events and can use timeouts. Multiple tasks can wait on the same event. To keep the timeouts as accurate as possible, the system takes task switching into account when determining the timeout intervals. (The `s$wait_event` subroutine, which causes a task to wait on an event, does not return before the timeout expires; however, it might return much later.)

Like `s$wait_event`, the `s$sleep` subroutine can be called by individual tasks. `s$sleep` calls `s$wait_event` with a timeout and an empty event ID list.

User Events

All of the tasks and processes interested in a particular user event must attach to that event by calling the `s$attach_event` subroutine.

Two or more tasks or processes may share an event. Some tasks or processes wait for the event to occur, and others perform the event notification. You must provide a valid path name when attaching a task or process to a user event shared between processes. Associating a file or device to an event has no effect on the file or device. Likewise, operations on the file or device have no effect on the event or its status, although the access control list for the named object applies to the event.

You may associate the event with an existing file or device if the access control list for the existing file or device is appropriate. You may also create a directory entry for an empty file solely for the purpose of attaching an event to it.

> **Note:** It is not necessary to name an event (that is, associate a file system entity with the event) if the event is used entirely within a single process.

A task or process waits for a user event to occur by calling the `s$wait_event` subroutine. This subroutine names one event or a list of events. The calling process or task waits until **any one** of the events in the list occurs before it continues processing. You can associate a time limit with the wait through the `s$wait_event` subroutine.

When an event occurs, notification must be issued. The application program performing the event issues the notification for the event. To issue notification, a program calls the `s$notify_event` subroutine. Event notification results in the following:

- The event count is incremented by 1 for **each** notification.
- All processes and tasks waiting for the event to occur are signaled. They then resume processing. (The operating system keeps track of events, event counts, and the processes waiting for those events in a system table.)

For user events, a 4-byte status code can represent different results of the event. These results are passed on to the waiting process or task. To communicate through the status code, the notifying program supplies the desired status code when it calls `s$notify_event`. Then the waiting program reads the status code by calling `s$read_event`.

You must be careful when using the event status code in your programs. If multiple notifications occur to the same event before the waiting task or process begins to execute, the value of the status code is determined by the last notifier. For this reason, the status code is generally not useful unless the application either handles or prevents multiple notifications.

System Events

For system events, the event is created by the operating system. The `s$set_no_wait_mode` subroutine attaches tasks or processes to system events. When a process sets a port to no-wait mode using `s$set_no_wait_mode`, an event ID is returned as an output argument. The system created the event and, in effect, attached the process to it.

A task or process waits for a system event to occur by calling the `s$wait_event` subroutine. However, a task or process should not wait on a system event unless it has verified that the event has not yet occurred. That is, a wait on a system event should occur only in response to the error code `e$caller_must_wait (1277)`. Many subroutines return this message. For example, suppose that a process in no-wait mode calls the `s$msg_receive` subroutine to receive a message from a queue. If there are no messages in the queue, the error message `e$caller_must_wait` is returned. At that point, it is acceptable for the caller to call `s$wait_event`.

When an event occurs, notification must be issued. Notification for a system event is issued by the operating system when the event occurs. When the event occurs, the `s$wait_event` subroutine returns with a zero error code. The results of system event notification are the same as the results described above for user events. User programs cannot notify system events.

The operating system does not use the status code field for system events.

Figure 2-7 illustrates the concept of event notification. The figure shows only one way to use user events to communicate between processes. The event concept is a VOS capability that can be used with or without tasking. The event-related subroutines mentioned in this manual are listed below. For descriptions of these and other event-related subroutines, refer to the VOS Subroutines manuals.

- `s$attach_event`
- `s$detach_event`
- `s$notify_event`
- `s$read_event`
- `s$wait_event`

Notifying Program	
`s$create_file`	Create a file to associate with the event.
`s$attach_event`	Initiate the event for the caller. An event ID is returned.
processing event	Perform the processing that is the event or that detects the occurrence of a real world event.
`s$notify_event`	Increment the event count, and optionally assign a new status code.
`s$detach_event`	Remove the caller's association with the event.
Waiting Program	
`s$attach_event`	Initiate the first event for the caller and get an event ID.
`s$attach_event`	Initiate the second event for the caller and get an event ID.
`s$attach_event`	Initiate the third event for the caller and get an event ID.
`s$read_event`	Get the current event counts.
`s$wait_event`	List all three event IDs. Discontinue processing until one of the three events occurs.
`s$read_event`	Optionally get the new status code of the event that was notified.
`s$detach_event`	Remove the caller's association with the first event.
`s$detach_event`	Remove the caller's association with the second event.
`s$detach_event`	Remove the caller's association with the third event.

Figure 2-7. User Event Example

Macro Events

A *macro event* allows you to build a two-level array of events. This capability is useful for systems-level and networking programs that must coordinate waits on multiple events from multiple routines. The coordinating program using the macro event might have functions such as storage management, scheduling management, or performance optimization.

The multiple routines set up their events and wait on them, using the `s$attach_event` and `s$wait_event` subroutines. The rest of this section uses the term *subordinate event* to identify normal events that are also included in a macro event. The coordinating program sets up the macro event. The coordinating program's process is notified when any one of the subordinate events occurs.

> **Note:** The macro event is a useful capability in some circumstances; however, there are some restrictions on its use. The most important restriction is that only one macro event can successfully exist in a process at any one time. Since the VOS task-handling program uses macro events to implement tasking, you cannot use a macro event in a tasking process. You can, however, use the macro event in its own separate process to communicate with or coordinate tasking processes.

Figure 2-8 shows the organizational concept of a macro event. The two dimensions of the macro event are the subordinate events and "tasks."

Subordinate Events †	**Tasks**	**Macro Event** ‡
10	1	50
11		
12		
13		
14	2	
15		
16	3	
20		
21	4	
22		

† Subordinate events are normal system or user events.
‡ The macro event is established in the coordinating program.

Figure 2-8. The Macro Event Concept

The subordinate events are normal system or user events established and waited on as discussed in the ''Events" section earlier in this chapter. The program building the macro event is waiting for any one of many subordinate events to occur. When one of the subordinate events occurs, the notification is passed to the macro event through the array structure.

The "tasks" provide the second level to the array structure. In this instance, the term "task" refers to a group of related subordinate events.

A distinction must be made between a TPF task and the "task" associated with a macro event. To execute, a TPF task requires its own address space for its own stack and static data area.

The macro event “task” is an arbitrary label attached to some executable code or data; it does not require its own stack and static data area to execute. Consider, for example, a network program that uses a macro event to manage connections to multiple systems. Each system is a “task,” and each terminal is a subordinate event associated with one of the “tasks.” The program needs storage space to manage each system, but it does not need distinct stacks and static areas for each “task.”

The macro event “task” ID is an arbitrary identification for a “task.” For example, in a communications program, a “task” could group subordinate events associated with a virtual circuit. Each virtual circuit would be a different “task.” In a network program, the “task” might represent a system; each “task” would represent subordinate events associated with terminals in each system. In an application that manages multiple terminals, “tasks” might correspond to terminals.

When one of the subordinate events occurs, the macro event’s process is notified. The subordinate event that occurred and the “task” to which the subordinate event was assigned are available to the coordinating program.

The coordinating program might associate data with each “task” by entering an indexed array, using the “task” ID as the index. Another purpose for grouping subordinate events into “tasks” might be to branch to a different section of code, depending on the value of the “task” when the macro event is notified.

To set up a macro event, the coordinating program calls a sequence of macro event subroutines, as described below.

1. Call `s$attach_event` to get an event ID for the macro event.

2. Call `s$set_task_wait_info` to establish the number of “tasks” included in the macro event.

3. Call `s$task_setup_wait` to list the subordinate events associated with each “task.” The subroutine must be called once for each “task.”

Timeouts may be associated with each “task” in the `s$task_setup_wait` subroutine, and a global timeout may be assigned to the macro event in the `s$task_wait_event` subroutine.

The main loop of the coordinating program using the macro event might consist of the following steps.

1. Call `s$task_wait_event` to wait until notification reaches the macro event. The subroutine returns with the “task” number, the event ID of the subordinate event that actually occurred, and the index of that event ID in its subordinate array.

2. When the subroutine returns, use the “task” number to locate data or to point to addresses applying to the “task.” If a timeout was associated with the task, handle the timeout.

3. Use the subordinate event index to branch to program code or to call subroutines appropriate to the subordinate event that was notified.

4. Loop back to step 1.

Pipe Files

A *pipe file* is a fixed, relative, sequential, or stream file that exists only for the purpose of transmitting data between processes while a program executes. No permanent data storage occurs. The file is empty when the program is not executing. Interprocess communication is achieved when different processes are reading from and writing to the file in a first in, first out order. Normal sequential file I/O methods are used to read from, and write to, a pipe file.

A pipe file is a very fast communication technique, appropriate in applications where the first in, first out message order is important. In general, however, queues achieve the same results, with much more flexibility. The following section compares queues and pipe files.

A pipe file is defined with the `s$set_pipe_file` subroutine or the `set_pipe_file` command. For more information on pipes, refer to the description of `s$set_pipe_file` in the VOS Subroutines manuals.

Queues

Queues are an efficient way for processes and tasks to pass messages or data to each other. Tasks or processes can write messages to a queue, or they can read and receive messages out of it. Message writes, reads, and receives are performed with subroutines developed specifically for queue I/O.

The requester/server design model provides a typical example of interprocess communication using queues. The tasks in the requester process write their I/O requests to the queue, and the server process reads the requests from the queue. Some queue types support two-way communication, whereby the server process sends replies back to the requester process.

For more information on queues, see Chapter 5, ‘‘Queues.”

The major advantages of queues over pipes are:

- Messages in a pipe file cannot have priorities assigned to them, whereas messages in a queue can be prioritized.
- Once a message in a pipe file is read, it cannot be accessed again. The various queue types work differently in this respect, but some of them allow you to access the same message multiple times.
- A pipe file cannot service transactions, since a pipe file cannot be declared a transaction file. A message queue can be transaction protected. For two-way server queues, transaction protection features can be transmitted from a requester program to the server program while the requester’s message is being serviced.
- The pipe file record size is limited to 32,767 bytes. Also, when a pipe file is full (16 blocks), the writing process must wait. Queues, in general, are much more flexible.

Shared Memory and Shared Files

Tasks within the same process can share memory through the external shared variable capability described in the ‘‘Shared Variables” section later in this chapter.

Processes running on the same module can share regions of memory through the shared virtual memory capability. Refer to the description of the `s$connect_vm_region2` subroutine in the VOS Subroutines manuals for more information.

Tasks in the same process can share files. Each task can attach its own port to the file, or one task can attach a port on which all tasks within the process access the file. The ‘‘Sharing Files among Tasks” section later in this chapter describes how tasks can share a port.

Multiple processes can each attach a port to the same file, thereby “sharing” the file. Before attempting to share files among processes, read the information about locking in the `s$open_file` subroutine in the VOS Subroutines manuals.

Login vs. Slave Terminals

A *login terminal* is a terminal from which a valid user can log into the operating system, perform operating system commands, submit program modules and macros for execution, and interrupt or stop execution of commands, program modules, and macros by using the CTRL BREAK key sequence.

On a *slave terminal*, the user cannot log in, submit VOS commands or programs for execution, or stop execution of programs. The slave terminal is nonfunctional until a program module gains control over it. Control is usually gained when a task is initialized using the slave terminal’s device name. From that point, the program has control of the terminal, and there is no way for the user to break out of the executing program. The monitor task is an exception. The monitor task operator can create a subprocess and perform any command-level function.

The user may interrupt terminal input and output from a slave terminal. (For specifics, see the ‘‘Break Handling in Tasking” section later in this chapter.) However, a slave terminal user cannot stop the task or stop program execution by using the CTRL BREAK key sequence.

Terminals are configured as login or slave devices in the device configuration table, which is established by the module’s system administrator. Refer to the *VOS System Administration: Configuring a System (R287)* for more information.

Temporary Slave Terminals

A login terminal can be temporarily configured as a slave terminal with the privileged command `update_channel_info` using the `-no_login` and `-reset` options. The `-reset` option makes the change effective immediately. This command is especially useful for testing purposes during the development of an application.

The `update_channel_info` command is described in the *VOS Commands Reference Manual (R098)*.

You can obtain a login terminal's device name by issuing, from that terminal, the following VOS command and function combination:

```
display_line (terminal_name)
```

Task-Related Subroutines

The following is a list of the task-related subroutines.

`s$add_task_epilogue_handler`
: Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, the operating system executes the epilogue handlers.

`s$cleanup_task`
: Closes the terminal assigned to a specified task in the current process and detaches the port attached to the terminal.

`s$control_task`
: Stops, pauses, or continues the execution of one or more tasks in the current process. It can also enable or disable metering of CPU time and page faults on a per task basis.

`s$create_task`
: Creates a dynamic task.

`s$delete_task`
: Deletes a dynamic task.

`s$delete_task_epilogue_handler`
: Deletes a specified entry from the list of epilogue routines for the current task.

`s$enable_tasking`
: Enables or disables tasking in the calling process.

`s$get_free_task_id`
: Returns the task ID of an available uninitialized task. (The task was created, but not yet initialized.)

`s$get_max_task_id`
: Returns the largest task ID among tasks currently in use or uninitialized and available for use.

`s$get_task_id`
: Returns the ID of the calling task.

`s$get_task_info`
: Returns information about a specified task in the current process.

`s$init_task`
: Initializes a specified uninitialized task.

`s$init_task_config`
: Reads a task configuration table, initializes the set of tasks described in the task configuration file, and starts the tasks. If the calling task is defined in the task

configuration table, then the calling task restarts at the entry point specified in the task configuration table.

`s$reschedule_task`
Tells the task scheduler to dispatch another ready task.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$set_task_wait_info`
Defines a macro event and identifies the number of “tasks” in the macro event.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, and allows the calling program to supply parameters required by the starting task.

`s$task_setup_wait`
Defines the events and “tasks” associated with a macro event and specifies timeouts for those events.

`s$task_wait_event`
Causes a process to wait until the occurrence of any one of the events belonging to a named macro event. A global timeout can be specified for the macro event.

Programming Considerations

This section describes programming considerations in the following topics.

- ‘‘Shared Variables”
- ‘‘Sharing Files among Tasks”
- ‘‘Unshared Files”
- ‘‘File Positioning When Sharing Ports and Files”
- ‘‘Tasking and the Requester/Server Design Model”
- ‘‘Number of Tasks in a Process”
- ‘‘The Executable Program Module”
- ‘‘Binding Tasking Programs”
- ‘‘Adjusting Task Stack Size”
- ‘‘Entry Values”
- ‘‘Sharing a Single Terminal among Multiple Tasks”
- ‘‘Breaking Up Large Tasks”
- ‘‘The Difference between `s$sleep` and `s$reschedule_task`”
- ‘‘Break Handling in Tasking”

Shared Variables

Each task has its own static data area where the current values of unshared variables are maintained during program execution. Even if five tasks are executing the same section of code, each task maintains its own values for all variables used in the code, **unless** the variable is defined as shared.

A *shared variable* is a variable in a tasking program that is accessed as the same data item by all tasks. Its value is the same for all tasks at any point. For example, if task 1 assigns a value of 10 to a shared variable, the value of the variable is 10 for **all** tasks until any task causes the value to change. A shared variable is stored in the external static data area — an area accessible to all tasks.

A shared variable must have two attributes:

- It must be declared an **external** variable in the source program data definition.
- It must be defined as **shared**, either in the source program data definition or in the binder control file.

An *external variable* is a variable that has been defined in several source modules that are separately compiled and then bound together as a single executable program module. An external variable is accessible as the same data item throughout the program module. For information on declaring external variables, refer to the VOS Language Manuals.

The shared attribute is a feature associated with tasking that allows tasks to access the same data item. The concept of shared external variables is unique to VOS. The rest of this section describes how to assign the shared attribute.

Shared variables can be defined in two places.

- The binder control file for the program module. Using the `variables:` directive, list the variable names with an attribute of `shared`. For example:

  ```
  variables:          xyz      shared,
                      abc      shared;
  ```

  Remember that shared variables must have been declared as external in at least one of the object modules being bound.

  See the ‘‘Binding Tasking Programs” section later in this chapter for more information about binding tasking programs.

- The source module. In the variable declaration, include the attributes that indicate whether the variables are external or shared. Table 2-1 shows the correct syntax for declaring shared variables in each of the six VOS languages.

If an attribute of type shared is specified in the source module, the variable is shared, no matter what is implied by the binder control file. If the source module does not specify the shared attribute, then the shared attribute can be added to the variable in the binder control file.

Table 2-1. Language Syntax for Declaring Shared Variables

Language	**Syntax**	**Format Example**
BASIC	`shared map (`*`variable_name`*`)&` *`variable_name`*`% =` *`data_type`*	`shared map (xyz)& xyz% = 15`
C	`ext_shared` *`data_type variable_name`*`;`	`ext_shared  short xyz;`
COBOL	`01` *`variable_name data_type`* `external global.`	`01  xyz  comp-4 external global.`
FORTRAN	`common /`*`variable_name`*`(shared)/` *`variable_name`* *`data_type variable_name`*	`common /xyz(shared)/ xyz integer*2 xyz`
Pascal	*`variable_name : data_type`* `shared;`	`xyz : short_integer_type shared;`
PL/I	`declare` *`variable_name data_type`* `static external shared;`	`declare xyz bin (15) static external shared;`

Sharing Files among Tasks

Tasks within a process can share a file by referencing the same file in I/O requests. I/O requests can be language I/O statements or VOS subroutine calls. Depending on the requirements of the statement or subroutine, a file is referenced by using either the file path name or the port attached to the file.

If tasks reference a file using the file path name, in most languages the reference can be in the form of either a constant equivalent to the file path name or a variable whose value is the file path name. If a variable is used, it can be declared as a shared variable, in which case it is set equal to the file path name only once in the process, and is thereafter accessible by all the tasks. If the variable is not shared, each task must set its copy of the variable equal to the file path name.

If tasks reference a file using the port, then either all tasks can use the same port, or each task can attach its own port to the same file. The one port per process method requires that only one task attach the port, and that all tasks reference the file using the same port ID. As with the file path name, the reference to the port in most languages can be either a constant or a variable whose value is the port ID. If a variable is used, it **must** be declared as a shared variable.

Record locking implications must be considered when tasks share ports to files. Record locking occurs on a **port** basis. If many tasks share one port, then a record write lock obtained by one task would allow another task write access to the same record. In most cases, that scenario is not desirable.

If each task attaches its own port to the file, the port names, and, subsequently, the port IDs, must be unique for each task within a process. If this method is used, the port name and port ID variables must not be declared as shared. One way to obtain unique port names and port

IDs is to call the `s$attach_port` subroutine with a null port name. In this case, the operating system assigns unique port names and IDs. There are also ways to obtain unique port names and IDs through language statements. Those methods are discussed in the next section, ‘‘Unshared Files.”

Although some VOS languages allow multiple openings on the same file, it is usually better to write your application so that only one task attempts to open a shared file. The primary task or another task designated as a controller task could perform this function before it starts the other tasks.

One method for sharing ports and files uses VOS subroutine calls. One task attaches a port to the file by calling the `s$attach_port` subroutine. This subroutine returns a port ID, which must be declared as a shared variable. The task that attaches the port should also open the file by calling the `s$open` subroutine. All other tasks can access the file by using the shared port.

The following paragraphs describe how to use language statements, instead of subroutine calls, to attach only one port to the shared file. The method described assumes that one task (maybe a controller task or the primary task) attaches a port to the file to be shared and opens the file using language I/O statements. All other tasks reference the file using the same port.

BASIC

The port name is represented by a channel number. When a channel number is assigned to a file, the operating system attaches a port to the file. The channel numbers 1 through 255, inclusive, are shared among the tasks in a tasking process. The `open` statement requires a file path name and a channel number. One task in the process uses the `open` statement to attach a channel to the shared file and open the file. The channel number can be fixed in the code, or the task that attaches the channel can assign the channel number to a shared external variable. All tasks refer to the file by that channel number.

If the `map` clause is used in the `open` statement, the map variable must be defined as shared.

C

The `fopen`, `freopen`, and `open` functions open a file and attach a port. The `fopen` and `freopen` functions return a pointer that is used to reference the file, and the `open` function returns a file descriptor that is used to reference the file. The pointer variable or the file descriptor variable must be declared as shared. If all tasks reference the file using the shared variable, they are using the same port.

COBOL

When a file is opened with an `open` statement, the operating system attaches a port to the file. The name of the port is either the name you specify in an `assign` clause or a literal or identifier that you specify in a `value of port name` clause in the file description entry for the file.

The file must be defined as external.

FORTRAN

The port is represented by a unit; the port name is derived from the `unit=` specifier in the `open` statement. For example, if `unit=25` is specified, the path name is `unit.25`.

When a file is opened with an `open` statement, the operating system attaches a port to the file. Separate tasks refer to the same file when they specify the same unit number in I/O statements.

Pascal

When a file is opened with a `reset`, `rewrite`, or `open` procedure, the operating system attaches a port to the file. The port's name is the same as the file's name. All tasks refer to the same file when they use this name.

PL/I

When a file is opened with an `open` statement, the operating system attaches a port to the file. The name of the port is the same as the file constant name supplied in the `open` statement. All tasks automatically share the file when they refer to it by this name. You do not need to declare the port name as shared.

Unshared Files

Multiple tasks can execute the same section of code. Sometimes that code may refer to a file, where the intent is a separate file rather than a shared file for each task. A problem is likely to occur when a program written to run on a one-process-per-terminal basis is incorporated into a tasking application. For example, consider a program that builds a report file using language I/O calls. When the program is incorporated into a tasking application, the intent is a separate report for each task, not one report for the entire process. To achieve the desired results, a unique file path name (that is, a unique file) and a unique port name (that is, a unique port) must be used for each task.

For programs where the files already exist, you must ensure that each task can obtain the path name of the file that it should use. You might use a table or an indexed array in external static storage (shared variables) for this purpose. You could also establish a file-naming convention incorporating the task ID. Each task can call the `s$get_task_id` subroutine to obtain its own task ID, and then access a file or a port named `x.task_id`. For programs where the files do not exist, you can use various methods to generate unique file path names. These methods are described below, along with methods for generating unique port names.

The files can be created or opened and unique ports can be attached to them either by the tasks that need to use the files or before the tasks begin execution. In the latter case, a controller task or the primary task could perform the required functions before the tasks using the files are started. Either way, each task must know its unique file path name or port name in order to perform I/O to its file. The table or array mentioned above, perhaps indexed by task ID, is one way to allow each task to find its file path name or port name. The table or array must be defined as shared.

There are various ways to ensure that files and ports are unique for each task.

- Use language capabilities that guarantee unique file path names and port names. Specific descriptions for each language are described below.
- Use subroutine calls instead of language statements. The `s$attach_port` subroutine with a null value for *`port_name`* generates a port name unique in the process. Each unique prot name as a unique port ID. If the file is an output-only file and does not yet exist, a null value can also be used for *`path_name`* to create a file with a unique path name. The VOS read and write subroutines reference the file using the unique port ID returned by the `s$attach_port` subroutine.
- Include in your tasking program a controller task that performs all I/O to the files. The other tasks communicate their I/O needs to the controller task.
- Use temporary files if the file is an output-only file and it is not needed after the task stops. Most of the VOS languages can generate temporary files with automatically generated port names and path names. Language-specific information is provided below.

The following paragraphs provide specific instructions in each language for generating unique port names and file path names. Information about generating temporary files is also included.

BASIC

The port name is represented by a channel number. When a channel number is assigned to a file, the operating system attaches a port to the file. The channel numbers 1 through 255, inclusive, are shared among the tasks in a tasking process. Therefore, each task must use its own channel, with a unique channel number.

Each task's file must be opened using the `open` statement with a unique file path name and a different channel number. If a `map` clause is used in the `open` statement, the map variable must not be defined as shared.

The application must keep track of the channel numbers it is using and must generate its own unique file path names. There are no language facilities that generate unique names and numbers in BASIC.

A temporary file is created by including the `temporary` clause in the `open` statement.

C

There are no language facilities that generate unique ports attached to the same file in C. Use VOS subroutines to create unique ports.

The `tmpfile` function creates a temporary file. The file is deleted when it is explicitly closed or when the program ends.

COBOL

Two language facilities that guarantee a unique port name are listed below.

- The `select` statement, with the `assign` clause omitted. For each task that opens a file whose associated `select` statement does not include the `assign` clause, the operating system attaches a port that is unique in the process.
- The `value of portname` clause in the file description entry, defined as a null value. The operating system creates a unique port name. To specify a null value, set *`data_name`* equal to all spaces, or use the literal `''`.

Two language facilities that create unique file path names are listed below.

- The `select` statement, with the `temporary` keyword in the `assign` clause. A scratch file is created with a unique path name.
- The `value of pathname` clause in the file description entry, defined as a null value. The operating system creates a file with a unique path name. To specify a null value, set *`data_name`* equal to all spaces, or use the literal `''`.

FORTRAN

The `open` statement includes a unit number specifier (`unit=`) and a filename specifier (`file=`). There are no language facilities that generate unique unit numbers. The tasking program must ensure that each task obtains a unique unit number (perhaps by incrementing a shared variable or by associating the unit number with the task ID).

If the filename specifier is left blank, the filename defaults to `file.`*`nn`*, where *`nn`* is the unit number.

A temporary file is created if the `status=scratch` specifier is used in the `open` statement.

Pascal

There are no language facilities that automatically generate unique port names. If the file path name is blank in an `open` procedure, a file is created that has the same name as the port name.

There are no language facilities that automatically create temporary files.

PL/I

A VOS port is associated with each file control block. There are no language facilities that generate unique file path names or file control block names. In the `open` statement, each task must supply a file path name in the `title` option and a file control block name in the `file` option.

The `-delete` clause in the `title` option of the `open` statement causes the file to be deleted when it is closed. You can use this facility to create and delete temporary files.

File Positioning When Sharing Ports and Files

Database file positioning and locking occurs on a **port** basis, except when the I/O request is within a transaction. A database call that requests a “read next record” could produce erroneous results if several tasks are accessing the file through the same port. You should use keyed access and absolute positioning instead of relative positioning to avoid errors when tasks are sharing ports.

When the I/O request is within a transaction, the file is locked for the duration of the transaction. The task executing the transaction “owns” the file until the transaction is either committed or aborted.

Tasking and the Requester/Server Design Model

Tasking is usually appropriate for the requester side of a requester/server application, but not for the server side. When deciding if tasking is appropriate, consider how much file I/O is to be performed by the program. If file I/O is minimal, tasking might be appropriate, but if file I/O is frequent, individual processes are more efficient than tasks.

File I/O makes tasking inefficient because of task switching. A task switch occurs when a task is waiting, performing terminal I/O, or performing queue I/O. Task switching does **not** occur during file I/O (unless a wait for a lock is involved). While the operating system is performing file I/O for a task, all tasks wait. Contrast that situation to the one during terminal I/O, where a task switch occurs and another task gets CPU time.

Tasking is efficient for terminal I/O, but not for file I/O. Requester programs usually do not perform file I/O, whereas the major function of server programs generally is file I/O.

Number of Tasks in a Process

The number of tasks that can be included in a single process is subject to the following system limitations.

- Number of ports used. A process is limited to a total of 255 ports.
- Virtual address space available versus the task stack and static area requirements.

For the Stratus modules that contain multiple CPUs, splitting a large number of tasks between several processes permits true simultaneous processing on the different CPUs.

The Executable Program Module

A process executes only one program module at a time. Thus, an application may require many processes and therefore an equal number of executable program modules to accomplish all its functions.

The executable program module may contain various object modules that were compiled separately. For example, you may have written and compiled separate programs for tasks that perform different functions. Those programs must exist in the same executable program module if the tasks are to run in the same process. All code that is to be executed within a process must be bound together into a single executable program module. Processes that execute the same program on a module share the program code space in memory. (Processes share the same program code only when the program module resides on a disk that belongs to the module of the executing processes.)

Binding Tasking Programs

An executable program module is created with the VOS `bind` command. This command is fully documented in the *VOS Commands Reference Manual (R098)*. This section discusses a few specifics about binding tasking programs.

Tasking programs should be bound with a binder control file (use the `-control` argument in the `bind` command). This is because a binder control file permits some extra specifications for tasking programs that your application may require. The binder control file allows you to do the following:

- specify the number of static tasks to be created
- specify variables to be shared among tasks
- specify subroutines and requests that are executable by the monitor task
- specify entry points that are individually executable by any task, including the monitor task
- specify the size of the task stacks

Figure 2-9 shows a sample binder control file for creating a tasking program module. The following explanations of the directives in the binder control file refer to the example in the figure. Only the directives that apply specifically to tasking programs are explained here. For a more detailed description of the `bind` command, refer to the *VOS Commands Reference Manual (R098)*.

```
name:               auction_tasks;

entry:              auction_tasks;

number_of_tasks:    26;

variables:          xyz    shared;

modules:           auction_tasks,      /* separately compiled objects */
                    display_bidder_info,

                  user_form,          /* FMS screen objects, optional */

                    display_task_info,   /* monitor requests */
                    control_task,
                    create_task,
                    initialize_task,
                    start_task,
                    stop_task;

retain:             user_tasks_entry,    /* task entry points */
                    monitor_task_entry,

                    task_epilogue,       /* epilogue handlers */
                    program_epilogue,

                   display_bidder_info, /* programmed monitor request */

                    display_task_info,   /* monitor requests */
                    control_task,
                    create_task,
                    initialize_task,
                    start_task,
                    stop_task;

stack_size:         20480;               /* equals 5000x */

end;
```

Figure 2-9. Sample Bind Control File

`name:`
: The name the binder gives to the bound program module.

`entry:`
: The name of the main entry point of the program module.

`number_of_tasks:`
: The number of static tasks to be created. In Figure 2-9, `number_of_tasks:` is defined as 26. This means that the `bind` command creates memory space for 26 tasks. These tasks can be assigned to specific terminals in the task configuration table. Your program can dynamically create more tasks (or all but the first task) with the `s$create_task` subroutine.

`variables:`
: Special variable attributes. In Figure 2-9, the variable `xyz` is specified as a shared variable. This means that any task that uses `xyz` refers to the same variable and the same value as all other tasks in the process. Any external variable except entry variables can be shared. You can also declare shared variables in the program's data declaration statement.

`modules:`
: All separately compiled object modules that are to be explicitly bound into the program module. This includes programs, FMS form object modules (these might be optional), and all TPF requests that are to be executed by the monitor task (except `quit` and `help`). The FMS form object modules are optional if they are declared in the source module. In COBOL, objects cannot be declared in the source module, but FMS accommodates this restriction. The `quit` and `help` monitor requests are part of the `s$monitor` subroutine and are therefore always available. Also, do not include standard VOS or TPF commands. See Appendix A for a list of the TPF commands and requests. Figure 2-9 lists two user program object modules, one FMS object module, and six TPF monitor requests.

`retain:`
: All entry points that are individually and directly executable as stand-alone code. All object modules that are to be executed by the monitor task (except `quit` and `help`) must be listed. All entry points for tasks and epilogue handlers must be listed, unless they were compiled as separate programs and listed in the `modules:` directive. In Figure 2-9, one program was compiled to form an object module named `auction_tasks`. That object module is listed in the `modules:` directive. Assume that `auction_tasks` includes various entry points — entry points for user tasks, the monitor task, and two different epilogue handlers. Those entry points are listed in the `retain:` directive. All preprogrammed TPF monitor requests must be listed in both the `modules:` and the `retain:` directives.

: **Note:** In some cases, two monitor task requests are defined in the same object module. If you list both requests in this directive, the binder produces warnings referring to external names that were defined multiple times. To eliminate the warnings, delete one of the requests from the `modules:` directive. Both requests must still be listed in the `retain:` directive.

`stack_size:`

The number of bytes to be reserved for each task stack. Only one number can be listed. The space reserved is equal to the bytes specified in this directive multiplied by the number of tasks. If this directive is not used, the default stack size is 32,768 bytes for each task. Values that are multiples of 1000x (hexadecimal) are page-aligned. Page alignment is helpful during debugging.

Adjusting Task Stack Size

In an application with a large number of tasks, considerable memory is saved if a stack size smaller than the default is used. The `display_memory_usage` request in the `analyze_system` command displays how many bytes of actual memory are used for each task stack in a running program module. For an explanation of how to determine the maximum stack size for each task, see the Explanation of the `display_memory_usage` request in the *VOS System Analysis Manual (R073)*. Use this request while the program is running in its worst scenario to determine if you can afford to reduce the size of the task stacks. If a reduction seems safe, adjust the stack size in the binder control file, and then rebind the program module.

If you use the `debug` or `mp_debug` command, considerable extra stack space might be required for those commands to execute on a tasking program. Those commands usually run on the last (highest-numbered) task's stack, since that stack is extensible and the other task's stacks are not. However, you still might have to rebind the program using a larger task stack size to accommodate the debugging commands. The amount of extra space required depends on the complexity of requests that you issue while in debugging mode.

Entry Values

The `s$start_task_full` and `s$monitor_full` subroutines use variables that contain addresses as input parameters. The data type of variables containing addresses is known as `entry` or `entry_variable`, depending on the language. Variables of this type must not be declared as shared in tasking programs.

Variables defining entry values cannot be shared because, besides pointing to the subroutine code address, they also point to the address of the current (calling) task's static data area, and that value is different for each task. The same field **name** can be used for different tasks, as long as it is an internal variable (that is, not shared). An internal variable, by definition, means that a unique value is maintained for each task.

An entry value consists of three components.

- Code pointer. This pointer points to the beginning statement address.
- Static pointer. This pointer points to the executing program's static storage area. For tasking programs, it points to the static area of the current (calling) task.
- Display pointer. This pointer applies only to subroutines and procedures written in PL/I and Pascal, and then only for internal or recursive procedures that reference entry variables. In those situations, the display pointer points to the address of the stack frame of the calling procedure. At all other times, the value of the display pointer is the null pointer value (1). In C, the `OS_NULL_PTR` is used.

The `s$find_entry` subroutine converts an entry-point name (a character string) into an entry value (an address). The entry value obtained from a call to `s$find_entry` is valid only for the calling task.

Sharing a Single Terminal among Multiple Tasks

Temporary Sharing Using `s$set_process_terminal`

The *process terminal* is the terminal from which the program module was started. This is usually the terminal attached to task 1, unless task 1 was reinitialized and assigned to a different terminal.

The `s$set_process_terminal` subroutine allows tasks within a process to temporarily use the process terminal. A task is originally initialized to a specific terminal, or to no terminal. At any time, a task's original configuration can be changed to the process terminal, and tasking will be disabled. The task must eventually give up the process terminal for its original terminal (or for no terminal) so that tasking can be enabled again. The task can continue to change back and forth between the process terminal and its original configuration. Each change is achieved with a new call to the `s$set_process_terminal` subroutine.

The `terminal_switch` argument controls the switching between the process terminal and the task's original configuration. Note that `terminal_switch` is an input and an output argument to the `s$set_process_terminal` subroutine. On input, `terminal_switch` indicates which state is desired. On output, it indicates which state existed immediately before the subroutine call. The switch must be initialized to 1 (which represents the process terminal state) before the initial (first) call to `s$set_process_terminal`. Thereafter, it will always be correct for the next call because it was set as an output argument in the previous call.

The program flow for a task that needs to share the process terminal is as follows:

- Set `terminal_switch` to 1 (do this **only** before calling `s$set_process_terminal` for the first time in the task).
- Call `s$set_process_terminal`. The task's port attachments are set to the port attachments of the process terminal, and tasking is temporarily disabled.
- The calling task then sends output to the process terminal.
- To switch to the task's original terminal, call `s$set_process_terminal` with `terminal_switch` equal to the value returned from the previous call.
- To switch again to the process terminal, call `s$set_process_terminal` with `terminal_switch` equal to the value returned from the previous call.

The above scenario works well for output functions, but may be inappropriate for input functions. This is because the task that calls `s$set_process_terminal` to temporarily use the process terminal is the only active task. If the task is waiting for input, the entire process waits. (When tasking is enabled, one task waiting for input does not cause the entire process to wait—a task switch occurs, and other tasks run while the task requiring input waits.)

Two examples of applications where it might be desirable to use `s$set_process_terminal` to share a terminal are described below.

- An application that requires some type of metering or bookkeeping on itself at regular intervals could be set up so that the metering task shares a terminal. The metering task would perform its job at regular intervals and "sleep" the rest of the time. While it is working, the task takes control of a terminal; while sleeping, it gives up control to other tasks.

- A pass-through program that transfers all traffic between two ports, such as a modem and a terminal, could be implemented using two tasks that share the same terminal. One task would write input from port A to port B; the other would write input from port B to port A.

This method of sharing the process terminal works even if the process "terminal" is a file. Chapter 3, ''The Monitor Task,'' describes how to run the monitor task using a file instead of a terminal as the I/O device. If the `s$set_process_terminal` subroutine is used in that scenario, output is sent to a file, not to a terminal. Input from a file used in this way is not allowed.

Permanent Sharing

Tasks can be initialized to share terminals. This method does not disable tasking. See the explanation of the *`terminal_name`* argument in the ''Attaching Terminals to Tasks'' section earlier in this chapter.

Breaking Up Large Tasks

Tasks that perform a large amount of calculations or file I/O can execute for a long time without providing an opportunity for a task switch. Those tasks degrade the performance of the tasking application because other tasks have to wait for them to finish to obtain CPU time. The `s$reschedule_task` subroutine permits the programmer to plan a task switch at appropriate times. You should use `s$reschedule_task` frequently in long-running tasks.

The Difference between `s$sleep` and `s$reschedule_task`

Both the `s$sleep` and `s$reschedule_task` subroutines might cause a task switch. Only the `s$sleep` subroutine can cause a process switch.

With both subroutines, the calling task is suspended, and if another ready task exists in the same process, a task switch occurs. The difference between the two subroutines occurs when there are no more ready tasks. With `s$reschedule_task`, the calling task resumes processing if there are no other ready tasks. The process does not give up the CPU. With `s$sleep`, the process gives up the CPU if there are no other tasks to execute. That is, if a task switch cannot occur, then a process switch occurs.

Break Handling in Tasking

The VOS program interrupt concept works at the process level, even in tasking programs. Therefore, a tasking program generally does not generally allow task users to interrupt the executing program. However, interrupts are permitted during terminal input and terminal output.

To signal an interrupt to terminal input, the tasking user presses the CTRL BREAK keys. To signal an interrupt to terminal output, the user presses the CANCEL key.

Two mode bits in the `terminal_info` data structure control the break process. The `terminal_info` data structure is maintained by `s$control SET_INFO_OPCODE` (202). This subroutine is documented in the VOS Subroutines manuals.

The two bits that control the break process are the `break_enabled` bit and the `break_char` bit. The `break_enabled` bit controls whether break signals issued from the terminal are acted on or ignored. If `break_enabled` is false, breaks are ignored, no matter how the other bit is set. If `break_enabled` is true, breaks are acted on.

The `break_char` bit controls whether the CTRL BREAK key sequence interrupts program execution or if it only interrupts terminal input (by clearing the input buffer). If the `break_char` bit is false, CTRL BREAK interrupts program execution. If the `break_char` bit is true, CTRL BREAK does not interrupt program execution, but it clears the terminal input buffer.

Note: The `break_char` bit is set to true for all tasks when they are initialized. This prevents tasking users from interrupting program execution. If you add program code to change the `break_char` bit to false in an attempt to allow program interrupts in the tasking program, you may receive undesirable results. The operating system interrupts the active task, but the active task is **not** always the task associated with the terminal that issued the break. Program interrupts work at the process level, not at the tasking level.

Interrupting Input

Your program must meet the following requirements to allow the tasking user to clear the terminal input buffer using the CTRL BREAK keys.

- The `break_enabled` bit must be set to true.
- The `break_char` bit must be set to true. (This is done for you when the task is initialized.)
- The terminal input must be performed using the `s$seq_read` subroutine.

Interrupting Output

All of the VOS subroutines that handle writes to the default terminal port accept the CANCEL key as a signal to stop output to the terminal. Some examples of subroutines that handle terminal output are `s$seq_write`, `s$write`, and `s$write_code`. In response to the CANCEL key, these subroutines abort the output. They also either return the error code `e$abort_output (1279)` or signal the warning condition. To capture the warning condition, the warning condition must have been previously set by the `s$enable_condition` subroutine or a language statement. See the description of `s$enable_condition` in the VOS Subroutines manuals.

Your program must do the following to allow the tasking user to interrupt terminal output.

- Check for the error code `e$abort_output` or register for the warning condition.
- When an error or warning is detected, the program must reset terminal output by calling `s$control RESET_OUTPUT_OPCODE` (224). Until the `RESET_OUTPUT` request is issued, write subroutines will continue to fail and return the error or the warning.

Sample Application

Appendix B contains annotated program code for a sample application called the Automated Auction System. The annotations in the code are enclosed in asterisks and are identified with reference numbers. The reference numbers for the comments illustrating tasking features start with "T."

The requester program in the Automated Auction System uses tasking to support its multiple user and terminal requirements. The tasking program includes one monitor task and many user tasks. All the tasks are static tasks, defined in a task configuration table. All user tasks have the same entry point (`menu_entry`), and each user task is associated with a terminal. The user tasks continuously display forms on the terminals, and the users enter requests for information and bid submissions.

The requests are serviced by the server program, which is not a tasking program. The requester and server programs communicate by using a two-way server queue. All user tasks share the same queue, but each task has its own unique port attached to the queue.

A user task stops when the monitor task decides to stop the application. The monitor task sets a flag that causes each user task to stop after its current message is completely serviced. Then the monitor task stops itself. Before a task stops, it executes a task epilogue handler. When all tasks are stopped, a program epilogue handler runs. The program epilogue handler sends a message to the server program, telling it to stop.

Chapter 3: The Monitor Task

This chapter contains the following sections.

Description of the Monitor Task

The monitor task provides application management capabilities for a tasking application. Specifically, the monitor task gives a user the tools to manage other tasks in the process, to query the queues, and to set processing parameters. The user can also perform other functions from the monitor task by using standard VOS commands and application-specific commands provided by the programmer.

The monitor task displays a prompt on the task's terminal, and then waits until the user enters a response to the prompt. The monitor task performs the request and then waits again.

The monitor task user can perform five types of functions.

- TPF monitor requests. TPF monitor requests are precoded requests that the programmer can make available to the monitor task. These requests allow the user to perform functions such as displaying information about tasks in the process or about parameters of the process; creating, starting, and stopping tasks; changing task priorities; and listing messages in queues. Examples of TPF monitor requests are `start_task` and `display_task_info`.

- Programmed requests. You can program administrative functions and make them available to the monitor task. To the monitor task user, there is no difference between the programmed requests and the TPF monitor requests described above.

- TPF commands. The TPF commands work like standard VOS commands, but they relate to the special facilities of OLTP applications. The user issues them from command level or from within the monitor task. An OLTP administrator might use them for troubleshooting or monitoring the application. Examples of TPF commands are `display_lock_wait_time` and `set_transaction_file`.

- Internal VOS commands. The user can issue any VOS command-level command from a monitor task. To the monitor task user, there is little difference between the VOS commands and the TPF commands described above.

- External VOS commands. The user can issue external VOS commands by entering `..login` to create a subprocess.

The TPF monitor requests (except for `help` and `quit`) and the programmed requests that you want to make available from the monitor task must be listed in the binder control file. For details, refer to the ‘‘Binding Tasking Programs” section in Chapter 2, ‘‘Tasking.”

The VOS commands, the TPF commands, and the `quit` and `help` monitor requests are not listed in the binder control file and are always available to any monitor task user.

See Appendix A for a list of the available TPF commands and monitor requests.

Creating the Monitor Task

The following steps are necessary to create a monitor task in a tasking process.

- Reserve one of the tasks in the process for the monitor task.

- Program the monitor task to call the `s$monitor` or `s$monitor_full` subroutine. Control does not return to the calling task until the monitor task user issues the `quit` request.

- In the binder control file for the program module, under both the `modules:` and the `retain:` directives, list the TPF monitor requests that you want the monitor task to be able to execute.

- If there are other application-specific functions that you want the monitor task to execute, code them as separate entry points (or separate programs), and include the entry-point names in the `modules:` and `retain:` directives in the binder control file.

The `s$monitor` and `s$monitor_full` Subroutines

The only coding required to create the monitor task administrative environment is to call the `s$monitor` or `s$monitor_full` subroutine. These subroutines perform all of the work involved in waiting for and carrying out user requests. They return with an error code of 0 when the `quit` request is issued from the monitor task.

The `s$monitor_full` subroutine works the same as the `s$monitor` subroutine, with the following additions.

- The caller can specify an initial request.

- The subroutine can return immediately after executing the specified initial request.

- The caller can list entry points that, even though they are listed in the binder control file, are not available to the current monitor task user.

- The caller can specify a routine (such as a cleanup routine) to be called after each request line completes.
- The caller can specify a routine (such as a cleanup routine) to be called when a request line fails.

Two of the input variables in the `s$monitor_full` subroutine must have an entry data type. Since entry variables contain addresses, they must not be declared as shared. Entry variables must exist separately in each task's static data area. Refer to the ''Entry Values'' section in Chapter 2, ''Tasking,'' for a description of the entry value data type.

Performing Functions from the Monitor Task

When the `s$monitor` or `s$monitor_full` subroutine is called, a prompt is displayed on the task's terminal. (You supply the wording of the prompt in the subroutine call.)

To perform a TPF or programmed request, the monitor task user enters the command or request name in answer to the prompt. For example:

```
PROMPT: start_task 5 user_entry
```

The formats for all TPF requests are included in the VOS Transaction Processing Facility Reference manuals.

To perform a TPF or VOS command, the user responds differently. Internal VOS commands are issued either from the monitor task or from a subprocess created from the monitor task. External VOS commands must be executed from a subprocess. To issue a command directly from the monitor task, the user precedes the command with two periods. For example:

```
PROMPT: ..attach_port port1 %system#d1>file1
```

To create a subprocess from the monitor task, the user enters:

```
PROMPT: ..login
```

Then, to issue commands, the user enters the command without the preceding periods. For example:

```
ready: attach_port port1 %system#d1>file1
```

To end the subprocess, the monitor task user enters:

```
ready: logout
```

Monitor-Related Subroutines

The following is a list of the subroutines related to the monitor task.

`s$monitor`
: Reads and carries out requests for administering a tasking application.

`s$monitor_full`
: Reads and carries out requests for administering a tasking application. This subroutine permits more control than `s$monitor`.

Programming Considerations

This section describes programming considerations for the monitor task. It includes the folowing topics.

- ‘‘Running the Monitor Task without a Monitor Terminal"
- ‘‘The PAUSE Line in Terminal Output"
- ‘‘Monitor Task for Multiple Requester Processes"

Running the Monitor Task without a Monitor Terminal

For most applications, the monitor task is attached to a terminal. However, sometimes it is desirable to run the monitor task without using a terminal.

One way to run the monitor task without a terminal is to have the monitor task wait for instructions from a queue, rather than from a terminal. The monitor task is a server task that waits for messages from a two-way server queue. (The server task can be one of the tasks in a requester process.) Messages in the queue have two parts: instructions in the form of monitor requests and commands; and the name of a temporary file created by a requester task, into which the server task puts the results of executing the instruction.

A basic design for the monitor task implemented as a server task is outlined below.

1. The task receives a message. This message is used as the `initial_command_line` argument for `s$monitor_full`.

2. The task calls `s$attach_default_output`, which sends the results of the next step to the temporary file.

3. The task calls `s$monitor_full`, with the `quit` switch set to true and the `initial_command_line` set to the message received earlier. The subroutine executes the initial command, places the results of the command in the temporary file, and returns immediately after executing the command.

4. The task calls `s$detach_default_output` and sends a reply through the queue, telling the requester that the work is done.

5. The task waits for another message from the queue.

This approach also requires a program that puts instructions into the queue. The basic features of this program are as follows:

- It accepts an instruction to be sent to the server.

- It opens the server queue.

- It creates a temporary file into which the server can put the results of executing the instruction.

- It puts the instruction and the name of the temporary file into the server queue and waits for a reply.
- When the reply is received, the program looks at the temporary file to see the actual results.

Variations on the above technique are as follows:

- The server program creates the temporary file.
- A pipe file is used instead of a queue.
- The reply is wholly contained in the queue reply, eliminating the need for a temporary file.

The `PAUSE` Line in Terminal Output

This section concerns programmed requests for the monitor task that output data to the terminal using any of the VOS write subroutines.

When the monitor task user interrupts terminal output by pressing the CANCEL key, output continues until the `PAUSE` line appears. Then all other output is terminated, and the write subroutine returns. The program must call the `s$control RESET_OUTPUT_OPCODE` (224) before attempting any more terminal writes. If the reset is not performed, all future calls to write subroutines will fail.

A problem might also occur when a large volume of output is sent to the process terminal from another task by calling the `s$set_process_terminal` subroutine. The `s$set_process_terminal` subroutine disables tasking until terminal output is complete and `s$set_process_terminal` is successfully called again. Tasking could be disabled for a long time if the terminal operator fails to respond to the `PAUSE` line in terminal output.

Monitor Task for Multiple Requester Processes

A monitor task performs administrative functions only for tasks in its own process. Therefore, if your application consists of multiple tasking processes (multiple requester processes), you may want to include a monitor task for each process. You can use `s$monitor` to establish full functionality in one process, and use `s$monitor_full` in other processes to establish restricted functionality.

If one monitor is desired for the entire application, a separate process must act as the monitor. This monitor process communicates with a specified task (usually task 1) in each of the tasking processes. This communication might occur through a queue. The monitor process accepts commands issued by a monitor user at a terminal. The monitor process sends the commands through a queue to the appropriate task in the user processes. The task performs the request and perhaps sends a reply back to the monitor process through the queue.

If necessary, data can be passed from the monitor process to any task in any of the tasking processes as follows:

1. Data is sent from the monitor process to the queue.
2. Data is sent from the queue to a task in the user process.
3. The task in the user process moves the data to shared variables in its process. Any task in the process can access the shared variables.

Sample Application

Appendix B contains annotated program code for a sample application called the Automated Auction System. The annotations in the code are enclosed in asterisks and are identified with reference numbers. The reference numbers for the comments illustrating the monitor task start with “MT.”

The requester program in the Automated Auction System implements task 1 as a monitor task. The binder control file for the requester program lists two TPF commands that the monitor task can execute: `list_messages` and `display_task_info`. In this sample application, the monitor task controls when the application stops by sending a uniquely identified message to the queue. The server program does not respond to the message until all user tasks are stopped.

Chapter 4: Transaction Protection

This chapter contains the following sections.

Introduction to Transaction Protection

Transaction protection guarantees consistent, synchronized changes to a data file or set of data files. Transaction protection offers the following benefits to an online application with sensitive data.

- A defined set of I/O operations is guaranteed to be either completely performed or not performed at all.

- The application’s data is always consistent. Multiple updates can be performed or backed out as a unit, guaranteeing a synchronized set of data files and consistency within each file.

- Locking mechanisms balance maximum data access with controls on update access. Multiple users (multiple tasks or processes) can access the same data files concurrently; however, concurrent users requiring the same data object are serialized (one user must wait until the other finishes).

- A roll-forward feature restores data files by reapplying updates in a consistent manner to backup versions of the files.

- A restart feature guarantees that, if I/O operations are interrupted by any type of module or system problem, those operations are handled consistently when complete processing resumes.

This section includes the following topics.

- ‘‘Components of Transaction Protection’’
- ‘‘Transaction Definition’’
- ‘‘Application Responsibilities for Implementing Transaction Protection’’

Components of Transaction Protection

Transaction protection is implemented through the following components.

- Transaction data files. The VOS file system supports a switch that specifies whether a data file is a transaction file. Transaction protection only protects the contents of transaction files.

- Subroutines. Subroutine calls define the bounds of a transaction. Subroutines also change parameters associated with transaction locking and lock conflicts.

- The TPOverseer. The TPOverseer is a background process that performs work involved with transaction updates. The TPOverseer must be executing on all modules involved in transaction protection.

- TP-Runtime. TP-Runtime is system code embedded within the operating system that implements the transaction subroutine calls, obtains locks for the transaction, and performs other work involved with transaction updates.

- Transaction log files. The log files contain information about transaction I/O requests. These files support the consistent updating and backing out of updates guaranteed by transaction protection. The log files are also important to the restart and roll-forward capabilities.

The term *TP system* is used in this chapter as a generic term for TP-Runtime and the TPOverseer.

Transaction Definition

A *transaction* is a unit of work defined within an application program in the form of executable statements. While any kind of work can be performed within a transaction, only I/O operations on transaction files are bound and protected by the transaction protection capability of TPF. A transaction changes a transaction file or set of transaction files from one consistent state to another consistent state. This means that, within a transaction, either **all** of the operations requested on transaction files are performed, or **none** of them are performed.

A transaction is defined within a program by subroutine calls that start and end the transaction. The actual transaction consists mostly of I/O requests; however, there are no limitations on the types of statements or actions that can be included within the bounds of a transaction.

A transaction starts when the `s$start_transaction` or `s$start_priority_transaction` subroutine is called. The TP system assigns a unique transaction identifier (TID) to each transaction. The TID is subsequently used by the TP system to identify and locate the transaction and to work on behalf of the transaction. The TID is discussed in greater detail in the ‘‘Coding a Transaction’’ section later in this chapter.

A transaction ends when one of the following situations occurs.

- The transaction is *committed* by a call to the `s$commit_transaction` subroutine. When this subroutine returns a zero error code, the TP system guarantees that all update requests in the transaction will be performed as soon as possible.
- The transaction is *aborted* by a call to the `s$abort_transaction` subroutine. When this subroutine returns a zero error code, the TP system guarantees that no update requests in the transaction were performed nor will be performed.
- The transaction is aborted by the TP system due to a resource conflict with another transaction. When this occurs, the TP system guarantees that no update requests in the transaction were performed nor will be performed. The transaction must be started again.

A transaction is in the *running* state after the call to `s$start_transaction` and before the abort or commit of the transaction. Only one transaction can run in a task or a process at a time. While a transaction is running, updates that it makes to transaction files are **not** visible to other transactions. Updates are visible to other transactions only after the updating transaction is successfully committed.

Application Responsibilities for Implementing Transaction Protection

The following three steps are necessary before an executing program is assured of full transaction protection.

1. The TPOverseer process must be running on all modules that are affected by the transactions in the program. The `-protect_transactions` option of the `tp_overseer` command must be set for all of the modules.
2. The VOS data files that are to be protected must be defined as transaction files.
3. The program performing the I/O must make I/O requests in the form of a transaction.

The rest of this chapter explains each of these three steps in detail. The section on the TPOverseer explains how to start the process, what happens if it is interrupted, and how to restart it. The section on transaction files explains how to set a transaction file, what makes a transaction file different from nontransaction files, and how locking works for transaction files. Another section discusses lock contention and resolution. The section on coding a transaction explains how to start, abort, and commit a transaction. It includes recommended coding techniques for the I/O portion of your transaction, including how to handle specific error messages that might be returned. A section at the end of the chapter discusses the transaction log files maintained by the TPOverseer.

The TPOverseer

The TPOverseer performs background processing involved in committing or aborting transactions. In multimodule applications where the transaction originates on one module and some data files reside on other modules, the TPOverseer process must be running on all of the modules involved.

This section includes the folowing topics.

- ‘‘Starting the TPOverseer”
- ‘‘Restarting the TPOverseer”

Starting the TPOverseer

The TPOverseer process should start when modules are booted. This means that the `tp_overseer` command must be included in the module’s `module_start_up.cm` file. The command’s arguments are described below. For more information about the `tp_overseer` command, see the *VOS System Administration: Administering and Customizing a System (R281)*.

- `-protect_transactions`. The default for this switch is on, which means that transactions are protected. This default value should be retained for all modules involved in transaction protection for the application. Otherwise, module recovery with a consistent view of transaction states and transaction data is not guaranteed.

- `-keep_transaction_log`. The default for this switch is off, which means that logs are **deleted** when no longer needed by TP-Runtime. The default is correct for applications that do not intend to use the `tp_restore` command, described in Chapter 6, ‘‘TP Restore,” as a file recovery method. If the application intends to use the `tp_restore` command to roll forward data files, this switch must be set to on for all modules involved in transaction protection, so that logs are kept. When logs are kept, log file management becomes important to conserve disk space. Log file backup and delete schedules are discussed in Chapter 6.

- `-tlf_size`. This argument specifies the number of blocks to be allocated for the transaction log files. These files are used extensively during various phases of transaction protection. The default of 500 blocks is appropriate for most applications. If the transaction log file is too small, performance may degrade to accommodate more frequent log switches. Transaction log files that are too large complicate the restart process.

- `-twa_size`. This argument specifies the number of blocks to be allocated to the transaction work area file, which the TPOverseer process uses when committing transactions. The default of 500 blocks is optimal for most applications.

Transactions will not execute without the TPOverseer. If the TPOverseer process stops while a transaction is running (that is, before the `s$commit_transaction` subroutine returns with a zero error code), the transaction is aborted by the TP system. All subsequent attempts to start transactions on the module receive the error code `e$tp_no_overseer` (`2934`). All attempts made by transactions on other modules to reference transaction files on the module with no TPOverseer also receive the error code `e$tp_no_overseer` (`2934`).

If the TPOverseer stops after the `s$commit_transaction` subroutine returns a zero error code, the transaction is guaranteed to complete. Since some processing occurs after the `s$commit_transaction` subroutine returns, the TPOverseer may not have finished all of its processing on behalf of the committed transaction. In this case, the commit is still guaranteed. When it is restarted, the TPOverseer resumes processing for this transaction. In multimodule applications, TPOverseers on other modules continue to process the transaction.

When the TPOverseer stops accepting new transactions in order to finish processing backlogged transactions, it enters one of two error messages in the file `syserr_log.`*`date`*. (Prior to VOS Release 13.3.0, the TPOverseer did not display an error message under these circumstances.)

If your transaction log file (TLF) is too small, the TPOverseer enters the following error message.

```
TLF too small, Forcing flush -- idle needed
```

If your transaction work area (TWA) is too small, the TPOverseer enters the following error message.

```
TWA too small, Speed flush selected
```

To avoid these error messages and to obtain better throughput performance in the future, increase the size of the TWA, the TLF, or both. The default size for each is 500 blocks. If your TPF application processes a large number of transactions, you should consider setting the size of the TWA and TLF to a minimum of 5000 blocks each. If you continue to see one of these error messages, increase the size of the TWA and TLF until you no longer see the message.

Restarting the TPOverseer

If the TPOverseer stops, the system administrator should research the problem by looking in either the `syserr_log` or the `>Overseer>tp_overseer.out` file. Once the problem is fixed, restart the TPOverseer by using a command file excerpt from the `module_start_up.cm` file.

Figure 4-1 shows a sample command file to restart a TPOverseer process.

```
& This is a command macro that restarts the TPOverseer.
& The tp_overseer command specifies all default argument values.
& Note: The user of this macro must be privileged.
& Note: The user must have proper access to the >Overseer directory.
&
&set_string curdir (current_dir)
change_current_dir >Overseer
delete_file tp_overseer.out.old -brief
rename tp_overseer.out tp_overseer.out.old
create_file tp_overseer.out
set_implicit_locking tp_overseer.out
start_process 'tp_overseer' -priority 9 -privileged -process_name
TPOverseer
change_current_dir &curdir&
& ********** end
```

Figure 4-1. Sample Command File for Restarting the TPOverseer

When the TPOverseer resumes after stopping for any reason, including a system or module suspension, it resumes at the point at which it was interrupted. When it restarts, the TPOverseer uses the current log file to continue processing transactions that were committing when the interruption occurred. The TPOverseer aborts all other transactions.

If a module suspends processing for any reason, then both the TPOverseer and the application stop. A knowledgeable user should solve the problem that caused the suspension, restart the TPOverseer, and then restart the application.

Transaction Data Files

Transaction protection only protects transaction files. Nontransaction files can be accessed and updated within transactions, but those updates are not protected. This means that, if the transaction is aborted, the updates made to nontransaction files are not backed out. Also, the `s$commit_transaction` subroutine is not guaranteed to complete or synchronize updates to nontransaction files.

This section includes the following topics.

- ‘‘Declaring a Transaction File”
- ‘‘Converting a Transaction File to a Nontransaction File”
- ‘‘Opening and Closing a Transaction File”
- ‘‘Transaction-Related Locks”
- ‘‘Details on the Transaction Lock Types”

Declaring a Transaction File

Any VOS file, except for stream files, pipe files, server queues, and direct queues, can be declared a transaction file. A stream file can be converted into a sequential file and then declared a transaction file.

A file is declared a transaction file in one of the following ways:

- at the command level, by a privileged user, with the `set_transaction_file` command.

- by an application program, with the `s$set_transaction_file` subroutine. This is a privileged subroutine and must be issued from a process begun by a privileged user.

Once a file is declared a transaction file, the file cannot be accessed outside a transaction. You can open and close a transaction file outside a transaction, but all read and update requests on a transaction file must be submitted from within a transaction. The error code `e$tp_no_tid` (`2872`) is returned when an attempt is made to access a transaction file when a transaction is not being processed.

Note: Although you can read a transaction file outside of the bounds of a transaction if you open the file in `dirty_input` mode, this operation is not recommended. The dirty read operation cannot access updates made to the file within transactions until those transactions are committed. Thus, the correctness of the data read in `dirty_input` mode is not guaranteed.

Converting a Transaction File to a Nontransaction File

The `set_transaction_file` command and the `s$set_transaction_file` subroutine convert a transaction file back to an unprotected file. You may need to do this conversion to perform functions that you cannot do to a transaction file, such as truncating or renaming the file. A file cannot be changed from a transaction file to a nontransaction file, and vice versa, when the file is in use.

The `display_file_status` command shows whether a file is currently a transaction file.

In general, you should avoid switching a transaction file to a nontransaction file. If you depend on the `tp_restore` command to roll forward a corrupted data file, see the ‘‘Warnings about File Status’’ section in Chapter 6, ‘‘TP Restore,’’ before changing a transaction file to a nontransaction file. If the recommended procedures are not followed, you might ruin the ability to roll forward the file when you make the change.

In situations where you must turn transaction protection off, perform the desired operations, turn transaction protection back on, and then back up the file. Back up the file **after** returning it to the transaction-protected state. Also, remember that any updates made to a file while the file is in the nontransaction state cannot be reapplied with the `tp_restore` command.

Do not change a backup version of a transaction file to a nontransaction file. Even if you reset the backup to a transaction file, you have ruined the file for the purposes of `tp_restore`. You will not be able to use that backup version as a starting point for the roll-forward process.

Opening and Closing a Transaction File

A transaction file can be opened and closed inside or outside a transaction. You open and close transaction files using the same subroutines or language I/O statements that you use for other files.

When a transaction file is opened, its locking mode is set to implicit locking, no matter what mode was specified in the opening statement. The implicit-locking mode means that the file is locked for the duration of an I/O operation and is then unlocked.

Transaction-Related Locks

The locking methods used during transaction I/O are completely different from those used during nontransaction I/O. In addition to the implicit-locking mode imposed on a file for the duration of an I/O operation, a transaction owns other types of locks. Once obtained, locks are owned by a transaction for the duration of the transaction.

A transaction can own locks on files, records, keys, and the end-of-file (EOF). A transaction requires some type of file lock on each transaction file that it attempts to read or update. In addition to a file lock, a transaction might need either record locks or EOF locks before it can actually perform I/O. If the file is indexed, key locks, in addition to record locks, might be necessary.

The transaction-related locks are summarized in Table 4-1.

Table 4-1. Transaction-Related Locks

File Locks	**Record Locks** †	**Key Locks** ‡	**End-of-File Locks** ‡
implicit read implicit write explicit read explicit write	read write	read write	read write modify

† Record read locks are obtained implicitly; record write locks are obtained implicitly or explicitly.
‡ All key and EOF locks are obtained implicitly.

Implicitly Obtained Locks

All locks required by a transaction can be obtained *implicitly* by TP-Runtime. Based on the I/O operation requested within the transaction, TP-Runtime knows which locks are required, and it attempts to get those locks for the transaction. TP-Runtime may not be able to obtain a lock if the desired lock conflicts with a lock already owned by another transaction.

If a read operation is requested, TP-Runtime attempts to obtain appropriate read locks. If a write operation is requested, TP-Runtime attempts to obtain appropriate write locks. If a record is added to the end of a file, an EOF modify lock might be obtained.

As an example of transaction-related implicit locking, consider the following pseudocode. Assume that `file1` and `file2` are transaction files. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode.)

```
s$start_transaction
s$keyed_read rec1, key1 in indexed file file1
s$keyed_rewrite rec1, key1 in indexed file file1
s$seq_write (append) recA to sequential file file2
if all I/O successful
   s$commit_transaction
else
   s$abort_transaction
```

If TP-Runtime successfully obtains all of the locks required by this transaction, the transaction would own the following locks:

- implicit file read lock on `file1`, later converted to an implicit file write lock
- implicit record read lock on `rec1`, later converted to an implicit record write lock
- implicit key read lock on `key1` (not converted to a write lock unless the key is updated)
- implicit file write lock on `file2`
- implicit EOF modify lock on `file2`

The implicit **file** locks do not prevent other transactions from accessing the file for reads or writes. Similarly, the EOF modify lock does not prevent other transactions from also writing to the end of the same file. The pseudocode example above could be executed by multiple transactions without creating any lock conflicts if each transaction updates a different record in `file1`. If two transactions simultaneously attempt to update the same record in `file1`, a lock conflict exists.

See the ‘‘Failure to Obtain Required Locks” section later in this chapter for a discussion of the error codes returned when required locks cannot be obtained and how your application should handle those errors.

Explicitly Obtained Locks

Transactions can obtain some locks *explicitly* by issuing specific lock requests. A transaction obtains file locks explicitly by calling the `s$lock_file` subroutine or equivalent language I/O statements. Record locks are obtained explicitly by the `s$seq_lock_record`, `s$rel_lock_record`, and `s$keyed_lock_record` subroutines or equivalent language I/O statements.

Locks on keys and the EOF cannot be obtained explicitly.

In general, the file and record locks obtained implicitly by TP-Runtime are less restrictive than the file and record locks obtained explicitly. It is usually more efficient to let TP-Runtime obtain locks implicitly.

Details on the Transaction Lock Types

Too many conflicting locks can affect application performance. If you design your application to minimize and effectively handle lock conflicts, fewer transactions will abort, and response time will be faster. For this reason, the various types of transaction locks are explained in detail in the following sections.

File Locks

File read and write locks can be obtained implicitly or explicitly. The implicitly obtained read and write locks permit other transactions to access the same file for reads and writes, whereas the explicitly obtained locks are more restrictive for other transactions. For this reason, file locks should be obtained implicitly as much as possible in applications where multiple tasks submit transactions that request I/O to the same file. In other words, avoid explicitly locking a file when multiple transactions are updating the same file.

The four types of file locks are listed below.

- Implicit read lock. The transaction is guaranteed that the file is available for reads. (Individual records, keys, or the EOF are not guaranteed to be available.) This lock does not prevent other transactions from obtaining implicit file read and write locks. It only prevents other transactions from obtaining explicit file write locks.

- Implicit write lock. The transaction is guaranteed that the file is available for writes. (Individual records, keys, or the EOF are not guaranteed to be available.) This lock does not prevent other transactions from obtaining implicit file read and write locks. It only prevents other transactions from obtaining explicit file read or write locks.

- Explicit read lock. The transaction owns the file for reads. Other transactions may lock the file for reads but not for writes. This lock is assigned when a transaction calls the `s$lock_file` subroutine with the `lock_type_switch` argument set to a read lock request.

- Explicit write lock. The transaction exclusively owns the file for writes. No other transactions may access the file. This lock is assigned when a transaction calls the `s$lock_file` subroutine with the `lock_type_switch` argument set to a write lock request.

When a transaction owns an explicit read lock on a file, it does not need to obtain locks on any other objects in the file to perform reads on that file. Similarly, if a transaction owns an explicit write lock on a file, it does not need to obtain locks on any other objects in the file to perform reads or writes on that file. Transactions that own implicitly obtained file locks, however, must also obtain locks on individual records (and keys, if the file is an indexed file) or the EOF.

Table 4-2 summarizes the types of locks that different transactions can simultaneously own on the same file.

Table 4-2. File Lock Compatibilities

If One Transaction Owns:	Other Transactions Can Have:
implicit file read lock	implicit file read lock implicit file write lock explicit file read lock
implicit file write lock	implicit file read lock implicit file write lock
explicit file read lock	implicit file read lock explicit file read lock
explicit file write lock	N/A
combination explicit file read lock and implicit file write lock	implicit file read lock

As an example of file locks obtained implicitly, consider the following pseudocode. Multiple transactions can execute this pseudocode without producing any lock conflicts as long as the records being written are different for each transaction. Assume that `file1` is a transaction file. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. Also, note that in this pseudocode `s$seq_read` reads the first record in the file. For additional information, see the section ‘‘File Positioning within a Transaction,’’ later in this chapter.)

```
s$start_transaction
s$seq_read a record in file1
s$seq_rewrite a record in file1
if successful (all previous error codes = 0)
  s$commit_transaction
else
  s$abort_transaction
```

The read request causes TP-Runtime to obtain an implicit read lock on `file1`. When the write request is issued, the implicit read lock is converted into an implicit write lock. Different transactions could own implicit file read and write locks on `file1` simultaneously.

As an example of file locks obtained explicitly, consider the following pseudocode. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. Also, note that in this pseudocode `s$seq_read` reads the first record in the file. For additional information, see the section ‘‘File Positioning within a Transaction,’’ later in this chapter.)

```
s$start_transaction
s$lock_file for writes on file1
s$seq_read on a record in file1
s$seq_rewrite on a record in file1
if successful (all previous error codes = 0)
  s$commit_transaction
else
  s$abort_transaction
```

The explicit lock request results in an explicit file write lock on `file1`. Thus, the transaction that first executes this pseudocode prevents other transactions from obtaining read or write access to `file1` until the first transaction ends.

A special situation exists when a transaction owns an explicit file read lock in combination with an implicit file write lock. When that situation exists for a transaction, TP-Runtime allows other transactions to obtain implicit file read locks on the file, but no other file lock types are allowed. As an example, consider the following pseudocode. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. Also, note that in this pseudocode `s$seq_read` reads the first record in the file. For additional information, see the section ‘‘File Positioning within a Transaction,’’ later in this chapter.)

```
s$start_transaction
s$lock_file for reads on file1
s$seq_read file1
s$seq_rewrite file1
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

In the example above, the `s$lock_file` subroutine obtains an explicit file read lock for the transaction. According to Table 4-2, this allows other transactions to also own implicit and explicit read locks on `file1`. The `s$seq_read` subroutine does not obtain any locks, since the transaction already owns a read lock. However, when the write request is issued, an implicit write lock is obtained and owned in combination with the explicit read. This is the only combination of two file lock types allowed. Note that this combination of locks is listed as a unique lock type in Table 4-2.

Record Locks

Transaction-related record locks are applied to individual records or ranges of records. Record locks are obtained implicitly by TP-Runtime as a result of read or write I/O requests. A transaction obtains record write locks explicitly by calling `s$seq_lock_record`, `s$rel_lock_record`, `s$keyed_lock_record` or by using equivalent language I/O statements. The two types of record locks are explained below.

- Record read lock. This lock lets the record be read, but not updated. Multiple transactions can own read locks on the same record.

- Record write lock. This lock lets the record be updated. Only one transaction can own a write lock on a record. A transaction cannot obtain a write lock on a record if other transactions own read or write locks on the record.

A record lock is placed on every record that is “touched” during the transaction. For example, if your program calls the `s$seq_position` subroutine, all records accessed during the positioning have individual read locks established on them. Consider calling `s$lock_file` to establish an explicit read lock on the entire file, rather than causing many individual read locks.

As an example of record locks obtained implicitly, consider the following pseudocode. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. For additional information, see the section ‘‘File Positioning within a Transaction,” later in this chapter.)

```
s$start_transaction
s$keyed_read rec1, key1 in keyed_file1
s$keyed_rewrite rec1, key1 in keyed_file1
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

In the above example, TP-Runtime obtains a record read lock on `rec1`, and then attempts to convert the read lock to a write lock. The conversion is successful if no other transactions own read locks on `rec1`.

The following pseudocode shows an example of explicitly obtaining a record write lock. A record read lock is never owned by the transaction executing this pseudocode. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. For additional information, see the section ‘‘File Positioning within a Transaction,’’ later in this chapter.)

```
s$start_transaction
s$rel_lock_record rec1 in relative_file1
s$rel_read rec1 in relative_file1
s$rel_rewrite rec1 in relative_file1
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

Key Locks

For indexed files, key locks are obtained in addition to record locks. All key locks are obtained implicitly by TP-Runtime. Keys can be read or write locked, as described below.

- Implicit key read lock. A read lock is obtained for every key or range of keys used to access records. When a record is write locked and the key is not changed, the key remains read locked.
- Implicit key write lock. A write lock is obtained only when the key is updated.

Only the keys used to access the record are locked. For example, if a file is defined with two indexes, but I/O access is requested using only one index, then only one key is locked. If an index allows duplicate key values, all duplicates of the selected key are locked.

As an example of key locks, consider the following pseudocode. Assume that `file1` is an indexed transaction file with two indexes, `index1` and `index2`, defined. I/O is performed using only `index2`. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. For additional information, see the section ‘‘File Positioning within a Transaction,’’ later in this chapter.)

```
s$start_transaction
s$keyed_read file1, index name is index2, value of key is 'A'
s$keyed_rewrite file1, index name is index2, value of key is 'A'
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

The only key locked in the above example is the key value `A` of `index2`. Since the key value is not being changed in the rewrite, the key remains read locked for the entire transaction.

The following pseudocode illustrates a rewrite that changes a key value and obtains a key write lock. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode. For additional information, see the section ‘‘File Positioning within a Transaction,” later in this chapter.)

```
s$start_transaction
s$keyed_read file1, index name is index2, value of key is 'A'
change value of key to 'B'
s$keyed_rewrite file1, index name is index2, value of key is 'B'
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

The transaction above first obtains a read lock on the key value `A` of `index2`. Since the key is being changed by the rewrite request, the read lock on `A` is converted to a write lock. A write lock is also obtained on the key value `B` of `index2`. Since the record was not accessed using `index1`, no keys in `index1` were read locked. Since the rewrite did not change the value of `index1`, no keys in `index1` were write locked.

When embedded keys are used, a record rewrite might change the value of the embedded keys. When this occurs, both the original and changed values of the embedded keys are locked.

EOF Locks

When a transaction appends records to the end of a file, an EOF lock is obtained. All EOF locks are obtained implicitly for the transaction by TP-Runtime. The EOF locks apply to any type of file organization.

The three EOF lock types are mutually exclusive. An EOF can only be read locked, write locked, **or** modify locked. It can never be locked by a combination of these lock types. The three EOF locks are listed below.

- Modify lock. Multiple transactions can own modify locks on a single EOF. An EOF modify lock is obtained when the file is extended with a nonspecific write. That is, records are written to the EOF without identifying the location of the EOF. For example, the `s$seq_write` and `s$keyed_write` subroutines write a record to the EOF but do not return information about the EOF’s location. Records can be added as long as no subroutine call is made that tells the transaction where the EOF is located.

  If a transaction already has an EOF read lock and tries to obtain an EOF modify lock, the transaction is given an EOF write lock.

- Read lock. Multiple transactions can own read locks on a single EOF. An EOF read lock is obtained by any read call that crosses the EOF boundary. Crossing the EOF boundary means determining the EOF’s location. For example, the `s$get_port_status` subroutine returns the record number of the last record in the file. Another example is when the `s$seq_read` subroutine is called after the EOF. In this case, the current position specifically points to the EOF.

If a transaction tries to obtain an EOF read lock and already has a modify lock on that EOF, the transaction is given an EOF write lock.

- Write lock. Only one transaction can own a write lock on a single EOF. The EOF write lock is obtained by any write call that crosses the EOF boundary.

Note that the modify lock does not restrict other transactions from accessing the EOF, whereas the write lock can be owned by only one transaction at a time. If your application involves multiple transactions simultaneously writing to the end of the same file, you want to ensure that your I/O requests are appropriate for EOF modify locks.

As an example of an EOF modify lock, consider the following pseudocode. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode.)

```
s$start_transaction
append record to file1
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

The above code could be executed by multiple transactions simultaneously without causing any lock conflicts. An EOF modify lock would be owned by each transaction. However, the following pseudocode would result in an EOF write lock and therefore would cause a lock conflict if multiple transactions were executing this pseudocode simultaneously. (Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the pseudocode.)

```
s$start_transaction
s$get_port_status for port attached to file1 (return
last_record_number)
write record last_record_number + 1 to file1
if successful (all previous error codes = 0)
   s$commit_transaction
else
   s$abort_transaction
```

In a file with a deleted record index or a record index, the EOF is not locked when a record is being written to a physical location occupied by a deleted record. In this situation, space is reused within the file and the EOF is not involved. This is true even if a call is made to `s$rel_write` with a record number less than or equal to 0. When all of the space that was previously deleted is filled, and the file must be extended, then the EOF is involved, and an EOF write lock is obtained as described above.

Lock Contention

Transactions cannot have conflicting locks. *Lock contention* occurs when two transactions want conflicting locks on an object. The following are examples of conflicting locks.

- An explicit file write lock is desired when other transactions own implicit write locks on the file.
- A record read lock is desired when another transaction owns a write lock on the record.
- A record write lock is desired when another transaction owns a read or write lock on the record.
- A key write lock is desired when another transaction owns a write lock on the key.
- A modify lock on the EOF is desired when another transaction owns a read or write lock on the EOF.

A transaction must obtain all of the requested file, record, key, and EOF locks before it can proceed. When a lock conflict occurs, TP-Runtime performs lock arbitration to resolve the conflict. System and process parameters control lock arbitration and the length of time that the losing transaction waits to obtain a lock. These parameters have default values that can be changed by commands and subroutines. See the ‘‘Setting Lock Arbitration Parameters” section later in this chapter.

This section includes the following topics.

- ‘‘Lock Arbitration”
- ‘‘Setting Lock Arbitration Parameters”
- ‘‘Lock Wait Time Parameters”
- ‘‘Synchronizing System Times”

Lock Arbitration

Two factors control lock arbitration.

- Priority assigned to the transaction. Your program can assign a priority to a transaction when it starts the transaction. A transaction priority can be a value of -1 through 9, inclusive, with -1 indicating `always_lose`. Higher priority transactions win locks over lower priorities, unless lock arbitration parameters cause arbitration to ignore priorities and use other criteria to resolve lock conflicts.
- Start time of the transaction. A transaction’s start time is incorporated in its transaction ID (TID). During lock arbitration, the transaction that started first is considered older. The older transaction wins unless lock arbitration parameters contradict that decision. Parameters might also specify that younger transactions win or that start time should be ignored altogether.

If all lock arbitration parameters for all transactions are left to the default values, then the transaction with the highest priority wins. If transactions have the same priority, then the transaction that started first wins.

Lock Arbitration Parameters

The parameters and switches that control lock arbitration are listed below:

- `transaction_priority`. Values are -1 through 9, inclusive, with the higher numbers meaning higher priority. A value of -1 means `always_lose`.
- `ignore_priority`. Values are true or false.
- `ignore_time`. Values are true or false.
- `younger_wins`. Values are true or false.
- `allow_deadlocks`. Values are true or false. When all transactions involved in the conflict have true values, the transaction that currently owns the lock keeps the lock and continues processing. All other transactions wait. A true **deadlock** situation exists when two or more transactions hold locks in such a way that some transactions must let go for processing to proceed. For example, transaction A holds a write lock on record R and needs a write lock on record S to complete the transaction. At the same time, transaction B holds a write lock on record S and needs a write lock on record R. One of the transactions must release its lock before either can proceed. However, no transaction waits longer than the lock wait time.
- `time_value`. Values are specified in seconds. This is the amount of time that the start times for two transactions can differ and still be considered the same start time. This parameter is especially useful for multimodule applications, because module times may differ slightly. (See the ‘‘Synchronizing System Times” section later in this chapter.)

Lock Arbitration Algorithm

When lock contention exists, TP-Runtime uses the following algorithm to resolve the conflict.

1. If exactly one transaction has a priority value of `always_lose`, that one loses. Otherwise, go to step 2.
2. If the `ignore_priority` parameter is true for both transactions, go to step 4. Otherwise, go to step 3.
3. If the priorities assigned to both transactions are equal, go to step 4. Otherwise, the higher priority wins.
4. If the `ignore_time` parameter is true for both transactions, go to step 7. Otherwise, go to step 5.
5. If start times are “equal” (that is, they differ by less than the `time_value` specified), go to step 7. Otherwise, go to step 6.
6. If the `younger_wins` parameter is true for both transactions, the transaction that started later wins. Otherwise, the transaction that started first wins.
7. If the `allow_deadlocks` parameter is true for both transactions, the transaction that currently owns the lock keeps the lock. The transaction that wants the lock waits, but

not longer than the lock wait time. If the `allow_deadlocks` parameter is not true for both transactions, go to step 8.

8. If the `younger_wins` parameter is true for both transactions, the larger TID (the later start time) wins. Otherwise, the smaller TID (the earlier start time) wins.

Results of Lock Arbitration

In any lock conflict, one transaction already owns the lock and another transaction is requesting the lock. If the losing transaction previously owned the lock, that transaction is aborted so the winning transaction can have the lock. If the losing transaction did not previously own the lock, it waits for the lock and is not aborted.

Figure 4-2 shows the results of lock arbitration. In the figure, Transaction A owns the lock. If Transaction A wins the conflict, then Transaction A continues processing without interruption, and Transaction B waits. However, if Transaction A loses the conflict, then it is aborted, and Transaction B continues processing.

Transaction
A
owns lock.

Transaction
B
wants lock.

If B wins, A aborts.

If A wins, B waits.
(If `lock_wait_time` is exceeded,
B receives `e$`*`xxx`*`_in_use` error code.)

Figure 4-2. Results of Lock Arbitration

Lock wait time parameters control the length of time the transaction waits for a lock that it needs but cannot currently have. If the time is exceeded, the I/O call returns one of the following error codes, whichever is appropriate. (The transaction is not aborted.)

```
e$file_in_use (1084)
e$record_in_use (2408)
e$key_in_use (2918)
```

For a discussion on how to handle these errors, see the ‘‘Failure to Obtain Required Locks” section later in this chapter.

Setting Lock Arbitration Parameters

On every module, a default value exists for each lock arbitration parameter. The default parameter values assigned when a module is booted are listed in Table 4-3. A system administrator can change the module defaults using the `set_tp_default_parameters` command. A program can change the module's defaults by calling the `s$set_tp_default_parameters` subroutine. Module defaults can be changed only by a privileged process.

The `display_tp_default_parameters` command lists module defaults online and the `s$get_tp_default_parameters` subroutine returns module defaults to the calling program.

Table 4-3. Lock Arbitration Parameter Defaults

Parameter	**Initial Default Value**
`transaction_priority`	0
`ignore_priority`	false
`ignore_time`	false
`younger_wins`	false
`allow_deadlocks`	false
`time_value`	10 seconds

As processes are created, the module defaults are copied into local defaults kept in each process. The parameters can be changed for an individual process by using the `set_tp_parameters` command or the `s$set_tp_parameters` subroutine. Only the parameters of the calling process are updated. The `display_tp_parameters` command and the `display_tp_parameters` monitor request list the current parameter values for the process. The `s$get_tp_parameters` subroutine returns the parameter values of the calling process.

If the module defaults are changed, only the processes still using the module defaults are given the new values. The processes that have called the `set_tp_parameters` command or the `s$set_tp_parameters` subroutine are not affected by the new module defaults.

The lock arbitration parameters can also be changed for an individual program with the `s$set_tp_parameters` subroutine or the `set_tp_parameters` monitor task request. The `display_tp_parameters` monitor request lists the current parameter values for the program. (For both the `s$set_tp_parameters` subroutine and the `display_tp_parameters` monitor task request, the `program_or_process` argument indicates whether you are affecting the parameters for the program or the process.)

In this case, the term *program* refers to an executable (bound) program module, which may include many separately compiled programs. When the program module stops executing, any parameter values specified for the program are lost. Only one program module executes in a process at a time.

For batch processes, the process is deleted when the program module stops, so the distinction between process and program is not important. However, for interactive processes, a user may start many different program modules during the day from the same process. Whether it is more appropriate to set parameters for each process or for a program module depends on your application.

Table 4-4 summarizes the various ways that the default lock arbitration parameters can be changed.

Table 4-4. Changing the Default Lock Arbitration Parameters

	Module-Wide Default	**Process Default**	**Program Default**
Subroutine	`s$set_tp_default_parameters` `s$get_tp_default_parameters`	`s$set_tp_parameters` `s$get_tp_parameters`	`s$set_tp_parameters` `s$get_tp_parameters`
Command	`set_tp_default_parameters` `display_tp_default_parameters`	`set_tp_parameters` `display_tp_parameters`	
Monitor Request (Tasking)			`set_tp_parameters` `display_tp_parameters`

Lock Wait Time Parameters

The lock wait time parameters control how long a process or a task waits to obtain locks required for I/O. The lock wait time affects all waits for I/O locks, whether or not transaction processing is involved.

Every module has a `default_lock_wait_time` parameter, which is set with the `set_lock_wait_time` command. The command should appear in the module's start-up macro, so the module default value is set when the module is booted.

A system administrator can change the module's default value with the `set_lock_wait_time` command. A program can change the module's default value by calling the `s$set_default_lock_wait_time` subroutine. The module's default can be changed only by a privileged process.

As processes are created, the module's default is copied into a local default kept in each process. This local value can be changed for an individual process with the `set_process_lock_wait_time` command or the `s$set_lock_wait_time` subroutine. Only the `lock_wait_time` for the current process is updated. The `display_process_lock_wait_time` command shows the current `lock_wait_time` for the process. Similarly, the `s$get_lock_wait_time` subroutine returns the current `lock_wait_time` for the calling process.

If the module's default is changed, only the processes on that module that have not set their own process lock wait times are given the new module default value. The `display_lock_wait_time` command and the `s$get_default_lock_wait_time` subroutine obtain the current default wait time for a module.

The `lock_wait_time` can also be changed for an individual program. See the ‘‘Setting Lock Arbitration Parameters” section earlier in this chapter for a description of the difference between a program and a process in this context. The `lock_wait_time` for an individual program is changed with the `s$set_lock_wait_time` subroutine. The `s$get_lock_wait_time` subroutine returns the current `lock_wait_time` for the program. (The `program_or_process` input argument indicates whether these subroutines apply to the process or the program.)

Table 4-5 summarizes the various ways that the default lock wait time parameters can be changed.

Table 4-5. Changing the Default Lock Wait Time Parameters

	Module-Wide Default	**Process Default**	**Program Default**
Subroutine	`s$set_default_lock_wait_time` `s$get_default_lock_wait_time`	`s$set_lock_wait_time` `s$get_lock_wait_time`	`s$set_lock_wait_time` `s$get_lock_wait_time`
Command	`set_lock_wait_time` `display_lock_wait_time`	`set_process_lock_wait_time` `display_process_lock_wait_time`	

Synchronizing System Times

In a multimodule system, every module has its own clock. The times on different modules are not guaranteed to be identical even if they are synchronized to the same time initially. Various factors can cause the times among modules to vary by approximately 4 seconds per day.

In lock arbitration, transaction start times are used to determine which transaction is older. In applications where transactions originating on different modules (that is, various requester modules) are involved in lock conflicts, the time variations among the clocks of different modules could affect the outcome of the lock arbitration.

One of the lock arbitration parameters is `time_value`. This parameter allows you to specify, in seconds, the amount of time that two or more transactions’ ages can differ and still be considered the same age.

Gross time differences in the module clocks can be prevented if a privileged user periodically executes the `set_jiffy_times` command. A privileged user can check the time differences among the modules in a system with the `check_jiffy_times` command. These two commands are described in the *VOS System Administration: Administering and Customizing a System (R281)*.

Coding a Transaction

To execute properly, a transaction should be coded as outlined below.

1. Start the transaction.

2. Perform the transaction in the form of I/O requests. For each I/O request, check the error code to see if the required locks were obtained and to see if TP-Runtime aborted the transaction. (A zero error code indicates that the request was successful.)

3. If locks were not obtained, retry the I/O request a limited number of times. Abort the transaction when the limit is reached.

4. If TP-Runtime aborted the transaction, restart the transaction a limited number of times. Perform an alternate action if the transaction cannot be committed after the maximum number of restarts.

5. End the transaction by either committing or aborting it.

This section includes the following topics.

- ‘‘Starting a Transaction”
- ‘‘Performing I/O Calls within the Transaction”
- ‘‘Failure to Obtain Required Locks”
- ‘‘Transaction Aborts by the TP System”
- ‘‘Details of `s$commit_transaction`”
- ‘‘Details of `s$abort_transaction`”

Starting a Transaction

Starting a transaction means telling the file system that subsequent I/O requests should be considered as a unit, that either all I/O requests should succeed or none should succeed, and that the updates should not be visible to other transactions until this transaction is successfully committed.

Two subroutines can be used to start a transaction.

- `s$start_transaction`. This subroutine starts the transaction and gives it a default priority value.

- `s$start_priority_transaction`. This subroutine starts the transaction and overrides the default priority value.

The priority values mentioned above refer to the priority the transaction has in winning lock conflicts. In a lock conflict, a transaction with a high priority generally wins over lower priority transactions. Recall that priorities are expressed as values of -1 through 9, inclusive. The default priority value is 0, but that value can be changed by a system administrator. See the ‘‘Setting Lock Arbitration Parameters” section earlier in this chapter for more details.

In a nontasking process, only one transaction per process can exist at a time. The process cannot start a new transaction until the previous transaction is either committed or aborted.

In a tasking process, only one transaction per task can exist at a time.

Transaction Identifier

When a transaction starts, TP-Runtime assigns a unique identifier called the *transaction ID* (TID) to it. The TID is a value that contains the time the transaction started and the module on which it started. During lock arbitration, TP-Runtime determines which transaction started first by looking at the start time in the TIDs of the transactions.

The TID identifies all I/O calls within a transaction as a single unit. In multimodule applications, the TID also allows TPOverseers on different modules to communicate about a transaction. The TID is for internal use only; it is not available to the application program.

This TID is a different entity and is used for different purposes, from the TID used in network operations.

File Positioning within a Transaction

The current file position is not maintained between transactions. The first file operation in a transaction resets the current position to the beginning of the file. You must reset the file pointer if you want to continue from a previous position in a prior transaction. In other words, sequential reads in separate transactions always start at the beginning of the file.

Performing I/O Calls within the Transaction

A transaction consists of one or more I/O calls to one or more VOS files.

Through its locking mechanism, transaction protection guarantees consistent data during a transaction. For example, a transaction that reads a record and then later decides to change that record is guaranteed that the record was not changed since it was last read. Also, no other transaction can simultaneously attempt to change that record.

When a transaction performs an update, the update is immediately visible only to the transaction that made the update. Other transactions will not see the update until the transaction making the update commits. For example, if a transaction changes a record, a subsequent read of the changed record in the same transaction obtains the changed information. However, a read of the same record from a different transaction results in a lock conflict. If the second transaction wins the conflict, it obtains the unchanged version of the record. (The original transaction aborts when it loses the lock conflict.)

The transaction-related locks are explained in detail in the ‘‘Lock Contention” section earlier in this chapter.

Failure to Obtain Required Locks

Your program must handle the possibility that required locks cannot be obtained for an I/O call. When a transaction that does not own a lock loses a lock conflict, it waits the current `lock_wait_time` to obtain the lock. If the transaction times out while waiting for the lock, one of the following error codes is returned:

```
e$file_in_use (1084)
e$record_in_use (2408)
e$key_in_use (2918)
```

The specific error code returned depends on whether the transaction timed out while it was waiting for a file lock, a record lock, or a key lock. In most applications, you can respond to all three error codes in the same way; the type of lock that caused the timeout is usually not important.

Your program should check for the above three error codes after every I/O call in a transaction. You can handle these errors in two ways.

- Retry the I/O request. The recommended procedure for retrying I/O requests is to execute, a limited number of times, a loop that waits a small amount of time and then reissues the I/O request. The `s$sleep` or `s$reschedule_task` subroutine can be used to elapse a small amount of time. Use a counter to keep track of the number of times a specific I/O request is retried. A maximum of five retries is recommended. Abort the transaction and perform an alternate action if the maximum is reached.

- Abort the transaction and start over. Use a counter to keep track of the number of times the transaction is aborted and restarted. A maximum of five restarts is recommended. Perform an alternate action if the maximum is reached. Some alternate actions might be to send a message to a requester to try again later, to send a message to a monitor task requesting instructions, or to stop the application.

Handling errors by retrying the I/O request usually offers the transaction a better chance of obtaining the required lock, since lock arbitration usually depends on the transaction's TID. When you retry an I/O request, the original transaction is still running with the same TID. Depending on how you set the lock arbitration parameters, an older transaction might have an advantage over a younger transaction in winning lock conflicts.

Transaction Aborts by the TP System

A transaction might be aborted by either TP-Runtime or the TPOverseer. Your program must expect this possibility and be prepared to handle it.

TP-Runtime might abort a transaction as a result of any I/O request. The following situations could cause an abort.

- A transaction that owns a lock loses the lock to a higher priority transaction.
- There is not enough disk space available to perform the commit of the transaction.
- The transaction exceeds the limit for the number of disk blocks that can be reserved for each file involved in a transaction. The limit to 10,000 disk blocks for each transaction file. The TP system reserves disk blocks so that it can guarantee that the actual physical updates to the files will be successfully completed.

In multimodule transactions, the TPOverseer might abort a transaction for the following reasons.

- One of the modules involved in the transaction was taken offline after the transaction started.

- The TPOverseer process is not running on one of the modules involved.

- A timeout related to the coordination of several modules occurs (for example, one TPOverseer takes a long time to respond to the coordinating TPOverseer concerning the transaction).

When the TP system aborts a transaction, the error code `e$tp_aborted` (`2931`) is returned. This error can be returned in response to any I/O subroutine call issued in the transaction, including `s$commit_transaction`. For multimodule applications, the TP system notifies all other modules involved in the transaction, and the transaction is aborted on all modules. All subsequent I/O requests against a transaction file in the task or process receive the error code `e$tp_aborted` (`2931`). Requests to commit or abort the transaction receive the error code `e$tp_no_tid` (`2872`), which means that no transaction is running in the task or process (that is, the transaction was aborted).

If language statements, rather than subroutine calls, are used to accomplish I/O, the program may not be notified of an abort that the TP system performed until the program calls `s$commit_transaction`. In this case, the error returned could be either `e$tp_aborted` (`2931`) or `e$tp_no_tid` (`2872`).

You may want to test for the `e$tp_aborted` (`2931`) error code after every I/O call and after a call to `s$commit_transaction`. An alternative is to check for both `e$tp_no_tid` (`2872`) and `e$tp_aborted` (`2931`) after `s$commit_transaction`. A specific action, such as sending a message to a system administrator, may be desirable when the TP system aborts a transaction.

Details of `s$commit_transaction`

A call to the `s$commit_transaction` subroutine indicates to the TPOverseer that the transaction has ended and updates may be made visible to other transactions. At this point, all information required to perform the updates is recorded in log files, enabling the successful continuation of this transaction if a restart becomes necessary.

Note: The `s$commit_transaction` subroutine can **only** be called by the process or task that started the transaction.

A commit is a two-phase procedure. During *phase 1*, the TPOverseer decides whether to apply all the updates requested within the transaction or to abort the transaction. In *phase 2*, the TPOverseer actually performs the updates. If the transaction was aborted in phase 1, then phase 2 is never reached.

The call to `s$commit_transaction` transfers control of the transaction to the TPOverseer. The calling program waits for the TPOverseer to complete phase 1. When phase 1 is complete, the `s$commit_transaction` subroutine returns, giving control to the calling program. The error code returned by the subroutine tells the calling program whether the transaction was aborted or guaranteed to commit. (An error code of zero means that the transaction is guaranteed to commit.)

If the transaction was guaranteed to commit, the TPOverseer continues working on phase 2. Meanwhile, the calling program can perform processing unrelated to the transaction, including starting other transactions. The calling program does not need to be notified when phase 2 is completed, since completion was already guaranteed.

Note: When the `s$commit_transaction` subroutine returns a zero error code, the updates are guaranteed to be performed; however, it may take some time for the updates to be completed. During that time, the TPOverseer owns write locks on the records and keys being updated. Therefore, you cannot expect your program to return from the `s$commit_transaction` subroutine and immediately begin another transaction that requires I/O to the same records and keys. In this case, the program would usually receive the error code `e$record_in_use` (`2408`) or `e$key_in_use` (`2918`). The program can, however, perform I/O on **different** records and keys in the files being updated.

More details about the two-phase commit are provided below.

- In phase 1, the TPOverseer releases all read locks held for the transaction and then decides whether to perform all of the updates in the transaction. In multimodule applications, the modules involved will either all commit or all abort the transaction. The TPOverseer on the module where `s$commit_transaction` was called coordinates the commit procedure among the other modules. The coordinating TPOverseer surveys the TPOverseers on all other modules involved to determine if all modules can guarantee to commit the transaction.

  When the TPOverseer decides whether it should abort or guarantee to commit the transaction, it notifies TP-Runtime.

  TP-Runtime determines which error code to return to the calling program. If the transaction was aborted, the `s$commit_transaction` subroutine returns the error code `e$tp_aborted` (`2931`). If the transaction was guaranteed to commit, the `s$commit_transaction` subroutine returns an error code of zero.

  If the TPOverseer decides that the transaction cannot be committed, it releases all locks still held for the transaction.

- In phase 2, the TPOverseer applies the updates to the data files, using the information in the transaction log files. As it completes each step in the phase 2 commit process, the TPOverseer records its actions in the transaction log file. If it is forced to stop processing and restart later, the TPOverseer has a record of how far it progressed in the commit and can continue where it left off.

  In multimodule applications, all of the TPOverseers on all modules involved in the transaction will complete the updates. If one of the TPOverseer processes stops or one of the modules suspends processing, all processing of that transaction is temporarily suspended. Processing of the transaction continues when all required TPOverseers resume running.

Details of `s$abort_transaction`

The `s$abort_transaction` subroutine tells the TPOverseer that the transaction should be canceled, and that all transaction files should exist as if the transaction was never started. A program might abort a transaction because a desired record is not in the file, or because information in the last record read invalidates the transaction (for example, in a banking application, when an account is overdrawn). When a program aborts a transaction by calling the `s$abort_transaction` subroutine, no further action concerning the transaction is necessary. All read and write locks associated with the transaction are released when the transaction is aborted.

In multimodule applications, the TPOverseer on the module that issued the call to `s$abort_transaction` coordinates the abort among all modules involved in the transaction. The abort is recorded in each module’s log file.

If `s$abort_transaction` returns the error code `e$tp_no_tid` (`2872`), a TID could not be found for the calling process or task. One reason for this error is that the transaction was previously aborted by the TP system.

The `s$abort_transaction` subroutine can be called on a different module or by a different process or task from where the transaction was started. Applications using two-way server queues can take advantage of this feature. For example, a transaction might be started by a requester process and be aborted by a server process. If this procedure is used, the aborting program should communicate the abort to the starting program (through the queue, for example). Otherwise, the starting program will receive the error code `e$tp_no_tid` (`2872`) when it attempts to commit the transaction or issue more I/O requests.

When a transaction is aborted, whether it is aborted by the user program or by the TP system, no further transaction activity can be accomplished until a new transaction is started by a call to `s$start_transaction`.

The Transaction Log Files

The transaction log files contain information about each step involved in a transaction. This information includes all of the locks obtained for the transaction, the data in the records owned by the transaction, and the changes made to those records. The log files also include records indicating the beginning of the phase 1 commit, the beginning of the phase 2 commit, and the completion of the phase 2 commit. If the transaction was aborted, the abort is recorded. The log files also contain information concerning other modules involved in the same transaction.

This section includes the following topics.

- ‘‘Transaction Log-File Names”
- ‘‘Log-File Maintenance”
- ‘‘Log Switches”
- ‘‘When Space Is Not Available”

Transaction Log-File Names

The log files reside in the following master disk directory:

```
>system>transaction_logs
```

The TPOverseer assigns file names to log files as follows:

`tlf.`*`s-m.nnnnnnnnnn.date`*

where:

`s-m` is the system number, followed by the module number.

`nnnnnnnnnn` is a sequential number, starting with 0000000001. The numbering assignment starts over if there are no log files existing on the module and no links in `>system>transaction_logs`.

`date` is the date the TPOverseer created the file.

The TPOverseer maintains a link in the `>system>transaction_logs` directory that points to the current transaction log. The link name is `transaction_log`. During log switching, a second link named `new_transaction_log` is created. When log switching is complete, this second link is unlinked, and the `transaction_log` link correctly points to the newly created log.

Log-File Maintenance

The TPOverseer creates and maintains the log files. When a log file fills up, the TPOverseer creates a new one. Users are not involved in log-file maintenance unless the disk where the files reside runs out of space.

The `tp_overseer` command, which starts the TPOverseer process, includes two arguments that affect the log files. One of these arguments is the `-keep_transaction_log` option. If this option is set to **off** (the default), the TPOverseer deletes old log files after it creates new ones. You do **not** want this to happen if you intend to use the TP roll-forward operation. To preserve old log files, specify the `-keep_transaction_log` option in the `tp_overseer` command.

The `-tlf_size` argument of the `tp_overseer` command specifies the size of the log files. In most cases, the default of 500 blocks is appropriate. If the log file's size is too small, the TPOverseer's performance might degrade to accommodate more frequent log switches.

There is one TPOverseer per module, writing to one set of log files. If several applications using transaction protection are running on the same module, transactions from all of the applications are written to the same set of log files.

Log Switches

When the TPOverseer detects that its log file is about to run out of space, it attempts to create a new log file. This action is called a *log switch*.

Normally, only one log file per module is open at a time. When that log file is full, the TPOverseer creates and opens a new log file and closes the old one. However, the TPOverseer does not close a log file until all of the transactions that started logging to that log file have ended. This means that two transaction logs may be open on a module. No more than two open logs are allowed.

If two logs are open and full, and neither can be closed, the TPOverseer continuously expands the newest log file. The TPOverseer predicts when a log expansion is required, and it attempts to preallocate additional space for the current log file. If the space is available, then the space is allocated, the log file is expanded, and processing continues uninterrupted. All log-file expansions are recorded in `syserr_log`.

A transaction should not include code that can cause the transaction to remain unfinished for a long time. For example, from within a transaction, your program should not call `s$sleep` with a long wait time, nor should your program wait for user input from a terminal. Those code segments could prevent log switches, causing the second log file to expand to an undesirable size. If these actions are needed, perform them before calling `s$start_transaction`.

When Space Is Not Available

If a log switch or a log expansion fails because the disk has run out of space, the TPOverseer aborts the current transactions on that module. The TPOverseer then suspends itself, which means that the TPOverseer is still running, but it does not process any new transactions. All `s$start_transaction` calls receive the error code `e$no_space` (`1007`). The suspension of operations is recorded in the `syserr_log`.

Disk space must be freed before processing can continue. The TPOverseer detects when space is available and resumes processing without user intervention. The TPOverseer can sometimes create space for itself to continue processing. This situation usually occurs during a log switch, when two log files exist. When the TPOverseer aborts transactions because of the out-of-space error, the old log can then be closed and deleted (if you are allowing old logs to be deleted). Enough space is thereby created to allow the TPOverseer to continue. However, the situation should not be ignored, since disk space is running short. Processing will probably be temporarily interrupted and transactions unnecessarily aborted during the next log switch.

Performance Considerations

Transaction-protected I/O is slower than unprotected I/O. This performance consideration must be evaluated against the consistency guarantees that transaction protection offers your application's data files. If you use transaction protection, your data-file record contents and program I/O requests should be designed to maximize program efficiency.

To maximize program efficiency, keep transactions small. There is no limitation on the number of I/O calls that can be included in a transaction. However, transaction protection works better in terms of response time and number of transactions aborted if transactions are kept small. A series of several small transactions works better than one large transaction because there are fewer lock conflicts. Also, less time is required for the TPOverseer to commit the transaction, so the wait time for the calling program is less, and the actual updates are visible to other transactions much sooner.

A careful design of your data files can also improve transaction efficiency. Design the files so that static data (data that is rarely changed) is in a separate file from the data that is changed frequently. For example, a financial application can separate static data such as names, addresses, and credit ratings from the constantly changing account balances. Also, try to minimize the number of files that a transaction updates. Transactions are more efficient if they update fewer files.

Try to minimize duplicate keys in indexed files. When a transaction owns a key lock, all duplicates of the key are also locked. As the number of records containing the duplicate key increases, the potential for lock contention also increases.

Performance Improvements

The `allow_deadlocks` lock-arbitration parameter can be used to improve performance on heavily loaded systems experiencing poor throughput. In cases where such performance problems are related to excessive pre-emptive transaction aborts, using `allow_deadlocks` eliminates most aborts and allows complete control at the application level.

Although this approach can be used successfully, it can sometimes cause undesirable side effects because the application is completely responsible for preventing deadlocks (since `allow_deadlocks` inhibits pre-emptive aborts). A *deadlock* typically occurs when one transaction is holding a lock that another transaction is waiting for, although other types of deadlocks can also occur. A list of possible side effects follows.

- If a deadlock occurs, the application often does not detect the situation until after significant delays occur.
- A deadlock situation can continually recur, causing extreme delays for all transactions requiring a lock held by a deadlocked transaction.
- An application developer has no effective tools to help isolate the cause of the deadlock.

The following sections describe how improvements to TPF address these and other performance issues.

- ‘‘Criteria Used to Abort Transactions”
- ‘‘Minimum Wait Period Change”
- ‘‘Lock-Conflict Logging Changes”
- ‘‘Transaction Metering Improvements”
- ‘‘Faster Key Searching”
- ‘‘Deadlock Detection”
- ‘‘Transaction Priority Enhancements”
- ‘‘I/O Operations and Aborts”
- ‘‘TP Lock Manager”
- ‘‘Processing Delays Eliminated during Transaction Log Switches”

Criteria Used to Abort Transactions

In order to ensure that it can make forward progress on all transactions that may possibly require identical locks, TPF uses certain criteria to determine when it should abort a transaction to allow another transaction to proceed. The criteria involve start time, priority, and transaction ID, which is generally associated with the transaction’s age. You can set the criteria using the `set_tp_default_parameters` or `set_tp_parameters` command. For information about these commands, see the VOS Transaction Processing Facility Reference Manuals.

In general, before TPF aborts a transaction, it waits a specified amount of time. This waiting period may allow the transaction to complete on its own and therefore voluntarily relinquish its locks. You can set this amount of time by using the `abort_wait_time` parameter of the `set_tp_param` request of the `analyze_system` command. For information about the `abort_wait_time` parameter, see the `set_tp_param` request in the So*ftware Release Bulletin: VOS Release 14.0.0 (R914).*

Minimum Wait Period Change

In VOS releases prior to VOS Release 14.0.0, lock arbitration could cause excessive transaction aborts. Consider the case in which two transactions that meet the same criteria contend for the same lock. The transaction desiring the lock waits for the transaction holding the lock to complete. The transaction waits for a minimum of five seconds and until the implicit or explicit wait time specified by the I/O call that resulted in the need to acquire the lock. You can modify the five-second minimum by specifying a value for the `abort_wait_time` parameter of the `set_tp_param` request of the `analyze_system` command. For information about this parameter, see the `set_tp_param` request in the *Software Release Bulletin: VOS Release 14.0.0 (R914)*.

If the wait period passes and the lock is not released, **and** the transaction desiring the lock started before (that is, meets higher criteria than) the transaction holding the lock, the transaction of higher criteria pre-emptively aborts the transaction holding the lock. This behavior prevents deadlocks. Note that this behavior does **not** occur if you set the `allow_deadlocks` parameter of the `set_tp_default_parameters` or `set_tp_parameters` command. For information aboutthe `allow_deadlocks` parameter, see the *VOS Transaction Processing Facility Guide (R215)*.

The actual waiting is done by user-level I/O processing. Whenever a lock for which any transaction is waiting is released, all processes waiting for a lock in that file awaken and retry the I/O operation, which, in turn, tries again to acquire the lock. Often, the lock will then be available and the transaction can proceed. However, in situations in which a transaction that does not meet higher criteria is still holding the lock, the transaction of higher criteria aborts the other transaction.

This algorithm works well when relatively few transactions are contending for records in the same file. However, when many locks are in contention, the TPOverseer constantly awakens processes when other locks in the file are being released. As a result, a lock will be retried (and the transaction holding it will be aborted), often without a sufficient wait period.

The minimum wait period **always** expires before a transaction is aborted, therefore eliminating the performance degradation that occurs when you do not set `allow_deadlocks` (the default behavior).

Lock-Conflict Logging Changes

The `lock_conflicts` parameter of the `analyze_system` request `set_tp_param` controls lock-conflict logging. For information about this parameter, see the `set_tp_param` request in the *Software Release Bulletin: VOS Release 14.0.0 (R914)*.

> **Note:** By default, log messages are written only to the system error log file, `>system>syserr_log.`*`date`*, and not to the system console. You can specify that log messages are written to the system console by using the `syserr_action` parameter of the `set_tp_param` request of the `analyze_system` command. For information about this parameter, see the `set_tp_param` request in the *Software Release Bulletin: VOS Release 14.0.0 (R914)*.

Transaction Metering Improvements

The transaction-metering capability has been extended in the `analyze_system` command so that users can monitor the number of pre-emptive transaction aborts and reasons for the aborts as well as related information. Also, the command now meters the amount of time a user process holds read locks and the time that the TPOverseer takes to release write locks.

See ‘‘New TPF-Related `analyze_system` Requests” in the *Software Release Bulletin: VOS Release 14.0.0 (R914)*.

Faster Key Searching

Key searching has been made significantly faster.

Deadlock Detection

TPF now detects deadlocks. TPF logs information in the file `syserr_log.`*`date`* about the two transactions involved in a deadlock and information about the lock involved. You control deadlock detection by using the following values of the `deadlock` parameter of the `set_tp_param` request (of the `analyze_system` command).

- The value `off` turns off deadlock detection.
- The value `on` specifies that deadlocks should be detected but not reported. This is the default value.
- The value `log` specifies that deadlocks should be detected and reported.

For information about this parameter, see the set_tp_param request in the *Software Release Bulletin: VOS Release 14.0.0 (R914)*.

When TPF detects a deadlock during lock arbitration and if the running transaction meets higher criteria than the other transaction in the deadlock, the running transaction will immediately abort the other transaction instead of waiting (this behavior occurs whether or not you specify the `-allow_deadlocks` argument of the command `set_tp_default_parameters` or `set_tp_parameters`). If the running transaction does not meet higher criteria, it does not abort; it waits for the transaction of higher criteria to retry the lock (within five seconds, the default amount of time), notice the deadlock, and abort the running transaction.

Transaction Priority Enhancements

The behavior of some priority values has changed to indicate that a wait should occur before aborting a lower-priority transaction due to lock conflict.

- For transactions with priorities between 0 and 4, the transaction desiring the lock will wait for up to five seconds (by default) before aborting a lower-priority transaction. You can change the default value using the `abort_wait_time` parameter of the `set_tp_param` request of the `analyze_system` command. Setting the default value to 0 causes behavior identical to that of prior releases: a higher-priority transaction desiring a lock will immediately abort a lower-priority transaction.
- For transactions with priorities between 5 and 9 (or if the lower-priority transaction is -1), the transaction desiring the lock will immediately abort a lower-priority transaction. This behavior has not changed from that of prior releases.

You control priority values by using the `abort_priority` parameter of the `set_tp_param` request of the `analyze_system` command. For information about the `abort_priority` and `abort_wait_time` parameters, see the `set_tp_param` request in the *Software Release Bulletin: VOS Release 14.0.0 (R914).*

I/O Operations and Aborts

The behavior of an I/O operation now differs from the behavior of an I/O operation in VOS releases prior to VOS Release 14.0.0. An I/O operation that requests a zero wait time, either implicitly or explicitly, now returns control to the application and returns the error message `e$file_in_use` (`1084`), `e$key_in_use` (`2918`), or `e$record_in_use` (`2408`) if a needed lock cannot be obtained; the I/O operation does **not** abort any transactions. In prior releases, an I/O operation requesting a zero wait time immediately aborted any transaction that held the desired lock and did not meet higher criteria. This change does not affect nonzero wait-time operations. Note that this new behavior occurs only if priority and start-time criteria do not apply. For example, a transaction with a priority between 0 and 4 that desires a lock will abort a lower-priority transaction after waiting up to five seconds (by default), while a transaction with a priority between 5 and 9 that desires the lock will immediately abort a lower-priority transaction, as described in ‘‘Transaction Priority Enhancements” earlier in this chapter.

TP Lock Manager

A new TP lock manager distributes locks on a priority/first-in, first-out (FIFO) basis, which means that TP locks are managed more efficiently. When a transaction releases a lock, the TP lock manager considers the following:

- It considers the specific lock for which a transaction is waiting. The transaction continues to sleep until that specific lock has been released. Therefore, the TP lock manager saves useless execution: a transaction does not waste time by attempting to obtain a lock that is not yet available.

- If multiple transactions are waiting for the same lock, the TP lock manager considers which transaction meets higher criteria. This behavior decreases the number of pre-emptive transaction aborts, therefore increasing throughput in any situation involving lock contention, regardless of CPU usage. The TP lock manager thereby eliminates the possibility of a transaction of lower criteria getting a lock first, only to be aborted at a later time.

Processing Delays Eliminated during Transaction Log Switches

TPOverseer performance has been improved to reduce or eliminate processing delays during transaction log switches. Heavy users of TPF should notice the following:

- little or no delay in operations during log switches
- much faster log switches
- fewer peaks in disk activity that could contribute to poor response time

Sample Application

Appendix B contains annotated program code for a sample application called the Automated Auction System. The annotations in the code are enclosed in asterisks and are identified with reference numbers. The reference numbers for the comments illustrating transaction protection features start with “TP.”

The Automated Auction System has two transaction files: the Bidder file and the Item file. The Bidder file contains bidder IDs, a limit on the amount of money each bidder can spend, and the total outstanding bids for each bidder. This last item is increased when a bidder submits a new high bid, and it is decreased when another bidder outbids that high bid.

The Item file contains information about the items up for sale. The file contains, for each auction item, an item ID, a description of the item, the current high bid, and the bidder ID of the highest bidder. The last two items are updated whenever a new bid is accepted for an item.

The server program starts two types of transactions. One type of transaction handles a user request for information about an auction item. If this type of transaction is successful, a description of the item and the dollar amount of the last bid accepted against that item are displayed on the user’s screen. This transaction performs a keyed read of the Item file. No file updates are performed.

The other type of transaction handles a user bid submission. The transaction performs keyed reads and rewrites to both the Item and the Bidder files. If this type of transaction is successful, both transaction files are updated. The updates to the two files are combined into a single transaction to ensure that the two files are always synchronized.

The program gives transactions involving bids greater than $10,000 a higher priority than other transactions.

Chapter 5: Queues

This chapter contains the following sections.

Introduction to Queues

A *queue* is a special type of file or file-like system object that acts as a communication medium between processes or tasks. One process or task puts a message in a queue, and another process or task reads or retrieves the message from the queue. All messages have priorities assigned to them, allowing messages of greatest importance to be serviced first.

Queues are usually the primary method of communication among requesters and servers in an application using the requester/server design model. A *requester program* is a process or task that puts messages into a queue. A *server program* is a process that takes messages out of a queue and services them. The application defines what constitutes servicing a message. Server programs generally should not be tasking processes. The high volume of I/O usually performed by servers makes it more efficient to execute each server program in its own process.

When a requester program adds a message to a queue, this action is called *sending the message*. When a server program gets a message from a queue in order to service it, this action is called *receiving the message*.

Every message has a *message status* that indicates whether the message has been received or if it is still waiting to be received. A status of *busy* means that the message was received and is currently being serviced or servicing is complete. A status of *nonbusy* means that the message has not been received yet. When a requester program sends a message to a queue, the message status is nonbusy. A server program can receive only nonbusy messages.

Another type of message access is known as *reading a message*. “Reading” means accessing the message without changing the message status. Both busy and nonbusy messages can be read, and the same message can be read many times. (You cannot read messages in direct queues.)

You must use subroutines, not language I/O statements, to perform operations on queues. The same set of subroutines is used to access all types of queues. For example, messages are sent to any type of queue by calling the `s$msg_send` subroutine, and messages are received from any type of queue by calling the `s$msg_receive` subroutine.

This section includes the following topics.

- ‘‘One-Way and Two-Way Queues”
- ‘‘Queue Structure”
- ‘‘Queue Types”
- ‘‘Subroutines Valid for Each Queue Type”
- ‘‘The Message ID”
- ‘‘Message Priorities and Message Order”
- ‘‘The `list_messages` Command”

One-Way and Two-Way Queues

A *one-way queue* handles messages, but it does not handle replies to messages. The program sending the messages does not expect or wait for replies. Requester programs send messages to the queue. Server programs receive messages from the queue.

A *two-way queue* handles messages and replies to those messages. Requester programs send messages to the queue and expect replies. Server programs receive a message from the queue and then *send a reply* to a message that it has received. The reply replaces the message in the queue. The requester program that sent the original message is the only requester that can *receive the reply.*

One-way queues use less resources than two-way queues. Both one-way and two-way queues allow multiple requester and server programs to be attached to the same queue.

Queue Structure

A queue has an established file structure. Each message unit in a queue consists of a header and a message.

The *message* is the character string sent by the requester program. The sending program defines the message length and message format through the variables required by the `s$msg_send` subroutine. On two-way queues, the reply replaces the original message in the queue.

The *header* contains information such as the message priority, the message ID, the process IDs of both the requester and the server programs, time and date data, and various switches. The header format is predefined. Most information in the header is provided by the operating system. Programs can obtain header information by using the `s$msg_read` and `s$msg_receive` subroutines.

The header is created when the message is sent. On a one-way queue, the header is deleted when the message is deleted. On two-way queues, some fields in the header are updated when the reply is sent (when the reply replaces the message). When the reply is received and deleted from the queue, the header is also deleted.

The location of the header and the message is one of the major differences among the various types of queues. These locations impact queue I/O processing speed.

Queue Types

The operating system supports the following queue types.

- Message queues. These queues are the least restrictive in terms of the actions that can be performed on a queue. Message queues are always one-way queues. Both the header and the message reside on disk. Message queues are the slowest type of queue.
- One-way and two-way server queues. These queues are more restrictive than message queues in terms of the actions that can be performed on a queue. The message header resides in the paged heap portion of memory, and the message text resides in disk cache. Server queues are faster than message queues but slower than direct queues.
- One-way and two-way direct queues. These queues are the fastest type of queue, but also the most restrictive in terms of the actions that can be performed on a queue. Both the header and the message reside in wired memory. Each requester port attached to the queue can handle only one message at a time.

Table 5-1 summarizes the major characteristics of the various queue types supported by the operating system. Each type is discussed in detail later in this chapter.

Table 5-1. Major Features of Queues *(Page 1 of 3)*

Queue Type	**Features**
Message	Transaction protection is allowed on the queue. Messages remain in the queue until explicitly deleted. Replies are not handled. This is the most flexible queue type in terms of operations permitted by requester and server programs. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, it is 2^{20} bytes.

Table 5-1. Major Features of Queues *(Page 2 of 3)*

Queue Type	Features
Two-Way Server	Transaction protection is transmitted by the queue. A message is replaced by a reply; the reply and message header are deleted when the reply is received. If a requester closes a queue, all messages entered by that requester and all replies to that requester are deleted from the queue, regardless of the message status. Messages sent by other requesters remain unaffected. A reply is required. This queue type is faster than a message queue. The sender can wait or cancel if no receiver is available. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, it is 2^{20} bytes. The maximum number of messages per module depends on the module's memory capacity and the maximum queue depth established for the queue.
One-Way Server	No transaction protection is offered. Messages are deleted when they are received. Messages entered by a requester are deleted if all requesters and servers with ports attached to the queue close the queue. Replies are not handled. This queue type is faster than a message queue or a two-way server queue. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, it is 2^{20} bytes. The maximum number of messages depends on the module's memory capacity and the maximum queue depth established for the queue.
Two-Way Direct	No transaction protection is offered. A reply is required. This queue type is faster than a two-way server queue. The queue and all of its server programs must reside on the same module. A message is replaced by a reply; the reply and the header are deleted when the reply is received. If no server program is active, a message cannot be sent. The maximum message length is 3,072 bytes (or less, if so specified when the queue is opened). The maximum number of messages is one message for each port attached to the queue. A requester cannot send a second message before receiving a reply to the previous message.

Table 5-1. Major Features of Queues *(Page 3 of 3)*

Queue Type	Features
One-Way Direct	No transaction protection is offered. Replies are not handled. This queue type is the fastest queue type. The queue and all of its server programs must reside on the same module. A message is deleted when it is received. If no server program is active, a message cannot be sent. The maximum message length is 3,072 bytes (or less, if so specified when the queue is opened). The maximum number of messages is one message for each port attached to the queue. The sender cannot send a second message until a server has received the previous message.

Subroutines Valid for Each Queue Type

Table 5-2 shows the queue-related subroutines that are valid for each queue type.

Table 5-2. Subroutines Valid for Each Queue Type

Queue Type	Valid Requester Subroutines	Valid Server Subroutines

Message	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
	`s$msg_receive`
	`s$msg_rewrite`
	`s$msg_cancel_receive`
	`s$msg_delete`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`
		`s$msg_rewrite`
		`s$msg_cancel_receive`
		`s$msg_delete`
		`s$truncate_queue`

Two-Way Server	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
	`s$msg_receive_reply`
	`s$call_server`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`
		`s$msg_send_reply`
		`s$msg_cancel_receive`

One-Way Server	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`

Two-Way Direct	`s$call_server`
	`s$msg_open_direct`
	`s$msg_receive_reply`
	`s$msg_send`
	`s$control ABORT_OPCODE (3)`
		`s$msg_open_direct`
		`s$msg_receive`
		`s$msg_send_reply`

One-Way Direct	`s$msg_open_direct` `s$msg_send`	`s$msg_open_direct` `s$msg_receive`

The Message ID

The operating system assigns a message ID to each message when the message is sent to the queue. Message IDs are assigned sequentially, beginning with 1. A separate numbering scheme is maintained for each queue.

When a message is sent (using the `s$msg_send` subroutine), the message ID is returned. For two-way queues, the returned message ID can then be used to receive a reply.

Message Priorities and Message Order

Every message has a priority associated with it. The priority value is an integer between 0 and 19, inclusive. Higher numbers represent greater urgency. The requester program assigns the priority when it sends the message (using the `s$msg_send` subroutine).

The messages in a queue are ordered by the operating system. The order of messages is based on (1) the priority assigned to each message when it is sent, and (2) the time that the message was sent. Messages with higher numbered priorities are placed first in the ordering scheme. Therefore, a high-priority message can be inserted near the "top" of the queue at any time. Among messages that have the same priority, the ones entered first are higher in the ordering scheme.

The `s$msg_read` and `s$msg_receive` subroutines access the messages in a queue. Both of these subroutines require the `msg_selector` argument, which specifies what message to read or receive. The values for `msg_selector` are described below.

- *`first_message_non_busy`*. This is the first nonbusy message in the ordering scheme (that is, the oldest nonbusy message with the highest priority).
- *`this_message_non_busy`*. This is a specific nonbusy message, identified by its message ID. If this value is specified for a message that is busy, the error code `e$message_busy` (`2750`) is returned.
- *`next_message_non_busy`*. This is the next nonbusy message in the ordering scheme following the one just accessed.
- *`first_message`*. This is the first message, busy or nonbusy, in the ordering scheme.
- *`this_message`*. This is a specific message, identified by its message ID. The message can be busy or nonbusy.
- *`next_message`*. This is the next message in the ordering scheme following the one just accessed. The message can be busy or nonbusy.

The first three values in the above list access only nonbusy messages. The `msg_selector` argument of the `s$msg_receive` subroutine is restricted to these three values, since only nonbusy messages can be received. The last three values do not distinguish between busy and nonbusy messages. The `s$msg_read` subroutine can use all six of the `msg_selector` values.

The *`next_message`* and *`next_message_non_busy`* values may seem like logical choices for sequentially scanning a message or server queue. However, if new messages are constantly being added to the queue during the sequential scan, those two values may not result in selecting the next message of highest priority. Refer to the ‘‘Accessing Messages Sequentially’’ section later in this chapter.

For direct queues, the only valid value for `msg_selector` is *`first_message_non_busy`*.

The `list_messages` Command

Because of a queue’s unique structure, a queue file image cannot be displayed online or printed. However, information about the messages in message or server queues can be displayed online in a standardized format, using the `list_messages` command. The command does not work for direct queues. The command displays information from the message header, such as message priority and message ID. It optionally displays the message text.

The `list_messages` command displays messages by priority. When priorities are equal, the oldest message appears first.

Figure 5-1 shows the results of a `list_messages` command, with no optional arguments specified, on a two-way server queue.

```
list_messages server2
  ID  TIME QUEUED             P  REQUESTER              FLAGS
  8  89-12-05 10:55:15 EST  9 Riversides.Sample     busy,reply_requested
   1  89-12-05 10:51:38 EST   4  Riverside.Sample         reply_requested
   5  89-12-05 10:56:05 EST   4  Riverside.Sample         reply_requested

Messages in queue: 3 (1 busy)
```

Figure 5-1. Output from the `list_messages` Command

Some of the flags from the message header that might be displayed by the `list_messages` command are explained below.

- `busy`. The message was received by a server program.
- `been_busy`. The message was received at least once by a server program, which later canceled its receive of the message. Since `been_busy` is a nonbusy status, the message can be received again.
- `reply_requested`. The queue is a two-way queue, and a reply has not been sent.
- `requester_aborted`. The requester program that sent the message closed the queue.

The `list_messages` command sequentially scans a queue. If the queue’s contents are dynamically changing while the `list_messages` command is executing, the resulting display might not match the true contents of the queue, and the totals in the last line of the display might not match the messages listed in the display.

Using Queues

This section includes the following topics.

- ‘‘Preliminary Steps for Sending Messages”
- ‘‘File Organization and I/O Type”
- ‘‘Messages and Ports”

Preliminary Steps for Sending Messages

A program cannot use a queue until the following steps have been performed.

- The queue must be created. A queue is created just like any other VOS file is created, by using the `create_file` command or the `s$create_file` subroutine (described, respectively, in the *VOS Commands Reference Manual (R098)* and the VOS Subroutines manuals).

- Requester and server programs must have access to the queue. Access rights to a queue are assigned in the same way that access is assigned to other files, by using the `give_access` and `give_default_access` commands. Every file (and queue) has an access control list associated with it. The `display_access_list` command shows the access control list for a file. For a detailed discussion of access rights, see the *VOS Commands User’s Guide (R089).*

  With a few exceptions, the execute, read, and write file-access types provide the same level of access. The exceptions are the access requested by the `s$msg_read` and `s$msg_receive` subroutines, and the `list_messages` command and monitor task request. For those access attempts, the access type given to the process determines which messages the process can read and receive. Some access types allow the process to access all messages in the queue; other access types allow access only to messages sent by a process whose person name matches the caller’s person name. See the explanations of `s$msg_read`, `s$msg_receive`, and `list_messages` in the VOS Transaction Processing Facility Reference manuals for more specific information. Any process with read, write, or execute access rights to a queue can write to the queue using the `s$msg_send` subroutine.

- Requester and server programs must each attach a port (or ports) to the queue, using either the `attach_port` command or the `s$attach_port` subroutine (described, respectively, in the *VOS Commands Reference Manual (R098)* and the VOS Subroutines manuals).

- Each port attached to a queue must open the queue, using either `s$msg_open` (for server queues and message queues) or `s$msg_open_direct` (for direct queues).

File Organization and I/O Type

The `create_file` command and `s$create_file` subroutine require a file organization to be specified when a file is created. Table 5-3 shows the value required for the organization argument when creating the various types of queues. The numbers in parentheses are the codes that should be used in the `file_organization` argument of the `s$create_file` subroutine to create queues.

Each port attached to a queue must open the queue. The subroutines used for this action require that the port opening the queue identify itself as a specific I/O type. The I/O type indicates how the port will use the queue. Table 5-3 shows the valid I/O types for each queue type. The numbers in parentheses are the codes that should be used in the `io_type` argument of the `s$msg_open` or `s$msg_open_direct` subroutine when a port opens a queue.

Explanations of the `io_type` arguments follow.

- *Requester*. This is a requester port that can send messages and receive replies.
- *Server*. This is a server port that can receive message and send replies.
- *Send-only requester*. This is a requester port that can only send messages.
- *Receive-only server*. This is a server port that can only receive messages.
- *Server notify-one*. This is a server port that can receive messages and send replies. Use this value when the single-event notify feature is desired on a two-way server queue. *Single-event notify* means that only one server port “wakes up” to service a new message when there are more servers than there are messages to be served.
- *Receive-only server notify-one*. A server port that can only receive messages. Use this value when the single-event notify feature is desired on a one-way server queue.

Table 5-3. File Organizations and I/O Types for Each Queue Type

Queue Type	File Organization	I/O Types	
		Requester Ports	**Server Ports**
Message Queue	message queue (6)	requester (5)	server (6)
Two-Way Server Queue	server queue (5)	requester (5)	server (6) or server notify one (12)
One-Way Server Queue	one-way server queue (31)	send-only requester (7)	receive-only server (8) or receive-only server notify one (13)
Two-Way Direct Queue	sequential (3)	requester (5)	server (6)
One-Way Direct Queue	sequential (3)	send-only requester (7)	receive-only server (8)

Messages and Ports

Each requester and server program must attach at least one port to each queue that it wants to use. Each program can attach multiple ports to the same queue. A requester program, for example, might attach two ports to a queue and alternate between the ports to send messages. In a tasking process, each task might attach its own port to the queue. Alternatively, all tasks could share the same port or ports.

Some queue I/O actions are port sensitive. Requesters using two-way queues must receive the reply to a message on the same port on which the message was originally sent. Servers using two-way queues must send a reply on the same port on which the message was received. Requesters and servers using message queues can rewrite messages only on the port on which the message was received.

The queue I/O action of reading messages is not port sensitive.

Other port-related issues that may affect your application are listed below.

- On direct queues, only one message is allowed per port. The message must be completely serviced and, on two-way direct queues, the reply must be received, before another message can be sent on the same port.
- On two-way server queues, if a requester port is detached or closed, messages (or replies) that were sent on the detached port are deleted from the queue, regardless of the message status. For more information, see the ‘‘Closing Server Queue Ports” section later in this chapter.

A port attached to a queue can exist in three states.

- Wait Mode. This is the default state of a port when a port is attached. When a port is in wait mode, a queue I/O subroutine waits for the I/O operation to finish before returning to its caller. The subroutine waits indefinitely. A port that was set to one of the other two states can be returned to wait mode with the `s$set_wait_mode` subroutine.
- No-Wait Mode. When a port is in no-wait mode, the queue I/O subroutines return immediately. If the I/O operation was not complete, the subroutine returns with the error code `e$caller_must_wait` (`1277`). A port is set to no-wait mode with the `s$set_no_wait_mode` subroutine. The `s$set_no_wait_mode` subroutine establishes a system event associated with the port. The operating system notifies this event when an I/O operation completes on the port.
- Time Limit. A time limit can be specified that determines how long an I/O subroutine waits for the I/O to complete before returning to its caller. If the I/O operation does not finish within the specified time, the subroutine returns with the error code `e$timeout` (`1081`). A time limit is set on a port with the `s$set_io_time_limit` subroutine.

  On message and server queues, timeouts that occur while reading messages, receiving messages, or receiving replies may affect other messages in the queue that were previously sent on the timed-out port. For more information, see the ‘‘Setting I/O Time Limits on Queues” section later in this chapter.

Message Queues

Message queues are always one-way queues. Requester programs can send messages to and receive messages from this type of queue. Requester and server programs can receive and rewrite (essentially resend) messages. Therefore, message queues can pass messages among requester programs, among server programs, or among requesters and servers.

Each program attached to a message queue adds messages to the queue independently of other programs. The program receiving the message does not need to be executing for the sender to successfully add messages to the queue.

In a message queue, both the header and the message reside on disk. Subsequent access to a message requires disk access, making this the slowest queue type. The disk storage method also makes message queues act more like normal data files than the other queue types. Messages remain in the queue until they are explicitly deleted with the `s$msg_delete` subroutine. All messages remain in a message queue even if one or all of the programs using the queue stop, or if one or all of the ports attached to the queue are closed.

An example of message queue use is the operating system's print and batch processing queues. User processes send requests to these message queues, and the operating system receives messages from the queue and services the requests.

This section includes the following topics.

Creating and Opening Message Queues

To create a message queue, use the `s$create_file` subroutine or the `create_file` command. The file organization argument must be set to `message_queue (6)`. Ports attached to a message queue open the queue using the `s$msg_open` subroutine. Requester ports must open the queue with the `io_type` argument set to `requester (5)`. Server ports must open the queue with the `io_type` argument set to `server (6)`.

Queue Size

For a message queue, the maximum message length is 2^{23} bytes when the queue and the calling process are located on the same module, and 2^{20} bytes when the queue and the calling process are located on different modules.

There are no limitations on the number of messages in the queue except for the amount of space available on the disk where the queue resides.

Basic Operations on Message Queues

Requester programs can send messages to a message queue using the s$msg_send subroutine. Requesters can read messages using the s$msg_read subroutine, and they can receive messages and cancel receives using the s$msg_receive and s$msg_cancel_receive subroutines. The message queue is the only queue type on which requesters can receive messages.

Server programs can read messages, receive messages, and cancel receives by using the smsg_read, smsg_receive, and s$msg_cancel_receive subroutines, respectively.

To *cancel a receive* means to change a busy message status back to nonbusy, so the message can be received again. A server program might cancel a receive if it could not service the message, thus allowing another server program to receive the message and attempt to service it.

The message queue is the only queue for which the concept of rewriting a message exists. Both requester and server programs can rewrite messages, using the s$msg_rewrite subroutine. When a rewrite occurs, the text specified for the rewrite replaces the text of the original message. The original message is lost, but the original message ID is retained. A message must be rewritten on the same port on which the message was received.

The msg_priority argument of the s$msg_rewrite subroutine must be either -1 or a valid priority. If the priority specified is -1 or is the same as the original priority of the message being rewritten, then the call to s$msg_rewrite does not change the position of the message in the queue. However, if msg_priority is different from the original priority of the message and is not -1, then the rewritten message is repositioned in the queue.

A message queue is deleted with the delete_file command or the s$delete_file subroutine.

Deleting Message Queue Messages

Messages remain in message queues until an application program deletes them. Both requester and server programs can call the s$msg_delete subroutine to delete a message. A program can delete only those messages that it has received.

The space released when a message is deleted is reused by other messages. However, the message IDs continue to increment. Use the s$truncate_queue subroutine to reset the message ID to 1 and to release space in the index that points to message locations. The s$truncate_queue subroutine can only be called by a server program for an empty message queue. (All messages in the queue must be received and deleted before the queue is truncated.)

Closing Message Queue Ports

Requester and server programs can close the ports attached to a message queue with the `s$close` subroutine. The subroutine requires a port ID as an argument. To reopen the port, the program calls `s$msg_open`. Closing a port is different from detaching a port. When `s$close` is called for a port attached to a queue, the port remains attached but not open. Closing a port makes the queue inaccessible on that port.

When a requester program closes a message queue port, there is no effect on messages in the queue.

When a server program closes a message queue port, the status of messages that were previously received on the closed port is changed from busy to been busy. The been busy status is considered nonbusy, which means that the message can be received again. To prevent messages from being received again, the server program should delete received messages, using the `s$msg_delete` subroutine, before closing a port.

Transaction Protection on Message Queues

Transaction protection is allowed on message queues. To protect a message queue, declare the queue a transaction file and then perform all I/O on the queue within transactions. In a requester/server application, both the requester and server programs must use transactions to access the queue.

To implement this feature, the program sending the message calls `s$start_transaction`, enters a message into the message queue, and then either commits or aborts the transaction. The message is unavailable to other programs until the transaction is committed. If the transaction is aborted, the message is never available to other transactions.

The program receiving the message calls `s$start_transaction`, then `s$msg_receive`, and then either commits or aborts the transaction. The message is unavailable to other programs during the transaction. If the transaction is committed, the message is marked busy (received). If the transaction is aborted, the message is returned to a nonbusy status, so that it can be received by another program.

Using `s$msg_read` on Transaction-Protected Message Queues

Some aspects of lock arbitration work differently when applied to transaction-protected message queues. Consider a program that continuously creates transactions that call `s$msg_read` to scan the messages in a queue. When the program has read all of the messages (or encounters an empty queue on the first read), the current transaction waits for a new message to be added to the queue. While it waits, the transaction owns a read lock on the record it is waiting for. Since the queue is transaction protected, the sending transaction cannot send (write) any new messages because it cannot obtain the necessary write lock on the next record. A lock conflict exists.

If you set arbitration parameters so that the sending transaction always wins the lock conflict, `s$msg_read` does **not** abort. (You would normally expect the losing transaction that previously owned a lock to abort, but this aspect of lock arbitration works differently for message queues.) Instead, the reading transaction receives the error code `e$key_in_use` (`2918`). The sending transaction can then write its message.

You can program the reading transaction to wait and retry the read when it receives the `e$key_in_use` error code. The key referred to in the `e$key_in_use` error code is the message ID of the next message to be added to the queue.

Message Queue Scenarios

This section contains scenarios illustrating requester and server operations on message queues. Assume that the queues are empty at the start of each scenario, that only one requester and one server program are executing, and that the requester and server ports attached to the queue are in wait mode. With those assumptions, you can think of each scenario as a description of what happens to a single message.

The first scenario illustrates a requester sending a message to a message queue and a server receiving, servicing, and deleting the message.

The second scenario illustrates a requester sending a message to a transaction-protected message queue and the server receiving the message. Assume that the queue was declared a transaction file, which means that all queue I/O must be performed within transactions.

The scenario includes two transactions: one in the requester program and one in the server program. Note that the message sent by the requester is not visible to the server (and, therefore cannot be received) until the requester's transaction is successfully committed. (If other messages exist in the queue, the server could receive those other messages.)

Scenario 1 - Empty Message Queue

Requester Program (wait mode)	**Server Program (wait mode)**
	1. Call `s$msg_open` (`io_type` = `6`). Open the queue for a server port.
	2. Call `s$msg_receive`. a. Wait for a message.
1. Call `s$msg_open` (`io_type = 5`). Open the queue for a requester port.	
2. Call `s$msg_send`. Add a message to the queue.	
	b. Receive a message.
	3. Service the message.
	4. Call `s$msg_delete`. Remove the message from the queue.

Scenario 2 - Empty Transaction-Protected Message Queue

Requester Program (wait mode)	Server Program (wait mode)
1. Call `s$msg_open` (`io_type = 5`). Open the queue for a requester port.	1. Call `s$msg_open` (`io_type = 6`). Open the queue for a server port.
	2. Call `s$start_transaction`. Start a transaction
	3. Call `s$msg_receive`. a. Wait for a message.
2. Call `s$start_transaction`. Start a transaction.	
3. Call `s$msg_send`. Add a message to the queue.	
4. Call `s$commit_transaction`. End the transaction and make the message accessible to the server program.	
	b. Receive the message.
	4. Service the message.
	5. Call `s$commit_transaction`. End the transaction and make any updates visible to other ransactions.

Server Queues

Server queue message headers are stored in the paged heap portion of memory. (Paged heap refers to an area of shared virtual memory on a module.) The message text is stored in disk cache. Depending on resource availability, the text in the cache might be written to disk. This storage arrangement makes server queue I/O faster than message queue I/O, but slower than direct queue I/O.

There are two types of server queues: one-way and two-way. One-way queues are faster than two-way queues.

This section includes the following topics.

- ‘‘Creating and Opening Server Queues”
- ‘‘Single-Event Notify for Servers”
- ‘‘Message Priorities on Server Queues”
- ‘‘Basic Operations on Server Queues”
- ‘‘Closing Server Queue Ports”
- ‘‘The `s$call_server` Subroutine for Two-Way Queues”
- ‘‘Deleted Messages”
- ‘‘Inactive Server Programs”
- ‘‘Queue Size”

- ‘‘Transaction Protection and Two-Way Server Queues”
- ‘‘Server Queue Scenarios”
- ‘‘Starting Transactions in the Server with Two-Way Server Queues”

Creating and Opening Server Queues

Create a server queue using the `s$create_file` subroutine or the `create_file` command. To create a one-way server queue, the organization argument must be set to one-way server queue (31). To create a two-way server queue, the organization argument must be set to server queue (5).

Ports attached to a server queue open the queue using the `s$msg_open` subroutine. For a two-way server queue, requester ports must open the queue with the `io_type` argument set to requester (5). Server ports must open the queue with `io_type` set to server (6) or server notify-one (12).

For a one-way server queue, the requester ports must open the queue with the `io_type` argument set to send-only requester (7). Server ports must open the queue with `io_type` set to receive-only server (8) or receive-only server notify-one (13).

Single-Event Notify for Servers

Single-event notify can reduce overhead in applications with more than one server process waiting on a given queue. Single-event notify is especially useful in applications where the number of unprocessed messages is small in relation to the number of server processes.

In an application that does **not** use single-event notify, whenever a new message is placed in a server queue, all active server processes are notified. Usually, the first server process to awaken performs the necessary work, and the other processes are suspended once again. This resumption and suspension of the processes that do not handle the message wastes system resources.

In an application that uses single-event notify, this unnecessary resumption and suspension of all active server processes is eliminated. Although all active server processes are notified, only the process that obtains the lock responds to service the message. Because that process decrements a notify-one-event counter, the other processes do not respond to the notification. Although an application enables single-event notify, the operating system automatically (and temporarily) disables single-event notify functionality if the number of unprocessed messages is large in relation to the number of server processes. Therefore, message-servicing performance is preserved.

Single-event notify is implemented by server programs through the `io_type` argument of the `s$msg_open` subroutine. To implement the feature, the server ports use an `io_type` of server notify-one to open a two-way server queue, and an `io_type` of receive-only server notify-one to open a one-way server queue. All server ports connected to the queue in all server processes must specify single-event notify before the capability is implemented.

Message Priorities on Server Queues

Special significance is attached to priorities 10 through 19 for one-way and two-way server queues. If no server programs are active, messages with those priorities are not added to the queue. The `s$msg_send` subroutine returns the error code `e$no_msg_server_for_queue` (`2817`). This concept is extended further for two-way queues, when the call to `s$msg_send` is successful (at least one server is active when the requester calls `s$msg_send`), but then all server programs stop before the message is received. In that situation, when the requester calls `s$msg_receive_reply`, the error code `e$no_msg_server_for_queue` (`2817`) is returned.

Messages with priorities of 0 through 9 that are sent to server queues are added to the queue regardless of whether a server program is currently available to service them. For two-way queues, if no servers are active or if all servers stop before the message is received, the requester waits when it calls `s$msg_receive_reply` until a server process receives and replies to the message (if the port is in wait mode).

In all other respects, the priority values 0 through 19 are treated the same. A higher number gives the message a higher priority. For example, a message with priority 16 has precedence over messages with priorities 6 and 15.

Basic Operations on Server Queues

Requester programs using one-way or two-way server queues can read and send messages using the `s$msg_read` and `s$msg_send` subroutines. A requester program using a two-way server queue can also receive replies to messages that it sent, using the `s$msg_receive_reply` subroutine. The requester program for a two-way server queue can use the `s$call_server` subroutine, which is a combination of the `s$msg_send` and `s$msg_receive_reply` subroutines.

Server programs using one-way or two-way server queues can read and receive messages using the `s$msg_read` and `s$msg_receive` subroutines. A server program using a two-way server queue can also cancel receives and send replies using the `s$msg_cancel_receive` and `s$msg_send_reply` subroutines.

A server queue is deleted with the `delete_file` command or the `s$delete_file` subroutine.

Closing Server Queue Ports

Requester and server programs can close the ports attached to a server queue with the `s$close` subroutine. The subroutine requires a port ID as an argument. To reopen the port, the program calls `s$msg_open`. Closing a port is different from detaching a port. When `s$close` is called for a port attached to a queue, the port remains attached but not open. Closing a port makes the queue inaccessible on that port.

When **all** ports attached to a one-way server queue are closed (that is, all requester and all server ports), all messages are deleted from the queue, regardless of the message status. Otherwise, if some ports are closed but at least one port remains open, messages in the queue are not affected.

When a requester program closes a port attached to a two-way server queue, all messages that were sent on the closed port are deleted from the queue. Also, all replies to messages that were sent on the closed port are deleted from the queue.

When a server program closes a port attached to a two-way server queue, there is no effect on messages that the server has not touched. There is also no effect on messages that the server has received and replied to. However, messages that were received but not replied to before the server port was closed are deleted. The requester receives the error code `e$server_aborted` (`2759`) when it attempts to receive replies to those deleted messages.

The `s$call_server` Subroutine for Two-Way Queues

The `s$call_server` subroutine can be called by a requester program accessing a two-way server queue. The subroutine combines the actions of `s$msg_send` and `s$msg_receive_reply`. It puts a message in a two-way queue and returns to the caller either the server's reply or an error code explaining why it does not have the reply.

The `s$call_server` subroutine is intended to be used primarily in wait mode. If the port used for `s$call_server` is in wait mode, then the subroutine does not return until a server has sent a reply to the message.

In general, `s$call_server` should not be used on two-way server queues in no-wait mode or with timeout limits. Use `s$msg_send` and `s$msg_receive_reply` instead. If the port used for `s$call_server` is in no-wait mode, then the subroutine returns immediately with the error code `e$caller_must_wait` (`1277`). The requester should not repeat the `s$call_server` call because that would put another message in the queue. The requester cannot use `s$msg_receive_reply`, since the message ID from the unsuccessful `s$call_server` call is unknown.

If the port has a time limit set on it, the subroutine waits for the time limit, tries again, and if it does not succeed, returns the error code `e$timeout` (`1081`). Refer to the ‘‘Setting I/O Time Limits on Queues’’ section later in this chapter for a discussion of various side effects that might occur when time limits are set on queues.

Deleted Messages

A message in a one-way server queue is deleted under either of the following circumstances:

- the message is received by a server program
- all ports attached to the queue are closed (using the `s$close` subroutine)

A message in a two-way server queue is deleted under either of the following circumstances:

- the reply to the message is received by the requester program (the sender)
- the requester port through which the message was sent is closed (using the `s$close` subroutine). This means that if the requester's port is closed before the message is received, the message is lost. If the message was received and replied to, but the reply was not received, then the reply is removed from the queue. If the message was received and not yet replied to (that is, the message is being processed), then the `requester_aborted` flag is turned on in the message header, and the server receives the error code `e$requester_aborted` (`2760`) when it attempts to send the reply. If other requester ports are still open, then the messages sent through the open ports

remain in the queue. If all requester ports attached to the queue are closed, then all messages in the queue are deleted.

Inactive Server Programs

If a server program is not attached to a server queue when a requester program sends a message, the following actions occur.

- If `s$msg_send` is used to send a message with a priority between 0 and 9 (low-priority messages), the message is added to the queue.
- If `s$call_server` is used to send a message with a priority between 0 and 9, the message is added to the queue. However, `s$call_server` then attempts to execute `s$msg_receive_reply`. Since no server program is running, there will be no message to receive. If the port is in wait mode, `s$call_server` waits indefinitely. The `s$call_server` subroutine should not be used in no-wait mode or with I/O time limits. Use `s$msg_send` and `s$msg_receive_reply` instead.
- If either `s$msg_send` or `s$call_server` is used to send messages with priorities between 10 and 19 (high-priority messages), the messages are not added to the queue. The error code `e$no_msg_server_for_queue` (`2817`) is returned.

Queue Size

The number of messages allowed in a server queue is related to the queue depth. The default maximum queue depth allowed in a server queue is 256. This number can be changed with the `s$set_max_queue_depth` subroutine or the `set_max_queue_depth` command. The range for *`max_queue_depth`* is from 1 to 32,767. The current value of *`max_queue_depth`* is returned as an output argument by the `s$get_file_status` and `s$get_open_file_info` subroutines and the `display_file_status` command. The *`max_queue_depth`* can also be obtained through the `analyze_system` subsystem, using the `dump_afte` request on an open queue.

Messages can accumulate in a server queue until the number of messages reaches the specified maximum queue depth. At this point, the following factors determine whether more messages can be added:

- the number of server programs running
- the priority values of the existing messages

See the explanation of the `s$set_max_queue_depth` subroutine in the VOS Transaction Processing Facility Reference manuals for more information about the maximum queue depth.

If the program must know when a server queue is full, set the queue's port in no-wait mode. When the program attempts to send a message to a queue that is full, the program gets the error code `e$caller_must_wait` (`1277`). The program should wait on the event, and then resend the message until either the message is successfully added to the queue or the subroutine fails for other reasons.

The maximum message length for a server queue is 2^{23} bytes when the queue and the calling process are located on the same module, and 2^{20} bytes when the queue and the calling process are located on different modules.

Transaction Protection and Two-Way Server Queues

A server queue cannot be declared a transaction file. Therefore, server-queue messages cannot be protected like message-queue messages. However, a unique transaction protection capability that allows a requester to transmit transaction protection to a server program is available when two-way server queues are used.

The requester starts a transaction and performs queue I/O within the bounds of the transaction. When a server program services a message sent to the queue as part of the requester's transaction, transaction protection is transmitted to the server program. That is, all I/O actions on transaction files performed by the server are protected while the server is servicing a message that originated in a requester transaction. This capability might be desirable when multiple server programs perform different functions and more than one server is necessary to service a single user request. You might also want to use this capability in an application in which the requester performs some data-file I/O, and then to complete the transaction, sends a message to a server program requesting updates on other (remote) data files. By using requester-transmitted transaction protection, all I/O performed to service a single user request is treated as a unit, no matter how many messages, server programs, or server queues are involved in the servicing. The local and remote data files are guaranteed to stay synchronized when all servicing of a single user request occurs within a single transaction. Requester-transmitted transaction protection allows the requester to control when to start and commit the transaction.

To transmit transaction protection using a two-way server queue, code the server program to receive, service, and reply to one message at a time. The requester program should use the `s$call_server` subroutine. If the requester program uses `s$msg_send` and `s$msg_receive_reply` instead, no transaction-protected operations should occur between the two subroutine calls for a single message.

A transaction can include one or more messages, and it may involve one or more server queues. If one transaction consists of several messages and the same server program can service all of the messages, then only one server queue is required. However, if each message requires a different server program, then several different server queues are required.

Details about coding a requester program to transmit transaction protection are provided below.

1. The requester program calls `s$start_transaction`.

2. The requester program might perform I/O on transaction files.

3. To initiate I/O performed by a server program, the requester program sends a message to a two-way server queue, preferably by using `s$call_server`. A server program receives the message, services it, and sends a reply.

   All I/O actions performed on transaction files by server programs are performed on behalf of the transaction that the requester started. (The server program inherits the TID of the transaction that was started by the requester.) I/O performed on transaction files is not visible to other transactions at this point.

4. The requester program receives a reply from the queue. From the reply, the requester program determines whether to continue or end the transaction.

5. The requester might continue the transaction by performing some transaction I/O itself, or by sending another message to the same or a different two-way server queue.

6. When the entire request is serviced, the requester ends the transaction. If the requester calls `s$commit_transaction`, all updates performed on behalf of the transaction, whether they were performed by the requester or the server, become visible to other transactions. If the requester calls `s$abort_transaction`, or if the transaction is not completed for any reason, then the I/O performed on behalf of the transaction never becomes permanent.

   Note: A server program can call `s$abort_transaction` to abort the transaction, or TP-Runtime might abort the transaction while a server program is servicing the message. In either situation, the server program's reply should indicate that the transaction was already aborted. The requester program can thereby avoid an attempt to commit or abort a transaction that was already aborted. This attempt would return the error code `e$tp_no_tid` (`2872`).

The following pseudocode illustrates a requester transaction that transmits transaction protection to a server program. The requester program performs local data-file updates and then sends a message to a server program through a two-way server queue. If either the server or requester program aborts the transaction, any updates made for the transaction by either program are backed out.

Real code must check error codes after every subroutine call. For simplicity, error code checking is omitted from the following pseudocode.

```
s$start_transaction
update local transaction data files
s$call_server (send message to a two-way server queue and receive
reply)
if server aborted transaction
   exit
else
   interpret reply received from server
   if reply is satisfactory and all previous error codes = 0
      s$commit_transaction
   else
      s$abort_transaction
```

Server Queue Scenarios

This section contains scenarios illustrating requester and server operations on server queues. Assume that the queues are empty at the start of each scenario and that the requester and server ports attached to the queue are in wait mode. For the first scenario, assume that only one requester program and one server program are executing. With those assumptions, you can think of the first scenario as a description of what happens to a single message. The second scenario includes two server ports and traces two messages.

In both scenarios, the server opens the queue before the requester sends a message. This order is important if the requester assigns values of 10 through 19 to message priorities. If messages with those priorities are sent before a server opens the queue, the messages are not added to the queue. If message priorities are always values of 0 through 9, then it does not matter whether a requester sends a message before a server port opens the queue.

The first scenario illustrates a one-way server queue. A requester program sends messages; a server program receives messages and services them.

The second scenario illustrates a requester program that uses a two-way server queue to transmit transaction protection to a server program. The requester sends two messages and waits for two replies for each transaction that it starts. The server performs I/O on behalf of the transaction and can abort the transaction. If the transaction is not aborted by the server, the requester decides whether to commit or abort it.

Scenario 1 - Empty One-Way Server Queue

Requester Program (wait mode)	**Server Program (wait mode)**
	1. Call `s$msg_open` (`io_type = 8`). Open the queue for a receive-only server port.
	2. Call `s$msg_receive`. a.Wait for a message.
1. Call `s$msg_open` (`io_type = 7`). Open the queue for a send-only requester port.	
2. Call `s$msg_send`. Add a message to the queue.	
	b. Receive a message.
	3. Service the message.

Scenario 2 - Empty Two-Way Server Queue Using Transaction Protection *(Page 1 of 2)*

Requester Program (wait mode)	**Server Program (wait mode)**
	1. Call `s$msg_open` (`io_type = 6`). Open the queue for server `port_one`.
1. Call `s$msg_open` (`io_type = 5`). Open the queue for arequester port.	
	2. Call `s$msg_receive`. a. Wait for a message.
2. Call `s$start_transaction`. Start a transaction.	
3. Call `s$msg_send`. Add a message to the queue.	
	b. Receive a message.
4. Call `s$msg_receive_reply`. a. Wait for reply.	
	3. Service the message. Call `s$abort_transaction` if transaction should be aborted.
	4. Call `s$msg_send_reply`. Remove the message from the queue and send a reply in its place.
b. Receive the reply.	
	Second Server Program (wait mode)
	1. Call `s$msg_open` (`io_type` is `6`). Open second queue for server `port_two`.
	2. Call s$msg_receive. a. Wait for a message.
5. Evaluate reply. If transaction was aborted by server, do not continue. Otherwise, continue.	
6. Call `s$msg_send` to a dd a message to the second queue.	
	b. Receive a message.
7. Call `s$message_receive_reply`. a. Wait for a reply.	

Scenario 2 - Empty Two-Way Server Queue Using Transaction Protection *(Page 2 of 2)*

Requester Program (wait mode)	**Server Program (wait mode)**
	3. Service the message. Call `s$abort_transaction` if transaction should be aborted.
	4. Call `s$msg_send_reply`. Remove the message from the queue and send a reply in its place.
b. Receive the reply.	
8. Evaluate reply. If the server has not aborted the transaction, call `s$abort_transaction` or `s$commit_transaction` to end the transaction.	

Starting Transactions in the Server with Two-Way Server Queues

A server program cannot start a transaction and then call `s$msg_receive` on a two-way server queue.

To protect a queue message and all work performed by a server program on behalf of the message, the requester program must start and commit the transaction. Either the requester or the server may abort the transaction. The preceding section describes this procedure in detail.

To protect the work performed by a server program but not the message itself, the server program may start the transaction. However, the server must call `s$msg_receive` before starting the transaction, not within the transaction. Otherwise, the subroutine returns the error message `e$tp_in_progress (2932)`. The following scenario shows the correct way for a server program to start a transaction that uses information in a message received from a two-way server queue.

Scenario 3 - Starting Transactions in the Server with Two-Way Server Queues

Requester Program	**Server Program**
`s$msg_send`	

	`s$msg_receive`
	`s$start_transaction`
	`service message`
	`s$commit_transaction or`
	`  s$abort_transaction`
	`s$msg_send_reply`

`s$msg_receive_reply`	

Direct Queues

Direct queues can be one-way or two-way queues. Both the header and the message portions of one-way and two-way direct queues reside in wired memory — nothing ever goes to disk. This storage method makes direct queues the fastest type of queue, but not the most efficient queue in terms of memory usage. However, memory for direct queues is allocated using a shared buffer scheme, which reduces memory usage to a manageable level for most applications requiring the speed advantages of direct queues. The shared buffer scheme is explained in detail in the ‘‘Programming Considerations” section later in this chapter.

When a direct queue is used in multimodule applications, the queue must reside on the **same** module on which all of the server processes are executing. The requester processes can be remote.

For any given requester port attached to a direct queue, one message exists. When a message is completely serviced, it is deleted from the queue, and another message can be sent on the port. For a one-way direct queue, a message sent on a requester port must be received by a server before the requester can send another message on the same port. For a two-way direct queue, the message must be received by a server, and the reply must be sent by the server and received by the requester before the requester can send another message on the same port.

For any given server port attached to a direct queue, one message exists. To process several messages concurrently, a server must use several ports.

> **Note:** Because of the way messages are buffered in wired memory, more than one message per port might exist on one-way direct queues. However, your program cannot depend on this situation either occurring or not occurring.

With direct queues, you have the option of omitting the header portion of messages. For a specific queue, either all of the messages will have headers, or no messages will have headers. The option is specified when the queue's server processes open ports to the queue. The `msg_header_version` argument of the `s$msg_open_direct` subroutine specifies whether or not space should be allocated for headers. If your application does not need header information, you can save space by choosing to omit the header.

Transaction protection is **not** available on direct queues.

This section include the following topics.

- ‘‘Creating and Opening Direct Queues”
- ‘‘Basic Operations on Direct Queues”
- ‘‘Closing Direct Queue Ports”
- ‘‘The `s$call_server` Subroutine for Two-Way Queues”
- ‘‘Queue Size”
- ‘‘Direct Queue Scenarios”

Creating and Opening Direct Queues

Create a one-way or two-way direct queue using the `s$create_file` subroutine or the `create_file` command. The organization argument must be set to sequential (3). The sequential file is not used to store messages. It only provides a way to name the queue. When `s$msg_open_direct` opens the queue, space is allocated in wired memory for the queue.

Open a direct queue by using the `s$msg_open_direct` subroutine. At least one server port must open a direct queue before any requesters open the queue. For a two-way direct queue, server ports must open the queue with the `io_type` argument set to server (6). Requester ports must open the queue with `io_type` set to requester (5).

For a one-way direct queue, the requester ports must open the queue with the `io_type` argument set to send-only requester (7). Server ports must open the queue with `io_type` set to receive-only server (8).

In a multitasking application where the requester tasks share a direct queue, the tasks can share a group of ports. This sharing of ports is especially appropriate when messages are large because it is an efficient use of wired memory space.

Basic Operations on Direct Queues

Requester programs using one-way or two-way direct queues can send messages using the `s$msg_send` subroutine. Also, a requester program using a two-way direct queue can receive replies to messages that it has sent, using the `s$msg_receive_reply` subroutine.

Server programs using one-way or two-way direct queues can receive messages using the `s$msg_receive` subroutine. Also, a server program using a two-way direct queue can send replies using the `s$msg_send_reply` subroutine.

A direct queue is deleted with the `delete_file` command or the `s$delete_file` subroutine.

Closing Direct Queue Ports

Requester and server programs can close the ports attached to a direct queue with the `s$close` subroutine. The subroutine requires a port ID as an argument. To reopen the port, the program calls `s$msg_open_direct`. Closing a port is different from detaching a port. When `s$close` is called for a port attached to a queue, the port remains attached but not open. Closing a port makes the queue inaccessible on that port.

When a port attached to a one-way direct queue is closed, messages in the queue are not affected.

If a requester program closes a port attached to a two-way direct queue and a message or reply to a message on the port has not been received yet, the message or reply is lost. If a server program closes a port attached to a two-way direct queue after the message was received but before the reply was sent, the message is lost.

The `s$call_server` Subroutine for Two-Way Queues

The `s$call_server` subroutine combines the actions of `s$msg_send` and `s$msg_receive_reply`. The `s$call_server` subroutine puts a message in a two-way queue and returns to the caller either a reply or an error code explaining why there is no reply.

If the port used with `s$call_server` is in wait mode, then the subroutine does not return until a server has sent a reply to the message.

If the port has a time limit set on it, `s$call_server` waits up to the time limit. If the time limit expires before a reply is sent, `s$call_server` returns the error code `e$timeout` (`1081`). If the port used for `s$call_server` is in no-wait mode, the subroutine returns with the error code `e$caller_must_wait` (`1277`).

The `s$call_server` subroutine is intended to be used primarily in wait mode. In general, `s$call_server` should not be used on two-way direct queues in no-wait mode or with timeout limits. Use `s$msg_send` and `s$msg_receive_reply` instead. When used on a direct queue, the `s$call_server` subroutine has two opportunities to wait. If a direct queue requester buffer is not immediately available to hold the message, `s$msg_send` waits; `s$msg_receive_reply` always waits for the server to send a reply. If your program receives the `e$timeout` or `e$caller_must_wait` error code, it cannot determine whether the wait is for `s$msg_send` or `s$msg_receive_reply`. The program will not know whether to resend the message, or whether it should simply try to get the reply to the message.

Queue Size

A direct queue is limited to one message per port. The maximum length of the queue's messages is determined when the queue is opened. The `maximum_msg_size` argument of the `s$msg_open_direct` subroutine specifies the maximum message length for a direct queue, in bytes. The argument value must be between 0 and 3,072, inclusive. All ports opening the same queue must specify the same maximum message size.

Direct Queue Scenarios

This section contains scenarios illustrating requester and server operations on direct queues. Assume that the queues are empty at the start of each scenario, that only one requester program and one server program are executing, and that the requester and server ports attached to the queue are in wait mode. With those assumptions, you can think of each scenario as a description of what happens to a single message.

Note that in both direct queue scenarios, the server program's port opens the queue before the requester program's port opens the queue. This order is required for both one-way and two-way direct queues.

The first direct queue scenario illustrates a one-way direct queue with a requester that sends messages and a server that receives messages.

The second direct queue scenario illustrates a two-way direct queue with a requester that uses the `s$call_server` subroutine to send messages and receive replies. The server program receives messages and sends replies.

Scenario 1 - Empty One-Way Direct Queue

Requester Program (wait mode)	Server Program (wait mode)
	1. Call `s$msg_open_direct` (`io_type = 8`). Open the queue for a receive-only server port.
	2. Call `s$msg_receive`. a. Wait for a message.
1. Call `s$msg_open_direct` (`io_type = 7`). Open the queue for a send-only requester port.	
2. Call `s$msg_send`. Add a message to the queue.	
	b. Receive a message.
3. Service the message.	

Scenario 2 - Empty Two-Way Direct Queue Using `s$call_server`

Requester Program (wait mode)	Server Program (wait mode)
	1. Call `s$msg_open_direct` (`io_type = 6`). Open the queue for a server port.
	2. Call `s$msg_receive`. a. Wait for a message.
1. Call `s$msg_open_direct` (`io_type = 5`). Open the queue for a requester port.	
2. Call `s$call_server`. a. Add a message to the queue.	
	b. Receive a message.
b. Wait for a reply.	
	3. Service the message.
	4. Call `s$msg_send_reply`. Remove the message and send a reply in its place.
c. Receive the reply.	

Direct Queues vs. Server Queues

Direct queues and server queues can be used by processes running on different modules connected by StrataLINK communications, or on different systems connected by StrataNET communications. For operations using StrataNET communications, there is little difference in speed between server queues and direct queues. However, when modules are connected by StrataLINK communications, direct queue operations are significantly faster than server queue operations.

Direct queues are particularly fast compared to other queues in multimodule applications that use the StrataLINK communications network because direct queues use a different, faster communication protocol from the other queues. When message and server queues communicate between modules, messages are sent to a StrataLINK link-server process. The link-server process distributes the messages to the appropriate application process. The direct queue protocol bypasses the link-server process. When direct queue messages are passed to another module, the messages are placed immediately in the second module's wired memory.

Direct queues faster than server queues on StrataLINK communications for the following reasons.

- Direct queues store messages in nonpagable memory, whereas server queues store messages in cache. With the cache storage method, if buffer space fills up, the buffer is written to disk, and the messages contained in the buffer must be read from disk later.
- The StrataLINK packet protocol for direct queues moves messages directly from one module's wired memory to another module's wired memory. For server queue communications between modules, messages must wait to be serviced by a link-server process before reaching the application's process.

On the StrataNET communications network, direct queues use the same protocol that server queues and all other network communicators use.

In general, the speed advantage of direct queues makes them preferable to server queues for message passing if the restrictions on direct queues do not affect the application. Direct queues have the following restrictions that do not apply to server queues.

- The maximum message size is 3,072 bytes.
- A single port can have only one outstanding request with direct queues, whereas multiple outstanding requests are possible with server queues.
- All server programs using a direct queue must execute on the module where the direct queue resides.
- Direct queues **cannot** be used with transaction-protected I/O.
- The `s$msg_cancel_receive` subroutine cannot be used to reverse the status of messages from busy to nonbusy in a direct queue.
- The `s$msg_read` subroutine cannot be used to read messages in a direct queue.

Queue-Related Subroutines

The following is a list of the queue-related subroutines.

`s$call_server`
: Used by a requester program to put a message in a two-way queue and to receive a reply from the queue.

`s$control ABORT_OPCODE (3)`
: Used by a requester to deallocate an unused buffer for a direct queue. Applies only when the direct queue is opened using version 2 in the `s$msg_open_direct` subroutine.

`s$msg_cancel_receive`
: Used by a server program to cancel the receipt of a message contained in a server queue. Used by a server or requester program to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Used by a requester or server program to delete a message from a message queue.

`s$msg_open`
: Opens a requester or server program for a one-way or two-way server queue or a message queue.

`s$msg_open_direct`
: Opens a requester or server program for a one-way or two-way direct queue.

`s$msg_read`
: Used by a requester or server program to read a message in a queue.

`s$msg_receive`
: Used by a requester or server program to receive a message from a queue.

`s$msg_receive_reply`
: Used by a requester program to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_rewrite`
: Used by a requester or server program to rewrite a message in a message queue.

`s$msg_send`
: Used by a requester program to put a message into a queue.

`s$msg_send_reply`
: Used by a server program to put a reply into a two-way server or two-way direct queue.

`s$set_max_queue_depth`
: Sets the maximum number of messages for a one-way or two-way server queue.

`s$truncate_queue`
: Used by a server program to truncate an empty message queue.

Programming Considerations

This section contains the following topics.

Using No-Wait Mode with Queues

Using `s$wait_event` and no-wait mode with queue I/O is a logical programming technique. This section describes two important programming concepts concerning the use of no-wait mode on ports attached to queues.

The first concept applies to using a port set to no-wait mode with a queue I/O subroutine that might have to wait. A port is set to no-wait mode using the `s$set_no_wait_mode` subroutine. That subroutine returns an event id. When a queue I/O subroutine has to wait but cannot wait because the port is set to no-wait mode, the queue I/O subroutine returns the error message `e$caller_must_wait` (`1277`). The `s$wait_event` subroutine can then be called using the event id returned from the `s$set_no_wait_mode` subroutine.

> **Note:** The program should not wait on a system event unless it has already attempted to perform the I/O and knows that the wait is necessary. Otherwise, the queue handler will not know that the program is waiting.

Some reasons for queue I/O subroutines to wait are listed below.

- `s$msg_receive` might wait if there are no messages to receive.
- `s$msg_receive_reply` might wait if there is no reply to receive.
- `s$msg_send` to a direct queue might wait if a message sent previously on the same port has not been completely serviced yet.
- `s$msg_send` to a server queue might wait if the maximum queue depth is reached.
- `s$msg_read` might wait if there are no messages to read.

The following pseudocode illustrates the correct way for a requester program to continuously send messages to a port in no-wait mode attached to a direct queue. If a previously sent message is not completely serviced when `s$msg_send` sends a new message, `s$msg_send` returns the error code `e$caller_must_wait` (`1277`). The requester then calls `s$wait_event` to wait on the event ID that was returned by the `s$set_no_wait_mode` subroutine. When the event is notified, the program re-enters the beginning of the loop and calls `s$msg_send`.

```
s$set_no_wait_mode on port1;
repeat
   s$msg_send on port1
   if error = e$caller_must_wait
   then s$wait_event on event id returned by s$set_no_wait_mode
until error not = e$caller_must_wait;
```

The same technique applies to reading and receiving messages on ports set to no-wait mode. Attempt to read or receive messages first. Then, if necessary, call `s$wait_event`.

The second concept concerns what to do after the event is notified when using `s$wait_event` with queue ports set to no-wait mode. This precaution applies to `s$msg_receive`, `s$msg_receive_reply` with `msg_id` set to `-1`, and `s$msg_read`. Server programs operating on a queue port set to no-wait mode should **repeatedly** call these subroutines after the event is notified. Repeated calls are necessary because more than one message (or reply) could arrive between the time the event was notified and the time the server program begins processing. The server should resume waiting on the event only when the `e$caller_must_wait` (`1277`) error code is returned again.

For example, consider a server program that calls `s$msg_receive`, and then calls `s$wait_event` because there are no messages in the queue. When the event is notified, the server calls `s$msg_receive` and receives a message. If the server goes back to waiting after receiving only one message when two had actually been sent, the server is not notified again until the third message is sent. The server would receive the second message and go back to waiting until the fourth message was sent, and so on.

Specifying Message Length

All subroutines that send and receive messages contain the arguments `msg_length_in` and `msg`. The `msg` argument defines the buffer that contains or will contain the message. The `msg_length_in` argument defines the byte length of the message to be sent or received. The value of `msg_length_in` cannot be larger than the data size specified in the declaration of `msg`. Otherwise, the message might overwrite memory or other data structures.

When sending messages to direct queues, remember that the message length is limited to the length specified in the `maximum_msg_size` argument of the `s$msg_open_direct` subroutine. The value of `msg_length_in` (in `s$msg_send`) should be coordinated with the value of `maximum_msg_size`.

When reading and receiving messages, if the message is longer than the value of `msg_length_in`, a nonzero error code is returned. For all queues, the actual size of the message is returned in the `msg_length_out` argument of the `s$msg_read` and `s$msg_receive` subroutines. The error code returned and the correct program response depends on the queue type, as described below.

- For message and two-way server queues, the portion of the message that fits in the buffer is returned, along with the error code `e$long_record` (`1026`). The message is marked busy. Your program should expand the buffer size and then call `s$msg_read` (not `s$msg_receive`) using `this_message` values in the `msg_selector` argument, to read the entire message.

- For one-way server queues and one-way and two-way direct queues, none of the message is returned. The error code `e$buffer_too_small` (`1133`) is returned. The message is still nonbusy. Your program should expand the buffer size and retry the `s$msg_receive` call. If more than one server program is servicing the queue, the message may have been received by another server in the interim.

Refer to Table 5-1 earlier in this chapter for the maximum message lengths possible for each type of queue.

The `s$msg_send` subroutine includes the `msg_length_in` argument, which is both an input and output argument. On input, the argument specifies the length of the message to send. When the `s$msg_send` subroutine returns, the kernel changes the value of this argument to show exactly how much of the message was sent. In most cases, the two values will be the same. However, if an error prevents the kernel from sending the message, the kernel changes the value of this argument to `0`. In this case, the calling routine must re-establish the correct message length value for the variable before repeating the call to the `s$msg_send` subroutine. For example, an error that prevents the kernel from sending the message occurs when the `s$msg_send` subroutine is called in no_wait mode for a queue that already has a number of pending messages equal to the established maximum queue depth. (The `s$set_max_queue_depth` subroutine sets the maximum queue depth for the maximum number of messages of a server or one-way server queue.)

Setting I/O Time Limits on Queues

This section describes how to set a limit on the length of time the operating system waits for I/O to complete on a queue. Timeouts on ports attached to message and server queues are specifically discussed.

With direct queues, only one message exists per port, so this discussion is irrelevant.

The `s$set_io_time_limit` subroutine sets time limits on I/O operations for a specified port. The port can be attached to any file or I/O device. If an I/O operation does not complete before the time limit has elapsed, the subroutine returns the error code `e$timeout` (`1081`) and aborts all I/O on the port that timed out. For more information about the `s$set_io_time_limit` subroutine, see the VOS Subroutines manuals.

When the `s$set_io_time_limit` subroutine is used on ports attached to queues, it must be used with care. As mentioned above, when a timeout occurs, all I/O on the port is aborted. With queues, this abort affects all of the messages sent or received on the timed-out port, not just the last message sent or received on the port. Undesirable results might occur in an application that allows messages to accumulate on a port that has a time limit or when ports are shared among tasks.

When messages are not allowed to accumulate, only the current message is affected by the time out.

The queue-related subroutines that can time out are `s$msg_receive`, `s$msg_read`, `s$msg_receive_reply`, and `s$msg_send`. If one of these subroutines times out on a port attached to a server queue, all messages sent to that queue on the timed-out port are deleted, regardless of the message status.

If an I/O operation times out on a port attached to a message queue, all processing stops on the timed-out port. Messages sent or received on the timed-out port remain in the queue, but any that were previously received are marked with a `been_busy` flag. This is a nonbusy status, which means that the message can be received again, whether or not processing was completed on it before the abort occurred.

Figures 5-2 and 5-3 illustrate the use of `s$set_io_time_limit` in two scenarios that may cause undesirable results.

Figure 5-2 illustrates a requester program using `s$set_io_time_limit` on the port attached to a two-way server queue. The requester program sends a series of messages and then goes back to receive the replies to those messages. If the server program is not executing, the first call to `s$msg_receive_reply` times out, and all messages in the queue are deleted. Two solutions to this problem are presented in Figure 5-2 and described below.

1. By not allowing messages to accumulate, the first solution eliminates the possibility of losing an unknown number of messages from a server queue. This solution uses the `s$call_server` subroutine, which is a combination of the `s$msg_send` and `s$msg_receive_reply` subroutines. The `s$call_server` subroutine does not return until the message is successfully sent **and** the reply is successfully received. If the port times out, only one message is lost, and the program immediately loops back to resend the message.

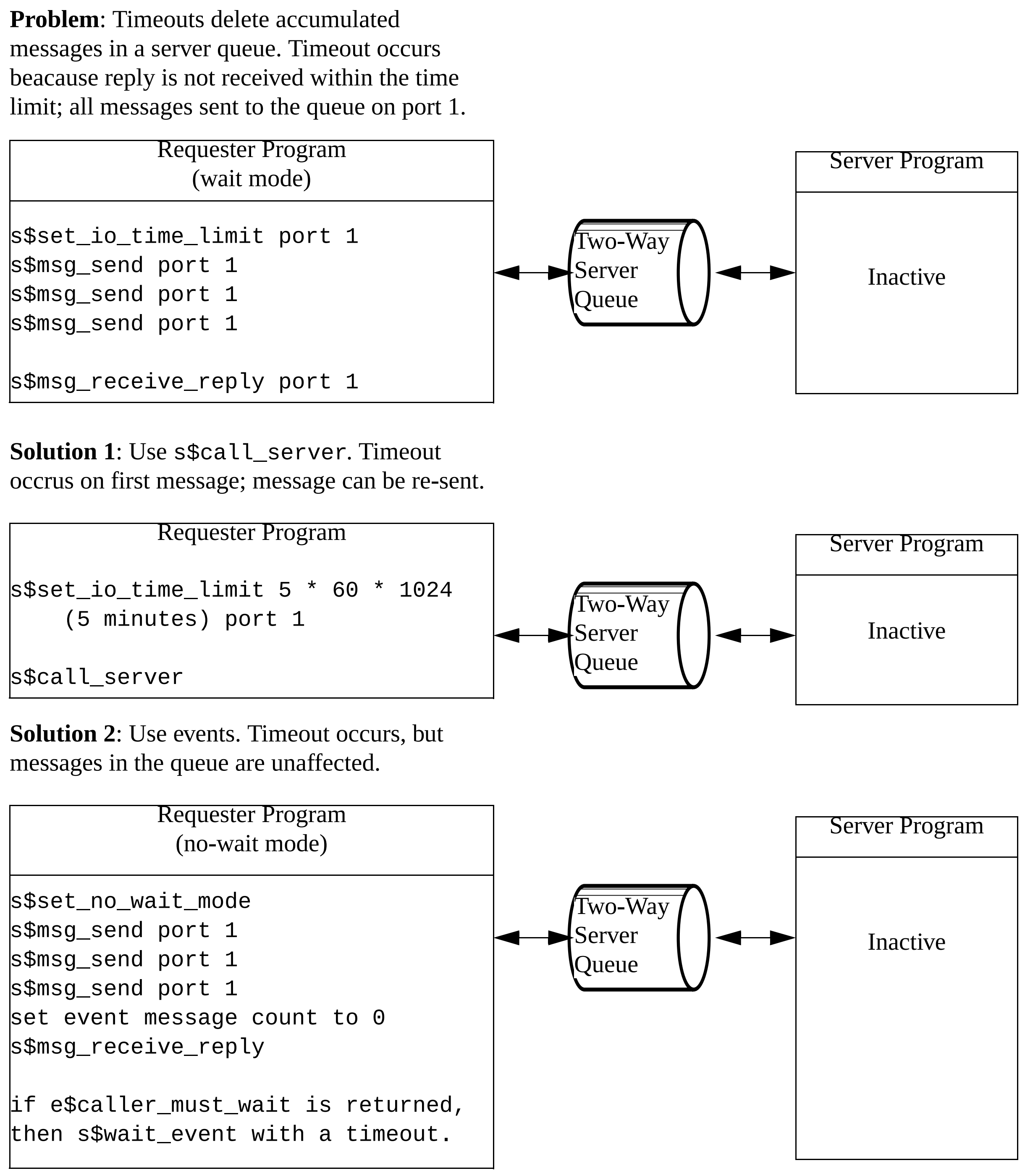
   The timeout specified for this approach must be long enough to accommodate the time needed for the server program to service the message.

2. The second solution uses the event mechanism to handle the timeout. When a call to `s$wait_event` results in a timeout, the contents of queues are not affected. Using the event mechanism, the program safely accumulates messages in a queue and applies a time limit to I/O operations on the queue.

   The port must be set to no-wait mode for this solution.

Figure 5-3 shows a server program using the `s$set_io_time_limit` subroutine to determine when no more messages exist in a message queue. When all messages are received, the `s$msg_receive` subroutine times out. However, because of the timeout, processing aborts on the port. The solution shown in this figure does not allow messages to accumulate.

A timeout on a port attached to a message queue changes `busy` status flags to `been_busy` on all messages sent on the timed-out port. The `been_busy` messages are considered nonbusy, and can be received. In most applications, it is undesirable to receive a message twice or to receive half-serviced messages. The server program should completely service and **delete** a message before receiving another message.



Problem: Timeouts delete accumulated messages in a server queue. Timeout occurs beacause reply is not received within the time limit; all messages sent to the queue on port 1.

```
s$set_io_time_limit port 1
s$msg_send port 1
s$msg_send port 1
s$msg_send port 1

s$msg_receive_reply port 1
```

Solution 1: Use `s$call_server`. Timeout occrus on first message; message can be re-sent.

```
s$set_io_time_limit 5 * 60 * 1024
    (5 minutes) port 1

s$call_server
```

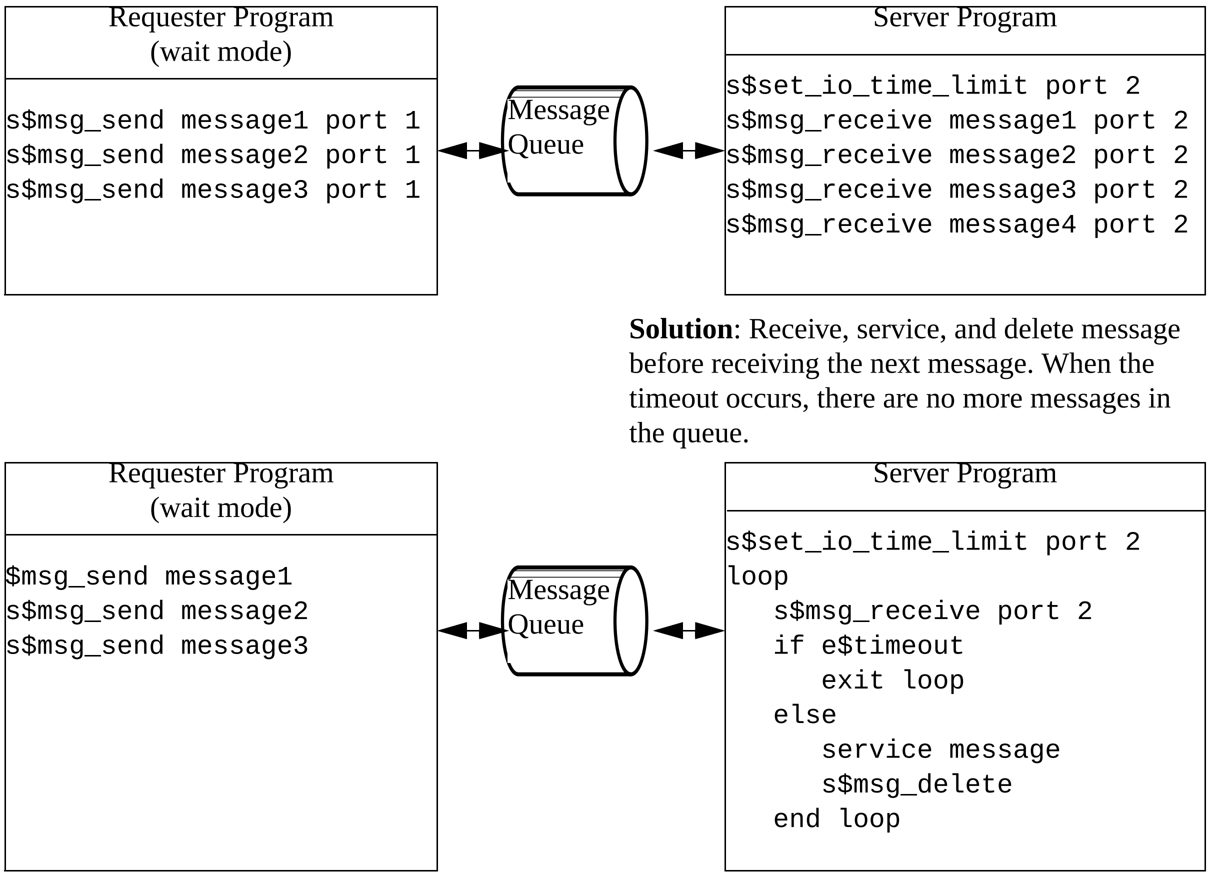
Solution 2: Use events. Timeout occurs, but messages in the queue are unaffected.

```
s$set_no_wait_mode
s$msg_send port 1
s$msg_send port 1
s$msg_send port 1
set event message count to 0
s$msg_receive_reply

if e$caller_must_wait is returned,
then s$wait_event with a timeout.
```

Figure 5-2. I/O Timeouts on a Two-Way Server Queue

Problem: Timeouts change the status of accumulated messages in a message queue. Timeout occurs because there is no message to receive. Messages 1, 2, and 3 are changed to nonbusy.



```
s$msg_send message1 port 1
s$msg_send message2 port 1
s$msg_send message3 port 1
```

```
s$set_io_time_limit port 2
s$msg_receive message1 port 2
s$msg_receive message2 port 2
s$msg_receive message3 port 2
s$msg_receive message4 port 2
```

Solution: Receive, service, and delete message before receiving the next message. When the timeout occurs, there are no more messages in the queue.

```
$msg_send message1
s$msg_send message2
s$msg_send message3
```

```
s$set_io_time_limit port 2
loop
   s$msg_receive port 2
   if e$timeout
      exit loop
   else
      service message
      s$msg_delete
   end loop
```

Figure 5-3. I/O Timeouts on a Message Queue

Accessing Messages Sequentially

To correctly perform sequential access of messages in a queue, you must understand two facts about how messages are ordered in a queue.

- A *`next_message`* value for `msg_selector` causes the current message pointer to move to the next message in its ordering scheme. If new messages are constantly being added, the *`next_message`* value will not necessarily read the next message of highest priority. New high-priority messages may be placed in positions that were already passed by the current message pointer. For example, consider a sequential read loop using the *`next_message`* value for `msg_selector`. The loop has already accessed all messages with priorities of 19, 18, and 17, and is now reading messages with priorities of 16. If a new message is sent to the queue with a priority of 19 (the greatest urgency), that new message is not accessed by a *`next_message`* value of `msg_selector`.

However, another new message with a priority of 15 that is added to the queue is accessed in its turn.

Refer to Figure 5-4 for an illustration of how the *`next_message`* value for `msg_selector` works.

- When the current message pointer reaches the end of its ordered list, an end-of-file error message is returned. When this occurs, all *`next_message`* requests fail. The current message pointer must be reset before any more messages can be accessed using *`next_message`*. The reset is accomplished by using one of the other values of `msg_selector` (*`first_message`*, *`first_message_nonbusy`*, *`this_message`*, or *`this_message_nonbusy`*). The priority values of the new messages are irrelevant; once the pointer is set to the end-of-file value, the *`next_message`* request continues to fail.

A suggested loop for a server program that needs to service the most important messages first is as follows:

```
begin loop
     receive first_message_non_busy
end loop
```

The first nonbusy message is always the next (unserviced) message with the highest priority.

A suggested loop for a program that is sequentially scanning a queue and needs to handle the end-of-file error message described above is as follows:

```
read first_message
do loop while not end-of-file
  read next_message
  if end-of-file
     read first_message
end loop
```

1. Assume that the current state of the file is as follows:

		Message ID	**Message Status**	**Message Priority**
		3	busy	8
current position of file pointer	-->	4	busy	8
		1	nonbusy	4
		2	nonbusy	2
		end of file		

2. `s$msg_read` *`next_message_nonbusy`* obatins message #1.

3. Assume that two messages are aded to the queue before the next read. Now the current state of the queue is as follows:

		Message ID	**Message Status**	**Message Priority**
added	-->	5	nonbusy	19
		3	busy	8
		4	busy	8
current position of file pointer	-->	1	busy	4
added	-->	6	nonbusy	4
		2	nonbusy	2

4. `s$msg_read` *`next_message_nonbusy`* obtains message #6, even though message #5 has a higher priority.

5. `s$msg_read` *`first_message_nonbusy`* obtains message #5.

Figure 5-4. Current Message Pointer Example

Reading and Receiving on Empty Queues

When `s$msg_read` *`first_message`* or `s$msg_read` *`first_message_nonbusy`* is called for an empty queue in wait mode by a server program, the server program waits for a message to become available. When a requester program calls `s$msg_read` *`first_message`* or `s$msg_read` *`first_message_nonbusy`* in wait mode and the queue is empty, the error code `e$end_of_file` (`1025`) is returned. This is true for message and server queues only; direct queues cannot be accessed by `s$msg_read`.

Similarly, when `s$msg_receive` *`first_message_nonbusy`* is called for an empty message or server queue in wait mode by a server program, the server program waits for a message to become available. However, when a requester program calls `s$msg_receive` *`first_message_nonbusy`* for a message queue in wait mode and the queue is empty, the error code `e$end_of_file` (`1025`) is returned.

Refer to the VOS Transaction Processing Facility Reference manuals for detailed information on the results of calling these subroutines for an empty queue in no-wait mode and when an I/O time limit is set.

Link Errors on Direct Queues

When a requester program for a direct queue resides on a remote module, StrataLINK outages might detach the requester's port connections to the queue. Remote port connections are detached whenever retransmissions exceed the retry threshold. This situation might be caused when a temporary power failure occurs on the remote module, a nonduplexed StrataLINK ring experiences a transient failure, a link breaks, the module is removed from service, or the server process stops.

When the remote requester port connection is detached because of a StrataLINK outage, the requester gets the `e$drq_connection_broken` (`3972`) error code if it attempts to abort I/O on the port or use the port to send a message or receive a reply. A remote requester program attached to a direct queue should test for the `e$drq_connection_broken` (`3972`) error message after every call to `s$msg_send`, `s$msg_receive_reply`, and `s$control ABORT_OPCODE (3)`.

To try to regain access to the queue in this circumstance, the application should close and detach the port to the direct queue, and then reattach the port and reopen the direct queue. This action creates a new connection if a new connection is possible.

The application should only attempt reconnection a limited number of times, or at some timed interval. If the initial problem is not recoverable, the direct queue might remain inaccessible. Closing and reopening the queue might not be successful if the initial problem persists. The `s$wait_for_module` subroutine might be useful in this condition.

Pooled Buffer Memory Scheme for Direct Queues

The direct queue is a powerful communication technique, especially for multimodule requester/server applications that do not require transaction protection. Direct queues are fast, but they also consume wired memory space. Knowledge of how direct queue messages are stored might help you evaluate whether direct queues or server queues would better serve your application's needs.

Two different schemes are currently used to store direct queue messages. The `version` argument of the `s$msg_open_direct` subroutine determines which scheme is used for a queue. All ports opening a direct queue must specify the same value for `version`. When `version` is set to `1`, a separate buffer exists in main memory for each requester port attached to the queue. When `version` is set to `2`, direct queue messages are stored in pools of buffers in main memory. All requesters on a module attached to the same direct queue share the same buffer pool. This section describes the pooled buffer scheme.

Buffer Pool Allocation

This scheme allows requester ports to share buffer space, which is more efficient than reserving a buffer for each requester. The programmer or the user can specify the ratio of buffers to requester ports that should exist. If the ratio specified is 100 percent, then a buffer is allocated for each requester on that queue.

The location of the requester and server programs affects buffer pool management. Two configurations are possible.

- Local requesters. Requester and server programs reside on the same module. In this case, all requesters share a pool of buffers and all servers share a pool of buffers. The number of buffers in the requester pool is based on the `buf_pct` parameter in the `s$msg_open_direct` subroutine. When a message exchange occurs between a requester and a server, the buffers assigned to the requester and server are swapped.

- Remote requesters. Requester and server programs reside on different modules. In this case, there is still a pool of requester and a pool of server buffers on the local (server) module, as described above. For two-way queues only, an additional pool exists on the remote requester's module. The number of buffers in this pool equals the number of buffers in the local requester pool. For one-way queues, the remote requester module sends messages directly to the server module using link buffers. Therefore, there is no need for a remote requester buffer pool.

For two-way queues, a separate buffer pool exists on each module for each direct queue/module combination to which ports have been attached. Figure 5-5 illustrates this concept. Module B (the server module) has two buffer pools allocated on it — one for messages coming from Module A and one for messages coming from Module C. Modules A and C each have one buffer pool allocated, since they each have attached ports to only one direct queue on one module.

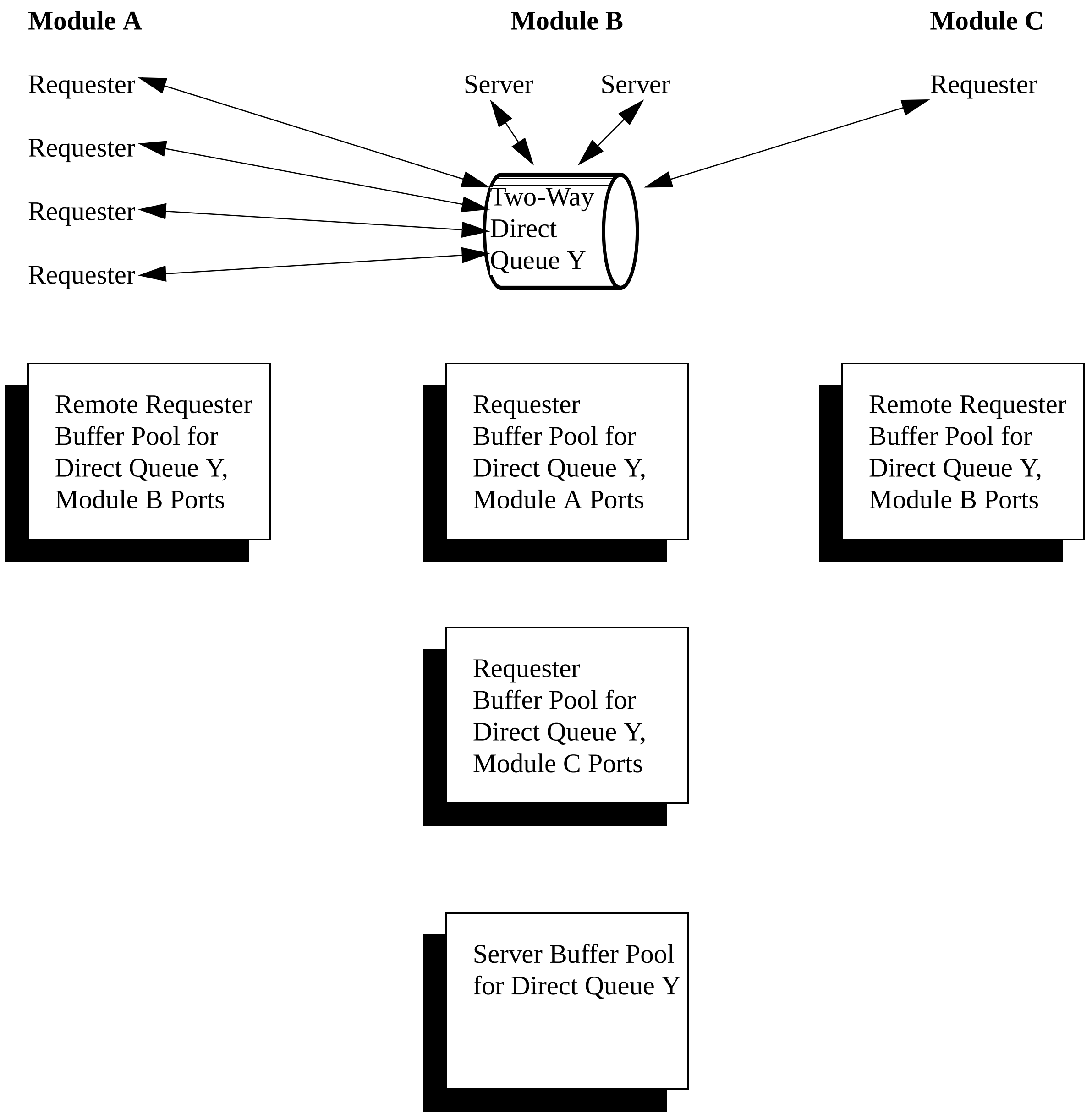

Module A
Module B
Module C
Requester
Requester
Requester
Requester
Server
Server
Requester
Two-Way Direct Queue Y
Remote Requester Buffer Pool for Direct Queue Y, Module B Ports
Requester Buffer Pool for Direct Queue Y, Module A Ports
Remote Requester Buffer Pool for Direct Queue Y, Module B Ports
Requester Buffer Pool for Direct Queue Y, Module C Ports
Server Buffer Pool for Direct Queue Y

Figure 5-5. Direct Queue Buffer Pool Allocations

The `buf_pct` Argument

The number of buffers in each pool is controlled by the `buf_pct` argument of the `s$msg_open_direct` subroutine. This argument specifies the number of buffers based on a percentage of requesters opened on the queue.

The argument is specified as an integer from 1 to 100, inclusive. A `buf_pct` of 100 means that the number of buffers in the pool approximately equals the number of requesters on the module opened on the queue. A `buf_pct` of 50 means that the number of buffers in the pool is approximately half of the number of requesters on the module opened on the queue. (The actual number of buffers is the specified percent of requesters, plus the number of servers opened on the queue.)

> **Note:** In `s$msg_open_direct`, the `buf_pct` argument is valid only when the `version` argument is set to 2. If `version` is set to 1, `buf_pct` defaults to 100.

The module on which the server program resides determines how many buffers to allocate in each of its pools, based on the `buf_pct` argument. If there are multiple servers for the same buffer pool, and those servers specify different `buf_pct` values, then the value specified by the last server invoked is the one used. This means that the `buf_pct` value can be changed without stopping an entire application. The new value does not cause buffers to be immediately added or freed. It is used to determine whether to add or free buffers if requesters add or detach ports.

The `buf_pct` argument is ignored in requester programs.

Buffer Pool Use

The server module communicates with the requester module, sending the requester module the number of buffers to establish in its buffer pools. As server and requester programs are added or terminated, the operating system adds and deletes buffers in the appropriate buffer pools. The operating system attempts to maintain an equal number of buffers in the matching pools on the requester and server modules.

When a requester sends a message to a direct queue (using `s$msg_send`), the message is stored in one of the buffers in the requester module's buffer pool. If `buf_pct` is less than 100 percent, a buffer might not be immediately available. If that is the case, then the requester is placed in priority order in the buffer wait queue to wait for an available buffer. When a buffer becomes available, it is assigned to the first requester in the buffer wait queue.

Once assigned, a buffer remains assigned to the requester until one of the following situations occurs.

- The requester uses the buffer by sending a message.
- The requester calls the `s$control` subroutine with `ABORT_OPCODE (3)`.
- The requester detaches the port from the queue.

Since buffers on the requester and server modules correspond on a one-to-one basis, a buffer is always available on the server module once the requester is assigned to a buffer.

A message transmitted to the server module is put in one of the buffers in a queue of buffers. When a server program is available, it is assigned to the first buffer in the buffer queue.

Note that a server program is assigned to a message when it successfully receives the message, not when it finishes servicing a previous message. This distinction becomes important when the user wants to terminate a server program. A server program can finish processing its current message and then stop without stranding any messages.

Requester Program Requirements

When a server program sets a queue's `buf_pct` value to less than 100 percent, the requester programs for that queue must be prepared to wait when sending a message. This is because a buffer may not be immediately available to hold the requester's message. (When `buf_pct` is 100 percent, a buffer is always available.)

Requester ports in wait mode are prepared to wait; extra considerations are not necessary. Requesters can send messages using `s$msg_send` or, for two-way queues, using `s$call_server`.

Requester ports in no-wait mode should send messages using `s$msg_send`, followed by a check for the error code `e$caller_must_wait` (`1277`). If `e$caller_must_wait` is returned, the requester should call `s$wait_event` and reissue the `s$msg_send` call.

Deallocating an Assigned Buffer

Buffers are freed when the message in the buffer is processed. However, there may be times when a requester waits for a buffer, gets the buffer, and then decides it does not need the buffer (that is, it does not need to send a message to the buffer). In this situation, it is important for the requester to deallocate the unused buffer. A requester can eventually tie up all of its buffers if it accumulates unused buffers without deallocating them.

A requester deallocates a buffer by calling the `s$control` subroutine with `ABORT_OPCODE (3)`. The `control_info` argument must be set to 0.

> **Note:** In some situations, the operating system might detect that a buffer is not being used and return that buffer to the pool. However, you should not depend on the operating system for this action. The application should deallocate a buffer that it does not need.

Changing the Buffer Percent Outside the Application

The `analyze_system` command includes a request that allows you to change the value of the `buf_pct` argument for all direct queues on a particular module. This value overrides values specified in the application program. The request name is `set_dq_defaults`.

> **Note:** If you use this method to control the number of buffers associated with a two-way queue, all requester processes running on the module using no-wait mode must be prepared to wait after sending a message. See the ‘‘Requester Program Requirements” section above.

Evaluating the Buffer Percent

The buffer percent allocation controls how many buffers exist in each buffer pool for each queue. Setting the number too small causes the servers to wait for requesters, or requesters to wait for buffers. Setting the number too large causes no problems except wasting wired memory space.

If the number of requesters is small, you can let the percentage default to 100. However, if the number of requesters is large, a percentage less than 100 would maximize memory space and processing efficiency. For example, if one server serves 50 requesters, there is no reason to reserve 50 requester buffers — processing would wait for the server anyway.

You can evaluate the processing efficiency of different buffer percent values by measuring the number of times requesters and servers receive the `e$caller_must_wait` (`1277`) error code in response to queue I/O subroutine calls. The optimum performance is a balance between the lowest number of `e$caller_must_wait` errors and the lowest buffer percent value. You might choose values that are less than the optimum values if the application has many requesters and memory usage is a concern.

Sample Application

Appendix B, ‘‘Sample Application,” contains annotated program code for a sample application called the Automated Auction System. The annotations in the code are enclosed in asterisks and are identified with reference numbers. The reference numbers for the comments illustrating queue features start with “Q.”

The Automated Auction System uses queues to support communication between a server program and requester (user) tasks. User requests are transmitted to the server program using a two-way server queue. Both requester and server programs attach ports to the queue and open the queue. The requester tasks send messages to the queue; the server program receives the messages, services them, and sends back replies. The requester tasks then receive the replies.

The server program creates the queue, and, in the program epilogue handler, deletes the queue.

Chapter 6: TP Restore

This chapter contains the following sections.

Features of TP Restore

An online transaction processing application must be able to recover its data files when unforeseen events destroy the files. The `tp_restore` command provides this capability. The command re-creates a corrupted transaction data file by using a backup version of the data file as the starting point, and then reapplying all transactions committed since the backup was made. Transactions are reapplied in the same order in which they were originally committed.

The `tp_restore` command is typically used with the `restore` command. Use the `restore` command first to install a backup version of the corrupted transaction data file, and then use the `tp_restore` command to reapply transactions to the backup version. To reapply transactions to an earlier version of a transaction data file is called *rolling forward*.

A single execution of the `tp_restore` command can roll forward multiple data files or a single data file. For example, an application that updates five transaction data files could have all five of its files corrupted, or only one of them corrupted. The `tp_restore` command can roll forward any or all of the five files. Similarly, if several applications using transaction protection run on the same module, the TPOverseer writes transactions from all the applications to the same log files. A single execution of the `tp_restore` command can roll forward files from one application or all of the applications, depending on the file names you provided in the command.

The `tp_restore` command can roll forward only VOS data files that are declared transaction files. The command is privileged. It may be run interactively or submitted as a batch request.

File Management

An appropriate set of transaction log files and data-file backups is required for `tp_restore` to perform successfully. If you intend to use the `tp_restore` command, you **must** ensure that the appropriate files are available when you need them. This section discusses various file management issues.

This section includes the following topics.

- ‘‘Backups”
- ‘‘Data-File Backups”
- ‘‘Warnings about File Status”
- ‘‘Log File Keep-and-Delete Schedule”
- ‘‘Log-File Backups”
- ‘‘Disaster Recovery Considerations”

Backups

You can back up log and data files using either of the following VOS commands.

- `save`. This command backs up multiple files, directories, and links onto tape or disk. It is described in the *VOS System Administration: Disk and Tape Administration (R284)*.

- `save_object`. This command backs up a single file, directory, or link onto tape or disk. It is described in the *VOS Commands Reference Manual (R098)*.

Use the following commands to install a backup version.

- `restore`. This command copies multiple files, directories, and links from tape or disk. It is described in the *VOS System Administration: Disk and Tape Administration (R284)*.

- `restore_object`. This command copies a single file, directory, or link from tape or disk. It is described in the *VOS Commands Reference Manual (R098)*.

The application may be running while backups are made. If that is the case, back up the data files first, and then the log files. Backing up the files in this order ensures that new updates are included in the log files, even if the application updates portions of data files after those portions are backed up. The `tp_restore` command can determine which transactions already exist in the data-file backup and which transaction it should roll forward first.

Data-File Backups

You must enforce an appropriate backup schedule for all transaction data files. These backups are the starting point for `tp_restore`. If a set of transaction data files contains integrated or related information, the backups of those files should be synchronized.

The number of log files created by the application should influence the frequency of your transaction data-file backups. A high-volume transaction application creates many log files. The greater the time span between transaction data-file backups, the more log files you must save to ensure a complete roll forward. The more log files there are, the longer the roll forward takes.

Warnings about File Status

The following are important points about changing a file from a transaction file to a nontransaction file.

- When you create a backup of a transaction file that you expect to use with `tp_restore`, the file **must** be a transaction file when you back it up.
- When you back up a transaction file, the backup version that you create is also a transaction file. **Never** change the backup version of a transaction file to a nontransaction file.
- When you use the `set_transaction_file` command or the `s$set_transaction_file` subroutine to change a transaction file to a nontransaction file, you erase transaction-related information in the file's file entry that is required during the roll-forward procedure. (The file entry is a data structure associated with every VOS file.) This information cannot be recovered. You cannot change the file back to a transaction file and expect that information to reappear.
- If you must change a transaction file to a nontransaction file, perform your actions in the order described below. This sequence ensures that future roll forwards are possible, since the backup is made when the file is a transaction file. It also ensures that changes made while the file was not a transaction file are preserved and are not lost when a roll forward is executed. Backups should be performed whenever a transaction file is changed to a nontransaction file and changes are made.

  1. Set transaction file off.
  2. Perform unrelated actions (that are not transaction protected).
  3. Set transaction file on.
  4. Back up the transaction file.

Log File Keep-and-Delete Schedule

The `tp_restore` command starts with a transaction data-file backup and reconstructs all transactions performed since the backup, using the information in transaction log files. The log files contain the transaction information to be reapplied to the data files. Log files must be kept long enough to enable a complete roll forward using the last backup of the data files.

For each transaction file involved, `tp_restore` usually needs to read the log file that contains the last successfully committed transaction in the backup version being used. This requirement ensures that the roll forward starts at the proper place. It ensures that `tp_restore` does not skip any transactions even when you make transaction data-file backups during online processing. Because of this overlap requirement, you may have to keep log files that were created before the data-file backup date.

The log-file keep schedule should match the data-file backup schedule, with some overlap included. Delete old log files according to the following schedule.

1. Back up all of your application's VOS data files.
2. Examine the last modified date on each of the backup files. Notice which file was **least** recently modified.
3. Subtract one day from the least recent modification date. Keep all logs created on and after that date.

Log-File Backups

You may want to back up the transaction log files, especially if your data-file backups are infrequent. For example, if you back up your data files weekly, you should probably back up the log files daily.

The amount of disk space you can afford to give to log files should determine the frequency of log-file backups. After you back up log files, you can delete all but the current log from the disk.

When backups are made while the application continues to support online operations, then a backup of the current log file is immediately out of date. If possible, you should use the online version, not a backup version, of the current transaction log for `tp_restore`. Similarly, if you have multiple backup copies of the same log file, always use the latest version for `tp_restore`.

See ‘‘Transaction Log-File Names’’ in Chapter 4 for a description of transaction log-file names and where the logs reside.

Disaster Recovery Considerations

Disaster recovery precautions for an application using transaction protection should consist of frequent data-file backups and even more frequent transaction log-file backups. For disaster recovery purposes, these backups should be stored offsite.

All transaction logs not stored offsite, including the current log, should be on the list of files to save during emergency exit procedures.

Performing `tp_restore`

For `tp_restore` to execute successfully, you must supply it with the following information:

- the path names of the transaction files when the transactions were originally committed, and where the backup versions of those files reside when the `tp_restore` command is issued
- the path name of the directory where the transaction log files reside when the `tp_restore` command is issued

This section describes how you should prepare for `tp_restore`. Preparation consists of the following four steps:

- installing the backup versions of the transaction data files
- assembling the original path names of the transaction data files to be rolled forward
- assembling the transaction log files into a single directory
- ensuring that the correct system-module ID will be inferred for the transaction data files

This section includes the following topics.

Installing Backup Versions of Transaction Data Files

The roll forward is performed on backup versions of the transaction data files. This step involves the normal restore process that takes a previously made backup version of a file from tape or disk and writes it to the desired disk location. You can choose to put the backup versions in either of the following:

- in their original directories (replacing the corrupted files)
- in a new directory, created specifically for this purpose. Specify this directory name in the `-destination` argument of the `tp_restore` command. The object names given to the backup versions of the transaction data files must match the object names of the original transaction data files.

If you replace the original transaction files with the backup versions, `tp_restore` performs the roll forward in the files’ original directories, on their original modules. This action is called performing the roll forward *in place*.

When a roll forward is performed in place on files that reside on different modules, `tp_restore` must make remote calls. In this situation, it is more efficient to move all of the files to a destination directory on the module where the `tp_restore` will be executed. The recovered files can be copied to their proper locations when the roll forward is complete.

When you perform the roll forward in place, the path names of the backup versions are the same as the path names of the original files. If you place all of the backup versions of the transaction data files in a directory specified by the `-destination` argument, the path names of the backup versions are not the same as the path names of the original transaction files. The `tp_restore` command makes the connection between the original transaction file and the backup of that file with the matching *object name*. (The object name is the last component in a full path name.) The object name of the backup version must match the object name of the original transaction file.

Assembling Transaction Data-File Path Names

You must provide the `tp_restore` command with the path names of all of the transaction data files that you want rolled forward. The path names must be the path names of the transaction data files when the transactions were originally committed.

If data-file path names were changed, the path names that you provide to the `tp_restore` command must match the path names used in the log files. During the roll forward, `tp_restore` reads the data-file path names that you provide, and then looks for references to those data-file path names in the log files. If the data-file path names have been changed since the log file was created, you must use the old data-file path names. If you are performing the roll forward in place, you also have to rename the actual file to its old name. After the TP restore, you can rename the file again to its new name.

If the data-file path names were changed in the middle of the period being rolled forward so that the log files reference the same data file using different path names, you must issue the `tp_restore` command several times — once for each time the file path names were changed.

If a data file is referenced in the log files but is not included in the list of data-file path names to be rolled forward, `tp_restore` ignores that file. This allows you to roll forward a single file in a set of data files, if only one of the files is corrupted.

If a data file is mistakenly omitted from the list of files to be rolled forward, you can reissue the `tp_restore` command using only the omitted data file (and the same log-file directory used the first time).

Supply the transaction data-file path names in one of the following arguments of the `tp_restore` command. The arguments are mutually exclusive.

- `path_names` argument. This argument specifies the original transaction data-file path names to be rolled forward. Use this argument if the list of files is small. Star names, relative path names, and link names are valid.

- `-control` argument. This argument specifies the name of a file that contains the list of transaction data-file path names to be rolled forward. The path names in the control file must be full path names. No star names, relative path names, or link names are allowed.

  Use a control file if (1) the list of files is too long to fit in the `path_names` argument, (2) the module or system on which the files resided when the transactions were originally committed was taken out of service or renamed, or (3) the system performing the roll forward does not include the module on which the transactions were originally committed. The last two items refer to situations where the correct module ID cannot be inferred from system configuration tables. The control file allows you to explicitly specify the correct module ID for those situations.

Assembling Log Files in a Single Directory

All transaction log files required for the roll forward must be moved to a single directory on the module where the `tp_restore` command will be issued. Specify the directory name in the `-restore_dir` argument of the `tp_restore` command. The directory specified in `-restore_dir` can be a new directory created specifically for this purpose, or it can be the same directory specified in the `-destination` argument of the `tp_restore` command (if you are using the `-destination` argument). When you move log files to the new directory, the object name of the moved log file must match the object name of the original transaction log.

The log files that you assemble for the roll forward are usually a combination of old log files and current log files. The current log file for a module resides in the `system>transaction_logs` directory. Some required log files may reside on different modules in your system. You must determine which logs to include, based on the following:

- the modules where the transaction files resided when the transactions were originally committed
- the time span covered by the roll forward

Modules to Consider

The updates that apply to a specific transaction data file are recorded in the logs from the module where the transaction file resided when the transactions were originally committed. The `tp_restore` command must be able to match the module ID associated with a transaction data file with the module IDs of the log files. For example, if the transaction data files `file1` and `file2` resided on module `#m2` during application run time, `tp_restore` would need logs from module `#m2` to roll forward `file1` and `file2`. If `file3` resided on `#m1`, `tp_restore` would need log files from `#m1` to roll forward `file3`.

The `tp_restore` command knows the module associated with a log file from the log file's object name, which includes a system-module ID. For data files, `tp_restore` can usually infer the module ID from the data file's original path name. The path name includes a system name and disk name that are mapped to system configuration tables to obtain a module ID.

Remember that you are concerned with module IDs at the time of the original commit of the transaction. The situation gets more complicated if modules were renamed or taken out of service between the time of the original commits and the roll forward, or if the roll forward is being performed on a system that is not linked or networked to the system on which transactions were originally committed. See the ‘‘The Control File and System-Module IDs” section later in this chapter for an explanation of how to specify correct system-module IDs to resolve these complications.

Time Span Covered

You need to include all log files covering the time span being rolled forward. All log-file object names include the date that they were created.

For the purposes of `tp_restore`, the **end** of the roll-forward time span for a single transaction data file is usually the current date and time. Therefore, you usually need to include logs up to and including the current online log for each module where a transaction file being rolled forward resided. However, the `-to_state` argument of the `tp_restore` command allows you to specify a stop time earlier than the current date and time. An early stop time means that only transactions in progress or already committed at the stop date and time specified are reapplied during the roll forward. If an early stop time is specified, then you only need to include the log files dated through the stop date.

When the roll forward involves a set of files from multiple modules, `tp_restore` attempts to keep all files synchronized with regard to the roll-forward stop time. Therefore, `tp_restore` might continue past the specified stop time to ensure consistency of data files.

Logically, the beginning of the roll-forward time span is the day the backup of the data file was made. However, because of the way `tp_restore` works, you must sometimes include logs dated earlier than the backup date.

> **Note:** To be safe, start the roll forward with the logs from the day before the date of the last committed transaction contained in the data file's backup.

As an example, consider the following sequence of events.

1. Data-file status set to `transaction_file`
2. Data-file backup made
3. Data file updated by transactions
4. Data-file backup made
5. Data file updated by transactions
6. Data file corrupted
7. `tp_restore` required, using the backup version made in step 4

The backup version's file entry includes information that enables `tp_restore` to identify the last successful transaction contained in the backup version. The `tp_restore` command searches the log files for that transaction and begins the roll forward immediately after. The transaction file is not rolled forward if `tp_restore` cannot find that last successful transaction. This requirement enables `tp_restore` to use backups that were made during online processing and still ensure that no transactions are skipped.

Since it is unlikely that you can identify the specific log containing the last successful transaction, the following procedure is recommended.

1. Find the date of the last transaction applied to the backup version of the transaction data file. (Look at the `last modified at` field of the `display_file_status` command.)

2. Subtract one day from the date.

3. Include the log files that have a creation date equal to or greater than that date. The creation date of a log file is included in its path name.

Figure 6-1 illustrates the concept of identifying which log files are required to roll forward a set of data files.

The Situation

An application maintains four transaction data files, named A, B, C, and D. A complete set of backups are made on the 15th of the month. On the 18th, the files are corrupted. A roll forward is necessary.

File	Date Last Modified	File	More Modifications
A	10th	A	17th
B	12th	B	17th
C	15th	C	18th
D	15th	D	18th
Backups made on the 15th		Files corrupted on the 18th	

To Roll Forward All Files

1. Start with the backup version of the files made on the 15th.
2. The file with the least recent modify date is File A. Go back one more day, to the 9th.
3. Issue the `tp_restore` command, using all four data files and all transaction log files dated the 9th and later.

To Roll Forward One File

1. Suppose that only File C was corrupted. Start with the backup version of File C only. Leave the other files as they existed on the 18th.
2. The last modify date for the backup version of File C is the 15th. Go back one day, to the 14th.
3. Issue the `tp_restore` command, using only File C and all transaction log files dated the 14th and later.

Figure 6-1. Log Files Required for a Roll Forward

There are times when there is no previous transaction for `tp_restore` to identify. In the following situations, the beginning of the roll-forward time span is the day the transaction-file backup was made.

- The backup version of the transaction data file was made immediately after the file was created and set to be a transaction file.
- The backup version of the transaction data file was made immediately after the file was changed to a transaction file. Remember that when a file is changed to a nontransaction file, transaction-related information in the file's file entry is erased. Therefore, changing a file from a transaction file to a nontransaction file and then back to a transaction file is equivalent to declaring the file a transaction file for the first time.

In either of these situations, there is no previous transaction for `tp_restore` to identify. You do not have to provide logs dated earlier than the backup date. The `tp_restore` command recognizes the first transaction that was committed after the file was changed from a nontransaction file to a transaction file. You must provide the log that contains this first transaction. If in doubt, use the log whose date matches the date that the backup was made. If the log containing the first committed transaction is not provided to `tp_restore`, the transaction file is not rolled forward.

As an example, consider the following sequence of events.

1. Data-file status is set to `transaction_file`.
2. Transaction data-file backup is made.
3. Transaction data file is updated by transactions.
4. Transaction data file is corrupted.
5. `tp_restore` is required, using the backup version made in step 2. Since the file was declared a transaction file immediately before the backup was made, the logs do not have to go back further than the backup date. `tp_restore` recognizes that the file was new and does not look for a previous transaction.

The Control File and System-Module IDs

The `-control` argument of the `tp_restore` command specifies the name of a file that contains a list of transaction data-file path names to be rolled forward. For example, the `-control` argument might be specified as follows:

```
-control application_filenames
```

The file named `application_filenames` must contain a list of transaction data-file path names that are to be rolled forward. The path names listed are the path names of the transaction files when the transactions were originally committed.

For example, a control file might contain the following path names:

```
%prod#dept1>application>data_files>file1
%prod#dept1>application>data_files>file2
%prod#dept2>application>data_files>file3
```

The control file must also provide, implicitly or explicitly, the system-module ID identifying where these files resided when the transactions were originally committed. The system-module ID is required so that the appropriate log files can be matched to the transaction data files for the roll forward. All log-file object names are in the format:

```
>tlf.s-m.number.date
```

where *s-m* is the system-module ID on which the log file was created.

Usually, `tp_restore` can use the system and disk names in the transaction data-file path name as an entry to system configuration tables, and infer the corresponding system-module ID. In the control-file example shown above, suppose that the system-module IDs inferred for the example files are, respectively, 1-1, 1-1, and 1-2. To roll forward `file1` and `file2`, `tp_restore` would use log files that include the system-module ID 1-1 in their object name. To roll forward `file3`, `tp_restore` would use log files that include the system-module ID 1-2 in their object name.

A complication exists when module or system names or IDs are changed after the transaction data-file backups were made. As a result, the system configuration table leads `tp_restore` to the wrong IDs, or the tables no longer have entries for the old names. For example, if the configuration of our sample `prod` system were changed so that the disk name `dept2` refers to system-module ID 1-4, then the system-module ID inferred for `file2` would be 1-4, instead of 1-2. As a result, `tp_restore` would incorrectly search for log files with 1-4 in their names, instead of 1-2.

As another example, suppose that, after the transaction data-file backups were made, disk names were changed from `dept`*x* to `off`*x* . In this case, even though module IDs were not changed, `tp_restore` would not find any entries in the configuration tables for disk names `dept1` or `dept2` and could not infer the module ID. The files would not be rolled forward.

In situations like the ones described above, you can explicitly provide the correct system and disk names in the control file, after the file path name. The `tp_restore` command uses those names when it looks up IDs in the configuration tables. For example, to correctly set up a control file for our example where the disk names were changed from `dept`*x* to `off`*x*, provide the original transaction data-file name, followed by the system and disk names that lead to the correct system-module ID, as follows:

```
%prod#dept1>application>data_files>file1 %prod#off1
%prod#dept1>application>data_files>file2 %prod#off1
%prod#dept2>application>data_files>file3 %prod#off2
```

The `tp_restore` command uses the `off`*x* names instead of the `dept`*x* names to infer the system and module IDs from the system configuration tables. In our example, `tp_restore` infers 1-1, 1-1, and 1-2. These IDs result in the correct log files for each transaction data file.

Another set of complications is possible if the correct system and module IDs do not exist in the system configuration tables, and therefore cannot be inferred from any name. This situation might exist when a different system is performing the roll forward or when a module is taken out of service. For example, suppose that you want to perform the roll forward on the `dev` system, which is not linked or networked to the `prod` system. The `dev` system configuration tables do not include entries for the `prod` system modules.

In this situation (and, optionally, in all of the situations described previously), you can explicitly provide the system-module ID, allowing `tp_restore` to bypass the system configuration tables. For example, the following control file would allow the `dev` system to perform the roll forward using transaction data files and log files that were created on the `prod` system:

```
%prod#dept1>application>data_files>file1 1-1
%prod#dept1>application>data_files>file2 1-1
%prod#dept2>application>data_files>file3 1-2
```

In summary, if `tp_restore` is not able to infer the correct system-module ID from the transaction data-file path name, you must provide it with one of the following:

- a system name/module name that leads to the correct ID through the system configuration tables
- the actual system-module ID

Restoring Large Transactions

When you have large transactions to restore, the `-gather_txns` argument provides the flexibility necessary to handle them. In most cases, you can allow the `-gather_txns` argument to use its default value, which is 50. For processing efficiency, the `tp_restore` command gathers information about multiple transactions and stores this information in the user heap area of memory. When it accumulates the number of transactions specified in the `-gather_txns` argument, the `tp_restore` command applies those transactions to the appropriate files and clears the user heap before starting to gather more transactions.

Depending on the number of transaction files involved in the transactions, as well as the number of modules, the number of log files involved, and the available resources in the user heap, `tp_restore` might run out of user heap space before it completes its information-gathering step. In that case, reconstruct the destination directory and then reissue the `tp_restore` command using a lower value in the `-gather_txns` argument.

Serial Executions of `tp_restore`

When you need to start the `tp_restore` session at the point where a previous `tp_retore` session stopped, use the `-continuation` argument. The `-continuation` argument provides support for serial executions of the `tp_restore` command. Whenever the `tp_restore` command executes, it saves information about its stopping point in a file named `tp_restore_continuation_info` in the restore directory. The command also adds a message to its `.out` file naming the starting log for the next `tp_restore` session. To perform serial `tp_restore` sessions, use the log file identified in the previous session's `.out` file as the starting log for the next session, and specify the `-continuation` argument in the next `tp_restore` command. The `-continuation` argument causes the tp_restore command to use the `tp_restore_continuation_info` file to determine the starting point for each of the data files in the destination directory.

For serial `tp_restore` sessions, you must use log files with names whose numbers sequentially follow the logs used in the previous `tp_restore` session, with some overlap. Since the overlap requirement varies, always start with the log file named in the previous session's `.out` file. For example, assume that the log files used in a `tp_restore` session were named as follows:

```
tlf.128-1.0000000003.98-01-27
tlf.128-1.0000000004.98-01-27
tlf.128-1.0000000005.98-01-27
```

Now assume that the `tp_restore .out` file included the following message:

```
To restore files from module 1 in the next tp_restore
-continuation session, you should start with log
tlf.128-1.0000000003.98-01-27
```

In that case, the next `tp_restore` session must include the first three files in the following list, and could have any number of subsequently numbered log files as well. The next `tp_restore` session might use the following log files.

```
tlf.128-1.0000000003.98-01-27
tlf.128-1.0000000004.98-01-27
tlf.128-1.0000000005.98-01-27
tlf.128-1.0000000006.98-01-27
tlf.128-1.0000000007.98-01-27
```

You should use the same restore directory for all executions of the `tp_restore` command in the series. Otherwise, you risk accessing incorrect continuation information.

If you specify the `-continuation` argument with a restore directory that does not contain a `tp_restore_continuation_info` file, `tp_restore` executes as if the `-continuation` argument were not specified. If you are in the middle of a series of `tp_restore` operations, you **do not** want this to happen. To avoid this problem, create a VOS command macro that executes the `tp_restore` command with the `-continuation` argument. The macro could check for the existence of the `tp_restore_continuation_info` file and if the file is not found, exit before executing the `tp_restore` command.

You can use the `-continuation` argument even if some of the transaction data files in the destination directory were not included in the previous `tp_restore` session. However, you must research the correct starting log file for the newly introduced transaction file and include the required log files in the restore directory. Follow the procedures described in "Determining Which Log Files to Use in the Restore Process" in the Explanation of `tp_retore` command in the VOS Transaction Processing Facility Reference Manuals to research the starting log file for a transaction data file. When the destination directory contains a transaction data file that was not included in the previous tp_restore session, the next `tp_restore` session knows that it must read **all** log files in the restore directory. However, when the transaction data files in the restore directory are identical to the files used in the previous session, `tp_restore` starts with the continuation log it identified in the previous session's `.out` file.

If you do not specify the `-continuation` argument, or if you are performing the first `tp_restore` command in a planned series, `tp_restore` looks for certain information in the log files that ensures a proper starting point for rolling forward transactions for each of the files in the destination directory. You must research which log files to move to the restore directory. Follow the procedures described in "Determining Which Log Files to Use in the

Restore Process” in the Explanation of `tp_retore` command in the VOS Transaction Processing Facility Reference Manuals.

Where to Perform `tp_restore`

All transaction log files required for the roll forward must be moved into a single directory on the module from which you plan to issue the `tp_restore` command. There must be enough disk space available to the module to hold all log files. If you are moving the transaction data-file backups to a single directory for the roll forward, space must be available to accommodate those files as well. If the transactions being applied will expand the transaction data files, then space must also be available for the file expansions.

The `tp_restore` command executes faster when all transaction data files being rolled forward are located in a single directory, compared to when the roll forward is performed in place.

Restarting the `tp_restore` Command

If the `tp_restore` command is interrupted before it completes, restart it by reissuing the `tp_restore` command, using the same arguments you gave initially. Do not move or update any files in the `-destination` or `-restore_dir` directories. The roll forward will then continue where it left off.

Verifying the TP Restore Process

The `tp_restore` command creates an output file containing the results of the roll forward. The first item in the output file shows the `tp_restore` command as it was issued. The command is followed by lists for each module involved, showing for each module the names of the log files processed, the number of transactions processed, and the number of transactions committed. A list of the data files that were rolled forward follows the module information.

The last section of the output file shows modules and data files that were referenced in some of the transactions but were not involved in the roll forward. If items are listed in this section, a data file was omitted from the information supplied in the `tp_restore` command.

Example Using `tp_restore`

Sample output from a roll forward is shown in Figure 6-2. The figure illustrates a `tp_restore` command that rolls forward three data files in place. Two of the files (`file3` and `file4`) originally resided on the module that issued the `tp_restore` command. This module is associated with the `%prod#deptA` system/disk name. The module ID is 10-17. The third file (`file5`) originally resided on the module associated with the `%prod#deptB` system/disk name. The module ID is 10-18.

```
tp_restore -control %prod#deptA>restore>control_file -restore_dir
```

```
%prod#deptA>restore>logs

Transaction logs processed:
  Module: %prod#deptA (system_no: 10, module_no: 17)
    Number of transactions encountered: 265
    Number of transactions committed:   193
    Log: %prod#deptA>restore>logs>tlf.10-17.0000000001.89-12-11
    Log: %prod#deptA>restore>logs>tlf.10-17.0000000002.89-12-11
         restored to transaction 09C4CECA020700F4

  Module: %prod#deptB (system_no: 10, module_no: 18)
    Number of transactions encountered: 152
    Number of transactions committed:   114
    Log: %prod#deptA>restore>logs>tlf.10-18.0000000001.89-12-11
    Log: %prod#deptA>restore>logs>tlf.10-18.0000000002.89-12-11
         restored to transaction 09C4CECA020700F4


Files restored:
  File: %prod#deptA>restore>data_files>file3
        restored to transaction 09C4CECA020700F4
  File: %prod#deptA>restore>data_files>file4
        restored to transaction 09C4CECA020700F4
  File: %prod#deptB>restore>data_files>file5
        restored to transaction 09C4CECA020700F4

Modules referenced but not involved during restore
  Module: %prod#deptC
```

Figure 6-2. `tp_restore` Output File - Example 1

For our example, the original names of the transaction log files were as follows:

```
%prod#deptA>system>transaction_logs>tlf.10-17.0000000001.89-12-11
%prod#deptA>system>transaction_logs>tlf.10-17.0000000002.89-12-11
%prod#deptB>system>transaction_logs>tlf.10-18.0000000001.89-12-11
%prod#deptB>system>transaction_logs>tlf.10-18.0000000002.89-12-11
```

For the roll-forward operation, the log files were moved to the directory named `%prod#deptA>restore>logs`. The object names of the transaction logs were not changed, so the system-module IDs 10-17 and 10-18 remain embedded in the object names.

The `tp_restore` command in Figure 6-2 specifies that a control file named `%prod#deptA>restore>control_file` contains the path names of the transaction data files when the transactions were originally committed. The contents of the control file are as follows:

```
%prod#deptA>application>file3
%prod#deptA>application>file4
%prod#deptB>application>file5
```

The `tp_restore` command uses the first component of these path names as entries into the system configuration tables. Assuming that the system configuration has not changed since the files were created, `tp_restore` would infer 10-17 as the module ID for `file3` and `file4`, and it would infer 10-18 as the module ID for `file5`. The `tp_restore` command will use log files whose object name contains 10-17 to roll forward the files named `file3` and `file4`. It will use log files whose object name contains 10-18 to roll forward `file5`.

The `-destination` argument is not specified in the sample `tp_restore` command in Figure 6-2. This means that the backup versions of the transaction data files replaced the corrupted versions of the files in their original directories.

Example Using `tp_restore` from a Different System

Figure 6-3 illustrates a `tp_restore` command performed on a system other than the one where the application normally runs. The roll forward is performed on system `s1`, disk `m1`. The application normally runs on disks named `deptA` and `deptB` on the system named `prod`.

```
tp_restore -control %s1#m1>restore>control_file -restore_dir
%s1#m1>restore>logs -destination %s1#m1>restore>data_files

Transaction logs processed:
  Module: %???#??? (system_no: 10, module_no: 17)
    Number of transactions encountered: 265
    Number of transactions committed:   193
    Log: %s1#m1>restore>logs>tlf.10-17.0000000001.89-12-11
    Log: %s1#m1>restore>logs>tlf.10-17.0000000002.89-12-11
         restored to transaction 09C4CECA020700F4

  Module: %???#??? (system_no: 10, module_no: 18)
    Number of transactions encountered: 152
    Number of transactions committed:   114
    Log: %s1#m1>restore>logs>tlf.10-18.0000000001.89-12-11
    Log: %s1#m1>restore>logs>tlf.10-18.0000000002.89-12-11
         restored to transaction 09C4CECA020700F4


Files restored:
  File: %s1#m1>restore>data_files>file3
        restored to transaction 09C4CECA020700F4
  File: %s1#m1>restore>data_files>file4
        restored to transaction 09C4CECA020700F4
  File: %s1#m1>restore>data_files>file5
        restored to transaction 09C4CECA020700F4

Modules referenced but not involved during restore
  Module: %prod#deptC
```

Figure 6-3. `tp_restore` Output File - Example 2

For our example, the original names of the transaction log files were as follows:

```
%prod#deptA>system>transaction_logs>tlf.10-17.0000000001.89-12-11
%prod#deptA>system>transaction_logs>tlf.10-17.0000000002.89-12-11
%prod#deptB>system>transaction_logs>tlf.10-18.0000000001.89-12-11
%prod#deptB>system>transaction_logs>tlf.10-18.0000000002.89-12-11
```

The log-file object names indicate that the system-module IDs for the modules named `deptA` and `deptB` are 10-17 and 10-18.

For the roll-forward operation, the log files were moved to system `s1`, disk `m1`, in the directory named `%s1#m1>restore>logs`. The object names of the transaction logs were not changed, so the system-module IDs 10-17 and 10-18 remain embedded in the object names. The IDs identify the logs as being created on system `prod`, disks `deptA` and `deptB`.

The `tp_restore` command in Figure 6-3 specifies that a control file named `%s1#m1>restore>control_file` contains the path names of the transaction data files when the transactions were originally committed. The contents of the control file are as follows:

```
%prod#deptA>application>file3 10-17
%prod#deptA>application>file4 10-17
%prod#deptB>application>file5 10-18
```

Note that the path names are followed by the system-module IDs identifying where the files resided when the transactions were originally committed. This is necessary because system `s1` does not include entries in its configuration tables for system `prod`. The `tp_restore` command will use log files whose object name contains 10-17 to roll forward the files named `file3` and `file4`. It will use log files whose object name contains 10-18 to roll forward `file5`.

In the sample output shown in Figure 6-3, the system and module names appear as question marks because the names were unavailable for inference. (The configuration tables for system `s1` do not contain entries for module IDs 10-17 and 10-18.)

The sample `tp_restore` command in Figure 6-3 shows that the backups of the transaction data files were moved to a directory called `%s1#m1>restore>data_files`. When the backups were installed, they were named so that their object names match the object names of the files listed in the control file. The `%s1#m1>restore>data_files` directory contains the following files:

```
#m1>restore>data_files>file3
#m1>restore>data_files>file4
#m1>restore>data_files>file5
```

Appendix A:
List of Transaction Processing Facility Commands and Requests

This appendix briefly describes the Transaction Processing Facility commands and requests. It contains the following sections.

- ‘‘Transaction Processing Facility Commands”
- ‘‘Monitor Requests”

For complete documentation on these commands and requests, see the VOS Transaction Processing Facility Reference manuals.

Transaction Processing Facility Commands

The Transaction Processing Facility commands are issued at command level like the standard VOS commands. Except for `tp_restore`, all of these commands can also be executed from the monitor task (as can all other internal VOS commands). To execute an internal command from the monitor task, precede the command by two periods. The `tp_restore` command is not an internal command.

Except for the commands dealing with lock wait time, these Transaction Processing Facility commands relate specifically to Transaction Processing Facility capabilities. (Lock wait time affects all processes, not just Transaction Processing Facility processes.)

For more information about these commands, see the VOS Transaction Processing Facility Reference manuals.

`display_lock_wait_time`
: Displays the default maximum time (in seconds) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

`display_process_lock_wait_time`
: Displays the maximum time (in seconds) that the current process (and any tasks that are part of that process) will wait to acquire an implicit lock during any I/O operation.

`display_tp_default_parameters`
: Displays the default lock contention parameters and their values for transaction locking on a specified module.

`display_tp_parameters`
: Displays the lock contention parameters and their values for transaction locking for the current process.

`list_messages`
: Lists the messages currently in a server or message queue.

`set_lock_wait_time`
: Sets the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation. The default wait time applies to all tasks and processes running on a given module or set of modules (unless the lock wait time for a task or process has been specifically changed from the default) and to all types of I/O operation. You must be privileged to use this command.

`set_max_queue_depth`
: Sets the maximum queue depth (or maximum number of messages) of a one-way or two-way server queue.

`set_process_lock_wait_time`
: Sets the maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation.

`set_tp_default_parameters`
: Sets the default lock-contention parameters for transaction locking for a specified module.

`set_tp_parameters`
: Sets the lock-contention parameters for transaction locking for the current process.

`set_transaction_file`
: Converts one or more specified ordinary files into transaction files, or converts one or more specified transaction files into ordinary files.

`tp_restore`
: Uses a transaction log to reconstruct a later state of a file from an earlier saved state.

Monitor Requests

The monitor requests are precoded requests that the programmer can make available to the monitor task. Unlike commands, requests cannot be issued at command level. They can be issued only from within the monitor task. To make a request available to a monitor task, bind the request name with the monitor task's program module.

`cleanup_task`
: Closes the terminal assigned to each specified task and detaches the port attached to the terminal.

`control_task`
: Changes the state of each specified task in the current process.

`create_task`
: Creates a dynamic task.

`delete_message`
: Removes a message from a message queue.

`delete_task`
: Deletes a dynamic task.

`display_task_info`
: Displays information about one or all tasks.

`display_tp_parameters`
: Displays the lock-contention parameters for transaction locking for the current program or process.

`help`
: Lists all of the requests you can make from the current monitor program. This request is always available. Do not bind it in the program module.

`initialize_configuration`
: Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`initialize_task`
: Initializes a specified uninitialized task.

`list_messages`
: Lists the messages currently in a server or message queue.

`quit`
: Tells the `s$monitor` or `s$monitor_full` subroutine, whichever is running, to stop accepting requests and to return to its caller. This request is always available. Do not bind it in the program module.

`set_no_wait_mode`
: Puts a specified port into no-wait mode.

`set_task_priority`
: Sets the scheduling priority for a task.

`set_tp_parameters`
: Sets the lock-contention parameters for transaction locking for the current program.

`set_wait_mode`
: Puts a specified port into wait mode.

`start_task`
: Makes each specified task that is initialized or stopped ready to run.

`stop_task`
: Stops each specified task.

Appendix B: Sample Application

This appendix illustrates a sample application using Transaction Processing Facility capabilities. The sample application is an automated auction system. It permits buyers to submit bids for auction items from locations remote from the auction display site. The remote terminals allow the users to view the current highest bid submitted on auction items and submit their own bids for items.

The application uses the requester/server design model. It uses tasking to manage the end-user terminals, with a monitor task to administer the system. A two-way server queue transmits the I/O requirements from the requester program to the server program. Requests for information and bid submissions from users are transactions requiring transaction protection.

This appendix contains the following sections.

FMS Screens Displayed by the Sample Application

The sample application uses Forms Management System (FMS) statements to display two screens on the terminals attached to the user tasks in the requester program.

Menu Screen

```
                 WELCOME TO THE AUTOMATED AUCTION SYSTEM
TYPE THE NUMBER OF THE FUNCTION THAT YOU WANT TO PERFORM HERE: _
1- DISPLAY AUCTION ITEM NUMBERS
2- DISPLAY ITEM INFORMATION
3- SUBMIT A BID ON AN ITEM
```

Bid Submit Screen

```
                       BID SUBMISSION
YOUR BIDDER ID: ______
ITEM NO   DESCRIPTION        HIGH BID   YOUR BID
______    ____________________   ______     ______
NEXT ACTION - CHOOSE ONE: _
i - display INFORMATION about an item
b - submit a BID on an item
n - clear the screen and start a NEW bid
q - QUIT and return to the menu screen
```

Sample Data for Bidder File

The following is sample data for the bidder file.

```
999999xxxxxxxxxxxxxxxxxxxx999999888888
111111smith,john          010000005595
222222brown,susan         010000003541
333333miller,george       010000001355
444444wood,mike           010000000000
555555dodson,jane         010000000030
```

Sample Data for Item File

The following is sample data for the item file.

```
999999xxxxxxxxxxxxxxxxxxxx999999888888
000001riding lawn mower   001306222222
000002diamond necklace    005595111111
000003mahogany desk       000030555555
000004birchbark canoe     002235222222
000005mustang             001355333333
```

Server Program Binder Control File

The following is the server binder control file.

```
name:                    server;
entry:                   server;
modules:                 server;
retain:                  epilogue_for_program;
end;
```

Requester Program Binder Control File

The following is the requester binder control file.

```
name:                    requester;
entry:                   requester;
number_of_tasks:         5;
modules:                requester,              /* source program     */

                        list_messages,          /* monitor requests   */
                         display_task_info;

retain:                 menu_entry,             /* user task entry    */

                        list_messages,          /* monitor requests   */
                         display_task_info,

                       epilogue_for_tasks,      /* epilogue entries   */
                         program_epilogue_handler;

end;
```

Server Program — COBOL Version

```
       identification division.
       program-id.  server.
************************************************************************
******* This is the server program in a requester/server application.
******* It reads messages from a two-way server queue, services the
******* messages, and sends back replies.  Two types of messages are
******* serviced.  One type is a simple request for information about an
******* auction item.  To service this type of request, the program reads a
******* transaction file to obtain the information and sends the
******* information to the requester in the reply to the message.
******* The other type of message serviced by this program is bid
******* submissions.  For these types of messages, the program
******* validates the bid submission, and, if the request is valid, the
******* program updates two transaction files.  The program tells the
******* requester whether the bid was accepted in the reply to the message.
******* If the bid was not accepted, the reply is an appropriate error
******* message.
*******
******* This program stops when a stop message is received from the
******* monitor task in the requester program.
*******
******* This program must be executed in a privileged process.  It can
******* be executed in a background process.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  This program starts two types
******* of transactions - one is a request for information and the other is
******* a bid submission.  Suppose that a user in a task asks for
******* information about an item (one transaction) and then submits a bid
******* on the item (a second transaction).  It is possible that, in
******* between these two transactions, a user in another task gets a bid
******* accepted on the same item.  This would make the information that
******* the first user received as a result of the first transaction
******* out of date.
*******
******* This situation can be avoided if the two types of transactions are
******* combined into a single transaction.  This change could be
******* efficiently implemented by having the requester program start
******* and end the transaction, based on the user's input.  The server
******* program would process information on behalf of the requester's
******* transactions, and this processing would be transaction-protected.
******* The server program can also abort the transaction.
******* See the section called "Transaction Protection and Two-Way Server
******* Queues" in Chapter 5, "Queues," of this manual for a description
******* of this capability.  The section called "Server Queue Scenarios,"
******* also in Chapter 5, illustrates how to use the feature.

       environment division.
       input-output section.
       file-control.
           select bidder_file assign to bidder_port
```

```
         organization is indexed
         access mode dynamic
         record key is bidder_id
         error code is error_code.
    select item_file assign to item_port
         organization is indexed
         access mode dynamic
         record key is item_no
         error code is error_code.
data division.
file section.
fd  bidder_file
    external
    value of pathname is "bidder_file".
01  bidder_record.
    05  bidder_id             pic x(6).
    05  name                  pic x(20).
    05  amt_approved          pic 9(6).
    05  amt_spent             pic 9(6).
fd  item_file
    external
    value of pathname is 'item_file'.
01  item_record.
    05  item_no               pic x(6).
    05  description           pic x(20).
    05  high_bid              pic 9(6).
    05  last_bidder_id        pic x(6).
working-storage section.
01  bidder_port_id            comp-4.
01  call_debug                comp-4.
01  caller                    pic x(32) display-2
         value 'server'.
01  char_string               pic x(16) display.
01  chase_switch              comp-4 value 0.
01  command_line              pic x(256) display-2.
01  config_path               pic x(256) display-2.
01  counter                   pic 9.
01  duration_switches         comp-4.
01  entry_name                pic x(32) display-2.
01  entry_value               entry.
01  epilogue_handler          entry.
01  error_code                comp-4.
01  e$file_in_use             comp-4 global value 1084.
01  e$key_in_use              comp-4 global value 2918.
01  e$object_not_found        comp-4 global value 1032.
01  e$record_not_found        comp-4 global value 1112.
01  e$record_in_use           comp-4 global value 2408.
01  e$tp_aborted              comp-4 global value 2931.
01  file_organization         comp-4.
01  io_success                pic x.
01  io_type                   comp-4.
01  item_port_id              comp-4.
01  maximum_record_length     comp-4.
01  message_reply.
```

```
    05  description            pic x(20).
    05  high_bid               pic 9(6).
    05  error_message          pic x(70).
%replace REPLY_SIZE by 96
01  msg_header.
    05  version                comp-4 value 1.
    05  header_msg_id          comp-5.
    05  date_time_queued       comp-5.
    05  requester_priority     comp-4.
    05  msg_subject            pic x(32) display-2.
    05  requester_process_id   comp-5.
    05  requester_user_name    pic x(32) display-2.
    05  requester_user_group   pic x(32) display-2.
    05  server_process_id      comp-5.
    05  switches               comp-4.
    05  requester_maximum_priority    comp-4.
    05  requester_maximum_processes   comp-4.
    05  jiffies_queued         comp-4.
    05  reserved1              comp-4 occurs 10.
    05  language               pic x(32) display-2.
    05  character_set          comp-4.
    05  reserved2              comp-4 occurs 3.
01  msg_id                     comp-5.
01  req_message.
    05  next_action            pic x.
    05  your_bidder_id         pic x(6).
    05  item_no                pic x(6).
    05  your_bid               pic 9(6).
%replace REQ_MESSAGE_SIZE by 19
01  reply_length_in            comp-5.
01  msg_length_in              comp-5.
01  msg_length_out             comp-5.
01  msg_priority               comp-4.
01  msg_selector               comp-4.
01  null_pointer               pointer.
01  object_type                comp-4.
01  port_name                  pic x(32) display-2.
01  protection_switch          comp-4.
01  rel_path_name              pic x(256) display-2.
01  save_last_bid              pic 9(6).
01  save_last_bidder           pic x(6).
01  server_queue_path_name     pic x(256) display-2.
01  server_queue_port_id       comp-4.
01  stop_application_flag      pic x.
    88 time_to_stop            value 'y'.
    88 keep_running            value 'n'.
01  suffix                     pic x(32) display-2 value ''.
01  time_period                comp-5.
01  trans_aborted              pic x.
01  trans_file_path_name       pic x(256) display-2.
01  transaction_priority       comp-4.
01  where_occurred             pic x(32) display-2.
```

```
       %replace FIRST_MESSAGE_NON_BUSY by 1
       %replace HIGH_PRIORITY by 9
       %replace MAX_RETRIES by 5
       %replace SERVER by 5
       %replace SERVER_IO_TYPE by 6
       %replace TRUE by 1

       procedure division.
*****************************************************************
*******    Set all bits in duration_switches to zero.  This variable
*******    is used by the s$attach_port subroutine.
*****************************************************************
          move 0 to duration_switches.
*****************************************************************
******* TP1.  Declare the Bidder file to be a transaction file.
*******       The s$set_transaction_file subroutine must be called
*******       from a privileged process.
*****************************************************************
           move 'bidder_file' to rel_path_name.
           call 's$expand_path' using rel_path_name,
                                      suffix,
                                      trans_file_path_name,
                                      error_code.
           if error_code not = 0
              move 's$expand_path: bidder_file' to
                         where_occurred
              perform 999_fatal_error.
           move TRUE to protection_switch.
           call 's$set_transaction_file' using trans_file_path_name,
                                               protection_switch,
                                               error_code.
           if error_code not = 0
               move 'error from s$set_trans_file bidder'
                      to where_occurred
               perform 999_fatal_error.
*****************************************************************
******* TP2.  Declare the Item file to be a transaction file.
*****************************************************************
           move 'item_file' to rel_path_name.
           call 's$expand_path' using rel_path_name,
                                      suffix,
                                      trans_file_path_name,
                                      error_code.
           if error_code not = 0
              move 's$expand_path: item_file' to
                         where_occurred
              perform 999_fatal_error.
           call 's$set_transaction_file' using trans_file_path_name,
                                               protection_switch,
                                               error_code.
           if error_code not = 0
              move 'error from s$set_trans_file item' to where_occurred
               perform 999_fatal_error.
```

```
*****************************************************************
******* TP3. Attach ports to transaction files and open the files.
*****************************************************************
           open i-o bidder_file.
           open i-o item_file.
*****************************************************************
******* Q1.  Create a two-way server queue if one does not exist.
*******      The test to see if a queue already exists permits
*******      multiple server processes to simultaneously execute
*******      this program.
*****************************************************************
           move '2-way_server_queue' to rel_path_name.
           move '' to suffix.
           call 's$expand_path' using rel_path_name,
                                      suffix,
                                      server_queue_path_name,
                                      error_code.
           if error_code not = 0
              move 's$expand_path: server_queue' to where_occurred
              perform 999_fatal_error.
           call 's$get_object_type' using server_queue_path_name,
                                          chase_switch,
                                          object_type,
                                          error_code.
           if error_code = 0 or error_code = e$object_not_found
              next sentence
           else
              move 'error from s$get_object_type'
                                              to where_occurred
              perform 999_fatal_error.
           if error_code = e$object_not_found
              move SERVER to file_organization
              move 20 to maximum_record_length
              call 's$create_file' using server_queue_path_name,
                                         file_organization,
                                         maximum_record_length,
                                         error_code
              if error_code not = 0
                 move 'error from s$create_file' to where_occurred
                 perform 999_fatal_error.

*****************************************************************
******* Q2.  Attach a port to the queue.
*****************************************************************
           move 'server_1' to port_name.
           call 's$attach_port' using port_name,
                                      server_queue_path_name,
                                      duration_switches,
                                      server_queue_port_id,
                                      error_code.
           if error_code not = 0
               move 'error from s$attach_port' to where_occurred
               perform 999_fatal_error.
```

```
*****************************************************************
******* Q3. Open the port to the queue.
********************************************************************
           move SERVER_IO_TYPE to io_type.
           call 's$msg_open' using server_queue_port_id,
                                   io_type,
                                   error_code.
           if error_code not = 0
               move 'error from s$msg_open' to where_occurred
               perform 999_fatal_error.
***********************************************************************
**********     Add program epilogue handler.
***********************************************************************
           move  !null() to null_pointer.
           move 'epilogue_for_program' to entry_name.
           call 's$find_entry' using null_pointer,
                                     entry_name,
                                     entry_value,
                                     error_code.
           if error_code not = 0
               move 'error from s$find_entry - prog epilogue'
                                    to where_occurred
               perform 999_fatal_error.

           call 's$add_epilogue_handler' using
                                     entry_value,
                                     error_code.
           if error_code not = 0
               move 'error from s$add_epilogue_handler'
                                      to where_occurred
               perform 999_fatal_error.

************************************************************************
******
*******    Continuously process new messages from the queue. Stop when the
*******     monitor task in the requester program sends a message saying
*******     that the application is stopping.  The requester program
*******    stops all user tasks before it sends this message, so this is
*******     the last message in the queue.  The server can stop itself
*******    immediately and will not leave any user messages unserviced.
************************************************************************
           move 'n' to stop_application_flag.
           perform 100_process_new_message with test before
              until time_to_stop.
************************************************************************
*******    Stop the progam.  The program epilogue handler will execute
*******    after s$stop_program.
***********************************************************************
           move 0 to error_code
           call 's$stop_program' using command_line,
                                       error_code.
```

```
       100_process_new_message.
************************************************************************
******* Q4.  Receive a message from the queue.  Since the port is in wait
*******      mode, if there are no messages in the queue, the program will
*******     suspend in s$msg_receive until a message is added to the queue.
************************************************************************
           move FIRST_MESSAGE_NON_BUSY to msg_selector.
           move REQ_MESSAGE_SIZE to msg_length_in.
           call 's$msg_receive' using server_queue_port_id,
                                 msg_selector,
                                 msg_id,
                                 msg_header,
                                 msg_length_in,
                                 msg_length_out,
                                 req_message,
                                 error_code.
           if error_code not = 0
               move 'error from s$msg_receive' to where_occurred
               perform 999_fatal_error.
*****************************************************************
*******       Initialize reply fields.
*****************************************************************
           move spaces to description of message_reply,
                          error_message of message_reply.
           move 0 to high_bid of message_reply.
*****************************************************************
*******       Set the stop flag to 'y' if the monitor task in the
*******       requester program sent a stop message.
*****************************************************************
             if msg_subject = 'MONITOR SAYS STOP'
                move 'received stop message -- stopping now' to
                   error_message of message_reply
                move 'y' to stop_application_flag
             else
                if next_action of req_message = 'i'
                     perform 200_get_info
             else
                if next_action of req_message = 'b'
                     perform 300_submit_bid
             else
                move 'invalid action code' to error_message
                            of message_reply.

************************************************************************
******* Q5.  Reply to the message.  The reply contains the results of
*******      servicing the message.
************************************************************************
             move REPLY_SIZE to reply_length_in.
             call 's$msg_send_reply' using server_queue_port_id,
                                      msg_id,
                                      reply_length_in,
                                      message_reply,
                                      error_code.
             if error_code not = 0
                 move 'error from msg_send_reply' to where_occurred
                 perform 999_fatal_error.

       200_get_info.
           move 'n' to trans_aborted, io_success.
           move 0 to counter.
           move 10 to time_period.
*****************************************************************
******* TP4. Start a transaction that obtains information from
*******      the Item file.
*****************************************************************
           call 's$start_transaction' using error_code.
           if error_code not = 0
               move 'error from s$start_trans' to where_occurred
               perform 950_unexpected_error.
************************************************************************
******* TP5. Perform a transaction that obtains information about the
*******      item number sent in the message.  The transaction
*******      reads a record in the Item file.  It keeps trying
*******      the I/O call until either the transaction is successful,
*******     the transaction is aborted, or five attempts have been made.
************************************************************************
           perform 210_info_transaction until
                 (trans_aborted = 'y' or io_success = 'y' or
                 counter = MAX_RETRIES).
************************************************************************
******* TP6. If the read was successful, commit the transaction.
************************************************************************
           if trans_aborted = 'n' and io_success = 'y'
             call 's$commit_transaction' using error_code
             if error_code not = 0
                 move 'error from s$commit_trans'
                                         to where_occurred
                 perform 950_unexpected_error
             else
                 move description of item_record to
                                 description of message_reply
                 move high_bid of item_record
                                 to high_bid of message_reply.

       210_info_transaction.
**********************************************************************
******* TP7. Read the Item file.
**********************************************************************
           move item_no of req_message to item_no of item_file
           read item_file record into item_record
               key is item_no of item_file
               invalid key continue.
```

```
      ************************************************************************
      ******* TP8. Test for the e$record_not_found error message.  Then test
      *******      for other errors related to locking.
      ************************************************************************
                 if error_code not = 0
                    if error_code = e$record_not_found
                         move 'You entered an invalid item id' to
                                           error_message of message_reply
                         move spaces to item_record
                         call 's$abort_transaction' using error_code
                         if error_code not = 0
                            move 'error from s$abort_transaction'
                                                   to where_occurred
                            perform 950_unexpected_error
                         else
                            move 'y' to trans_aborted
                     else
                         perform 900_common_error_code_check
                   else
                     move 'y' to io_success.

             300_submit_bid.
      ************************************************************************
      ******* TP9.  Start a transaction that updates two transaction files.
      *******       Assign a higher priority to transactions with dollar
      *******       amounts > $10000.
      ************************************************************************
                 if your_bid of req_message < 10000
                    call 's$start_transaction' using error_code
                 else
                    move HIGH_PRIORITY to transaction_priority
                    call 's$start_priority_transaction' using
                                               transaction_priority,
                                               error_code.
                 if error_code not = 0
                    move 'error from server start trans' to where_occurred
                    perform 950_unexpected_error.
      ************************************************************************
      ******* TP10. Perform the transaction.  Retry each I/O attempt up to five
      *******       times.  If the transaction is aborted, do not perform the
      *******       rest of the transaction.
      ************************************************************************
                 move 'n' to trans_aborted.

                 move 0 to counter.
                 move 10 to time_period.
                 move 'n' to io_success.
                 perform 310_file_io_1 until
                  (trans_aborted = 'y' or counter = MAX_RETRIES
                                                    or io_success = 'y').
```

```
           if trans_aborted = 'n'
             move 0 to counter
             move 10 to time_period
             move 'n' to io_success
             perform 320_file_io_2 until
               (trans_aborted = 'y' or counter = MAX_RETRIES
                                        or io_success = 'y').

           if trans_aborted = 'n'
             move 0 to counter
             move 10 to time_period
             move 'n' to io_success
             perform 330_file_io_3 until
              (trans_aborted = 'y' or counter = MAX_RETRIES
                                        or io_success = 'y').

           if trans_aborted = 'n'
             move 0 to counter
             move 10 to time_period
             move 'n' to io_success
             perform 340_file_io_4 until
              (trans_aborted = 'y' or counter = MAX_RETRIES
                                        or io_success = 'y').

           if trans_aborted = 'n'
             move 0 to counter
             move 10 to time_period
             move 'n' to io_success
             perform 350_file_io_5 until
              (trans_aborted = 'y' or counter = MAX_RETRIES
                                        or io_success = 'y').
************************************************************************
******* TP11. Attempt to commit the transaction if it has not been aborted.
*******      If the commit is not successful, either ask the user to try
*******      again later or perform the error routine, depending on the
*******      error code.
************************************************************************
           if trans_aborted = 'n'
              call 's$commit_transaction' using error_code
              if error_code not = 0
                if error_code = e$tp_aborted
                move 'Your bid cannot be processed - please try later'
                   to error_message of message_reply
                else
                  move 'error from s$commit_trans' to where_occurred
                  perform 950_unexpected_error
              else
                  move 'Bid accepted' to error_message
                                     of message_reply.
```

```
       310_file_io_1.
****************************************************************
*******       Read the Bidder file and validate information.
****************************************************************
             move your_bidder_id of req_message to bidder_id
                                                of bidder_file.
             read bidder_file record into bidder_record
                  key is bidder_id of bidder_file
                  invalid key continue.
             if error_code not = 0
                if error_code = e$record_not_found
                   move 'You are not an authorized bidder' to
                                   error_message of message_reply
                   call 's$abort_transaction' using error_code
                   if error_code not = 0
                      move 'error from s$abort_transaction'
                                             to where_occurred
                      perform 950_unexpected_error
                   else
                      move 'y' to trans_aborted
                else
                   perform 900_common_error_code_check
             else
                move 'y' to io_success
                if (amt_approved) - (amt_spent + your_bid) < 0
                    move 'You do not have that much money' to
                               error_message of message_reply
                    call 's$abort_transaction' using error_code
                   if error_code not = 0
                      move 'error from s$abort_transaction'
                                     to where_occurred
                      perform 950_unexpected_error
                   else
                    move 'y' to trans_aborted.
           if trans_aborted = 'y' and error_code not = 0
              move 'error from s$abort_trans' to where_occurred
              perform 950_unexpected_error.

       320_file_io_2.
****************************************************************
*******       Update the Bidder file.
****************************************************************
            compute amt_spent = amt_spent + your_bid of req_message
            rewrite bidder_record
               invalid key continue.
            if error_code not = 0
               perform 900_common_error_code_check
            else
               move 'y' to io_success.
       330_file_io_3.
****************************************************************
*******       Read the Item file and validate information.
****************************************************************
           move item_no of req_message to item_no of item_file
           read item_file record into item_record
              key is item_no of item_file
              invalid key continue.
           if error_code not = 0
              if error_code = e$record_not_found
                 move 'You entered an invalid item number' to
                             error_message of message_reply
                 call 's$abort_transaction' using error_code
                 if error_code not = 0
                    move 'error from s$abort_transaction'
                                           to where_occurred
                    perform 950_unexpected_error
                 else
                    move 'y' to trans_aborted
              else
                 perform 900_common_error_code_check
           else
              move 'y' to io_success
              move description of item_record
                                   to description of message_reply
              move high_bid of item_record
                                   to high_bid of message_reply
              if your_bid of req_message <= high_bid of item_record
                 move 'you did not outbid the previous bidder' to
                            error_message of message_reply
                 call 's$abort_transaction' using error_code
                 move 'y' to trans_aborted.
           if trans_aborted = 'y' and error_code not = 0
                move 'error from s$abort_trans' to where_occurred
                perform 950_unexpected_error.

         340_file_io_4.
****************************************************************
*******       Update the Item file.
****************************************************************
             move high_bid of item_record to save_last_bid.
             move last_bidder_id of item_record to save_last_bidder.

             move your_bid of req_message to high_bid of item_record
             move your_bidder_id of req_message to last_bidder_id
             rewrite item_record
               invalid key continue.
             if error_code not = 0
                  perform 900_common_error_code_check
             else
                 move 'y' to io_success.
         350_file_io_5.
************************************************************************
*******      Find the previous high bidder on the item and refund
*******      that bidder's account for the amount of the previous
*******      high bid.
************************************************************************
             move save_last_bidder to bidder_id of bidder_file
             read bidder_file record into bidder_record
                  key is bidder_id of bidder_file
                  invalid key continue.
             if error_code not = 0
********************************************************************
*******      Do nothing if the previous bidder is not in the file.
********************************************************************
                if error_code = e$record_not_found
                    move 'y' to io_success
                else
                   perform 900_common_error_code_check
             else
                compute amt_spent = amt_spent - save_last_bid
                rewrite bidder_record
                  invalid key continue
                end-rewrite
                if error_code not = 0
                    perform 900_common_error_code_check
                else
                    move 'y' to io_success.


       900_common_error_code_check.
************************************************************************
******  TP12. Check for the e$record_in_use, e$key_in_use, and
******        e$file_in_use error messages.  Retry the I/O call if TP
******        locks were not obtained.  If an I/O call is not successful
******        after five attempts, abort the transaction.  The length of
******        the sleep time increases with each retry.
************************************************************************
            if error_code = e$record_in_use or
               error_code = e$key_in_use or
               error_code = e$file_in_use
            then
               add 1 to counter
               add 10 to time_period
               call 's$sleep' using time_period,
                                    error_code
               if error_code not = 0
                 move 'error from s$sleep' to where_occurred
                 perform 950_unexpected_error
               else
                 next sentence
            else
               perform 950_unexpected_error.
```

```
           if counter = MAX_RETRIES
             move 'y' to trans_aborted
             move 'The system is busy - try your bid again'
                   to error_message of message_reply
             call 's$abort_transaction' using error_code
             if error_code not = 0
                move 'error from s$abort_transaction' to where_occurred
                perform 950_unexpected_error.

       950_unexpected_error.
************************************************************************
******* On miscellaneous errors within a transaction, try to clean
******* up the current message.  (Send a reply
******* to the message, record the error in the server.out log, and
******* abort the transaction.) If the abort is not successful,
******* call the fatal error routine.  Otherwise, the server continues
******* processing other messages.
************************************************************************
           move 'Request not completed. Program error. Notify SysAdmin.'
               to error_message of message_reply.
           display 'unexpected error in a transaction'.
           call 's$error' using error_code,
                                caller,
                                where_occurred.
           call 's$abort_transaction' using error_code.
           if error_code not = 0
              move 'error from s$abort_trans' to where_occurred
              perform 999_fatal_error
           else
              move 'y' to trans_aborted.
       999_fatal_error.
************************************************************************
*******    Stop the program on miscellaneous errors.
************************************************************************
           call 's$error' using error_code,
                                caller,
                                where_occurred.
           move 0 to error_code.
           call 's$stop_program' using command_line,
                                       error_code.


           entry 'epilogue_for_program'.
         begin_program_epilogue.
************************************************************************
******* Q6. Provide the program epilogue handler.  The program
*******     epilogue handler closes the transaction files and the queue.
************************************************************************
           display 'entering program epilogue.'
           close bidder_file.
           close item_file.
           call 's$close' using server_queue_port_id,
                                error_code.
```

```
if error_code not = 0
   move 'error from closing queue' to where_occurred
   move 'server program epilogue' to caller
   call 's$error' using error_code,
                        caller,
                        where_occurred.
exit program.
```

Requester Program — COBOL Version

```
       identification division.
       program-id.  requester.
************************************************************************
******* This is the requester program in a requester/server application.
******* This is a tasking program, with one monitor task and many user
******* tasks.  All user tasks have the same entry point (menu_entry).
*******
******* The user tasks display forms on the users' terminals.  Users enter
******* requests, which are changed into messages and sent to a two-way
******* server queue.  The server program receives the messages, services
******* them, and sends back replies.  The user tasks receive the replies
******* and put the results in the FMS forms displayed on the users'
******* terminals.
*******
******* The monitor task controls when the application stops.  When the
******* monitor task wants to stop, it sets a flag that causes each user
******* task to stop after its current message is completely serviced.
******* Then the monitor task stops itself.  When all tasks are stopped,
******* the program epilogue handler executes.  It sends a message to the
******* server program telling the server program to stop.  When the
******* program epilogue handler receives the reply to this message,
******* this program stops.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  The design described above
******* means that there can only be one requester process and one server
******* process.  To stop multiple servers, the requester could repeatedly
******* send the stop message until the error message
******* e$no_msg_server_for_queue is returned.  In the case of multiple
******* requesters, you would probably not want the entire application to
******* stop because one requester wants to stop.  An alternative is to
******* write an external program that controls application startup and
******* shutdown.
************************************************************************

       data division.
       working-storage section.
       01   menu_record.
              copy 'menu.incl.cobol'.
********   20 choice pic x(1).                                    // 07, 69
       01   bid_record.
              copy 'bid_sub.incl.cobol'.
********   20 your_bidder_id pic x(6).                            // 04, 19
********   20 item_no pic x(6).                                   // 11, 05
********   20 description pic x(20).                              // 11, 15
********   20 high_bid pic 9(6).                                  // 11, 42
********   20 your_bid pic 9(6).                                  // 11, 60
********   20 next_action pic x(1).                               // 15, 29
       01   abbrev_switch           comp-4.
       01   action_code             comp-4.
       01   answer                  pic x.
```

```
01   bid_form_id               comp-4.
01   call_debug                comp-4.
01   caller                    pic x(32) display-2.
01   char_string               pic x(16) display.
01   command_line              pic x(256) display-2.
01   config_path               pic x(256) display-2.
01   date_time                 pic x(32) display-2.
01   duration_switches         comp-4.
01   entry_name                pic x(32) display-2.
01   entry_value               entry.
01   error_code                comp-4.
01   e$timeout                 comp-4 value 1081.
01   file_organization         comp-4.
01   form_reply                pic x(50) display-2.
01   io_type                   comp-4.
01   maximum_length            comp-4.
01   maximum_record_length     comp-4.
01   menu_form_id              comp-4.
01   message_info.
     05  next_action           pic x.
     05  your_bidder_id        pic x(6).
     05  item_no               pic x(6).
     05  your_bid              pic 9(6).
%replace MESSAGE_SIZE by 19
01   message_reply.
     05  description           pic x(20).
     05  high_bid              pic 9(6).
     05  error_message         pic x(70).
%replace REPLY_SIZE by 96

01   msg_id                    comp-5.
01   msg_length_in             comp-5.
01   msg_subject               pic x(32) display-2.
01   msg_priority              comp-4.
01   null_pointer              pointer.
01   port_name                 pic x(32) display-2.
01   port_id                   comp-4.
01   prompt_string             pic x(32) display-2.
01   rel_path_name             pic x(256) display-2.
01   reply_length_in           comp-5.
01   reply_length_out          comp-4.
01   server_queue_port_id      comp-4.
01   suffix                    pic x(32) display-2.
01   task_id                   comp-4.
01   task_id_alpha redefines task_id  pic x(4).
01   task_id_msg.
     05  words_in_msg          pic x(24)
                value 'error in requester task '.
     05  task                  pic x(4).
01   task_port_name.
     05  task_words            pic x(5) value 'task '.
     05  task                  pic x(4).
01   terminal_switch           comp-4.
```

```
       01   time_period              comp-4 value 10240.
       01   where_occurred           pic x(50) display-2.
************************************************************************
******* T1.  Specify variables shared by all tasks.
************************************************************************
       01 stop_flag          pic x value 'n' display-2 external global.
       01 server_queue_path_name  pic x(256) display-2
                                                 external global.

       %replace CONFIG_FILE by 'requester_config'
       %replace FALSE by 0
       %replace HIGH_PRIORITY by 17
       %replace LOW_PRIORITY by 10
       %replace TRUE by 1
       %replace STOP_TASK by 1
       %replace CURRENT_TASK by 0
       %replace STOP_TASK_CLEANUP by 7
       %replace REQUESTER_IO_TYPE by 5

       procedure division.
       main_section.
************************************************************************
******* T2.  Begin execution at the program's main entry point. This is
*******      the entry for task 1 by default.
************************************************************************
************************************************************************
******* T3.  Initialize and start the static tasks that are listen the
*******      task configuration table.
************************************************************************
           move 'primary task' to caller.
           move CONFIG_FILE to rel_path_name.
           move '.table' to suffix.
           call 's$expand_path' using rel_path_name,
                                      suffix,
                                      config_path,
                                      error_code.
           if error_code not = 0
              move 's$expand_path: requester_config.table' to
                         where_occurred
              perform 999_error_routine.
           move FALSE to call_debug.
           call 's$init_task_config' using config_path,
                                           call_debug,
                                           error_code.
           if error_code not = 0
              move 'error from s$init_task_config' to where_occurred
              perform 999_error_routine.
```

```
      *****************************************************************
      ******* T4.  Get the shared queue path name (shared by all tasks).
      *******      The queue is created in the server program.
      *****************************************************************
                 move '2-way_server_queue' to rel_path_name.
                 move '' to suffix.
                 call 's$expand_path' using rel_path_name,
                                            suffix,
                                            server_queue_path_name,
                                            error_code.
                 if error_code not = 0
                    move 's$expand_path: server_queue' to
                               where_occurred
                    perform 999_error_routine.

      ***********************************************************************
      *******     Add the program epilogue handler.  It is executed after all
      *******     task epilogue handlers.
      ***********************************************************************
                 move  !null() to null_pointer.
                 move 'program_epilogue_handler' to entry_name.
                 call 's$find_entry' using null_pointer,
                                           entry_name,
                                           entry_value,
                                           error_code.
                 if error_code not = 0
                     move 'error from s$find_entry - prog epilogue'
                                          to where_occurred
                     perform 999_error_routine.

                 call 's$add_epilogue_handler' using entry_value,
                                                     error_code.
                 if error_code not = 0
                     move 'error from s$add_epilogue_handler'
                                              to where_occurred
                     perform 999_error_routine.
      *****************************************************************
      ******* MT1. Call s$monitor.  Task 1 becomes the monitor task.
      *****************************************************************
                 perform 100_start_monitor until stop_flag = 'y'.

             100_start_monitor.
                 move 'MONITOR:' to prompt_string.
                 move TRUE to abbrev_switch.
                 call 's$monitor' using caller,
                                        prompt_string,
                                        abbrev_switch,
                                        error_code.
                 if error_code not = 0
                     move 'error from s$monitor' to where_occurred
                     perform 999_error_routine.
```

```
*****************************************************************
******* MT2. Respond to the quit monitor task request, which causes
*******      s$monitor to return with a zero error code.  Offer a chance
*******      to retract the quit request.  If the monitor task user
*******      wants to stop the application, move 'y' to stop_flag.
*******      This is a shared variable.  Other tasks will read this
*******      value and stop themselves.  Stop task 1.
*****************************************************************
           display 'Do you really want to stop the process? (y or n)'.
           accept answer.
           if answer = 'y'
               move 'y' to stop_flag
               display 'calling stop task'
               move CURRENT_TASK to task_id
               move STOP_TASK to action_code
               call 's$control_task' using task_id,
                                            action_code,
                                            error_code
               if error_code not = 0
                   move 'error from s$control_task' to where_occurred
                   perform 900_task_error_routine.

       menu_entry.
************************************************************************
******* T5.  Specify a common entry point for static user tasks.
************************************************************************
       entry 'menu_entry'.
************************************************************************
******* T6.  Prepare to use s$set_process_terminal later in the task.
************************************************************************
           move 1 to terminal_switch.
*****************************************************************
******* T7.  Add a task epilogue handler.
*****************************************************************
           move function null() to null_pointer.
           move 'epilogue_for_tasks' to entry_name.
           call 's$find_entry' using null_pointer,
                                      entry_name,
                                      entry_value,
                                      error_code.
           if error_code not = 0
               move 'error from s$find_entry' to where_occurred
               perform 900_task_error_routine.
           call 's$add_task_epilogue_handler' using
                                      entry_value,
                                      error_code.
           if error_code not = 0
               move 'error from s$add_task_epilogue_handler'
                                           to where_occurred
               perform 900_task_error_routine.
```

```
      **********************************************************************
      ******* Q1.  Attach a port to the two-way server queue and open the port.
      **********************************************************************
      ******* T8.  Get the task's task ID.  This value is used at various places
      *******     in the program to identify the running task. Form a queue port
      *******     name containing the task ID.  Each task attaches a unique port
      *******     to the queue.  All bits in duration switches are set to zero.
      **********************************************************************
                 call 's$get_task_id' using task_id.
                 move task_id_alpha to task of task_port_name.
                 move task_port_name to port_name.
                 move 0 to duration_switches.
                 call 's$attach_port' using port_name,
                                            server_queue_path_name,
                                            duration_switches,
                                            server_queue_port_id,
                                            error_code.

                 if error_code not = 0
                     move 'error from s$attach_port' to where_occurred
                     perform 900_task_error_routine.

                 move REQUESTER_IO_TYPE to io_type.
                 call 's$msg_open' using server_queue_port_id,
                                         io_type,
                                         error_code.
                 if error_code not = 0
                     move 'error from s$msg_open' to where_occurred
                     perform 900_task_error_routine.
      **********************************************************************
      *******      Initialize screens and save formats in the user heap.
      **********************************************************************
                 perform screen initialization 'menu' into (menu_record)
                                     with formid (menu_form_id)
                                          status (error_code).
                 if error_code not = 0
                     move 'error from menu screen init' to where_occurred
                     perform 900_task_error_routine.

                 perform screen save with formid (menu_form_id)
                                          status (error_code).
                 if error_code not = 0
                     move 'error from menu screen save' to where_occurred
                     perform 900_task_error_routine.

                 perform screen initialization 'bid_sub' into (bid_record)
                                      with formid (bid_form_id)
                                           status (error_code).
                 if error_code not = 0
                     move 'error from bid screen init' to where_occurred
                     perform 900_task_error_routine.
```

```
           perform screen save with formid (bid_form_id)
                                    status (error_code).
           if error_code not = 0
               move 'error from bid screen save' to where_occurred
               perform 900_task_error_routine.
************************************************************************
******* T9.  Display the menu form and accept the user's input.
************************************************************************
           move '' to form_reply.
           perform 300_display_menu thru exit_display_menu
               with test before until stop_flag = 'y'.
           perform 950_stop_task.

       300_display_menu.
           perform screen input 'menu'  update (menu_record) with
                                        formid (menu_form_id)
                                        timeout (time_period)
                                        message (form_reply)
                                        status (error_code).
           if error_code not = 0
              if error_code = e$timeout
                 go to exit_display_menu
              else
                 move 'error from menu screen input' to where_occurred
                 perform 900_task_error_routine.
           if choice = 1
                move 'Function 1 is not operational.  Choose Number 3.'
                        to form_reply
             else
               if choice = 2
               move 'Function 2 is not operational.  Choose Number 3.'
                        to form_reply
              else
                 if choice = 3
                    move spaces to next_action of bid_record
                    move '' to form_reply
                    perform 400_display_bid_form with test before
                       until (stop_flag = 'y' or
                              next_action of bid_record = 'q').
       exit_display_menu.  exit.

       400_display_bid_form.
*****************************************************************
******* T10. Stop the task if the monitor task has initiated the
*******      stop procedure.  Otherwise, display the bid form and
*******      accept the user's input.
*****************************************************************
           move spaces to next_action of bid_record.
           perform screen input 'bid_sub' update (bid_record)
                                    with formid (bid_form_id)
                                         message (form_reply)
                                          status (error_code).
           if error_code not = 0
               move 'error from bid screen input' to where_occurred
               perform 900_task_error_routine.

           if next_action of bid_record = 'i' or 'b'
              perform 500_process_request
              else if next_action of bid_record = 'n'
                 move '' to bid_record,
                            form_reply
              else if next_action of bid_record  = 'q'
                 move '' to form_reply
              else
                 move 'invalid action code' to form_reply.


       500_process_request.
************************************************************************
******* Q2.  Assign a high priority (17) to messages containing bids
*******      greater than 1000.  Other messages have the lower priority
*******     of 10.  The s$msg_send subroutine will return an error message
*******      if no server program is active when a message is sent.  The
*******      highest priority (19) is saved for future use.
***********************************************************************
           if next_action of bid_record = 'b' &
                      your_bid of bid_record  > 1000
              move HIGH_PRIORITY to msg_priority
           else
              move LOW_PRIORITY to msg_priority.
************************************************************************
******* Q3.  Send a message to the queue.
************************************************************************
           move 19 to msg_priority.
           move MESSAGE_SIZE to msg_length_in.
          move next_action of bid_record to next_action of message_info.
           move your_bidder_id of bid_record
                        to your_bidder_id of message_info.
           move item_no of bid_record to item_no of message_info.
           if next_action of bid_record = 'b'
                move your_bid of bid_record to your_bid of message_info.
           call 's$msg_send' using server_queue_port_id,
                                   msg_priority,
                                   msg_subject,
                                   msg_length_in,
                                   message_info,
                                   msg_id,
                                   error_code.
           if error_code not = 0
              move 'error from s$msg_send' to where_occurred
              perform 900_task_error_routine.
```

```
*****************************************************************
******* Q4.  Receive a reply from the queue.
*****************************************************************
           move REPLY_SIZE to reply_length_in.
           call 's$msg_receive_reply' using server_queue_port_id,
                                            msg_id,
                                            reply_length_in,
                                            reply_length_out,
                                            message_reply,
                                            error_code.
           if error_code not = 0
               move 'error from s$msg_receive_reply' to where_occurred
               perform 900_task_error_routine.

************************************************************************
******* T11. Redisplay the form with information obtained from the reply.
*******     User error messages are displayed on the last line of the form.
************************************************************************
           move description of message_reply to
                        description of bid_record.
           move high_bid of message_reply to high_bid of bid_record.
           if error_message of message_reply = spaces
               move 'Enter your bid now' to form_reply
           else
            move error_message of message_reply to form_reply.
           if next_action of bid_record = 'i'
               move 0 to your_bid of bid_record.
           move spaces to next_action of bid_record.

       900_task_error_routine.
************************************************************************
******* T12. Handle task errors.  If an error occurs in a task, the task
*******      disables tasking and displays its error message on the
*******      process terminal.  The task then reenables tasking and stops
*******      itself.  Other tasks continue to run.
************************************************************************
           call 's$set_process_terminal' using terminal_switch.
           move task_id_alpha to task of task_id_msg.
           call 's$error' using error_code, task_id_msg, where_occurred.
           call 's$set_process_terminal' using terminal_switch.
           perform 950_stop_task.

       950_stop_task.
************************************************************************
******* T13. Stop the current task.  When a task stops, the task epilogue
*******      handler runs.
************************************************************************
           move CURRENT_TASK to task_id.
           move STOP_TASK_CLEANUP to action_code.
           call 's$control_task' using task_id,
                                       action_code,
                                       error_code.
               if error_code not = 0
                   move 'error from s$control_task' to where_occurred
                   perform 999_error_routine.

           999_error_routine.
      ************************************************************************
      *******      End the program (all tasks) on miscellaneous errors.
      ************************************************************************
               call 's$error' using error_code, caller, where_occurred.
               stop run.

           entry 'epilogue_for_tasks'.
      ************************************************************************
      ******* T14. Specify the task epilogue entry point.  The task epilogue
      *******      handler closes the queue for each task.
      ************************************************************************
               display 'entering task epilogue'.
               call 's$close' using server_queue_port_id,
                                    error_code.
               exit program.


           entry 'program_epilogue_handler'.
      ************************************************************************
      ******* T15. Specify the program epilogue entry point.
      *******     When all tasks are stopped, the program epilogue handler runs.
      *******      The operating system restarts task 1 to execute the program
      *******      epilogue handler.
      ************************************************************************
      ************************************************************************
      ******* Q5.  The program epilogue handler attaches a port to
      *******      the server queue and opens the port.  All bits in
      *******      duration_switches are set to zero.
      ************************************************************************
               call 's$set_process_terminal' using terminal_switch.
               display 'entering program epilogue'.
               move 0 to duration_switches.
               move 'cleanup_port' to port_name.
               call 's$attach_port' using port_name,
                                          server_queue_path_name,
                                          duration_switches,
                                          server_queue_port_id,
                                          error_code.
               if error_code not = 0
                   move 'error from s$attach_port in program epilogue'
                             to where_occurred
                   perform 999_error_routine.
               move REQUESTER_IO_TYPE to io_type.
               call 's$msg_open' using server_queue_port_id,
                                              io_type,
                                              error_code.
               if error_code not = 0
                   move 'error from s$msg_open in program epilogue'
                              to where_occurred
                   perform 999_error_routine.

    ************************************************************************
    ******* Q6.  Send a message to the queue.  The program epilogue handler
    *******      tells the server program to stop itself.
    ************************************************************************
               move 000000 to item_no of message_info.
               move 'MONITOR SAYS STOP' to msg_subject.
               move MESSAGE_SIZE to msg_length_in.
               call 's$msg_send' using server_queue_port_id,
                                       msg_priority,
                                       msg_subject,
                                       msg_length_in,
                                       message_info,
                                       msg_id,
                                       error_code.
               if error_code not = 0
                   move 'error from s$msg_send in prog epilogue'
                                                   to where_occurred
                   perform 999_error_routine.

    *****************************************************************
    ******* Q7.  Receive a reply from the queue.  When the program
    ******* epilogue handler receives the reply from the server
    ******* program, this program ends.
    *****************************************************************
               move REPLY_SIZE to reply_length_in.
               call 's$msg_receive_reply' using server_queue_port_id,
                                                msg_id,
                                                reply_length_in,
                                                reply_length_out,
                                                message_reply,
                                                error_code.
               if error_code not = 0
                   move 'error from s$msg_receive_reply' to where_occurred
                   perform 999_error_routine
               else
                   display error_message of message_reply.
    *****************************************************************
    *******      The operating system closes and detaches ports
    *******      because the hold_attached and hold_open bits were
    *******      set to 0 when s$attach_port was called.
    *****************************************************************
               exit program.
```

Server Program — C Version

```
/*********************************************************************
******* This is the server program in a requester/server application.
******* It reads messages from a two-way server queue, services the
******* messages, and sends back replies.  Two types of messages are
******* serviced.  One type is a simple request for information about an
******* auction item.  To service this type of request, the program reads a
******* transaction file to obtain the information and sends the
******* information to the requester in the reply to the message.
******* The other type of message serviced by this program is bid
******* submissions.  For these types of messages, the program
******* validates the bid submission, and, if the request is valid, the
******* program updates two transaction files.  The program tells the
******* requester whether the bid was accepted in the reply to the message.
*******
******* If the bid was not accepted, the reply is an appropriate error
******* message.
*******
******* This program stops when a stop message is received from the
******* monitor task in the requester program.
*******
******* This program must be executed in a privileged process.  It can
******* be executed in a background process.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  This program starts two types
******* of transactions - one is a request for information and the other is
******* a bid submission.  Suppose that a user in a task asks for
******* information about an item (one transaction) and then submits a bid
******* on the item (a second transaction).  It is possible that, in
******* between these two transactions, a user in another task gets a bid
******* accepted on the same item.  This would make the information that
******* the first user received as a result of the first transaction
******* out of date.
*******
******* This situation can be avoided if the two types of transactions are
******* combined into a single transaction.  This change could be
******* efficiently implemented by having the requester program start
******* and end the transaction, based on the user's input.  The server
******* program would process information on behalf of the requester's
******* transactions, and this processing would be transaction-protected.
******* The server program can also abort the transaction.
******* See the section called "Transaction Protection and Two-Way Server
******* Queues" in Chapter 5, "Queues," of this manual for a description
******* of this capability.  The section called "Server Queue Scenarios,"
******* also in Chapter 5, illustrates how to use the feature.
*********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
```

```
struct  {
        char_varying(6)    bidder_id;
        char_varying(20)   name;
        int                amt_approved;
        int                amt_spent;
        }        bidder_record;

struct  {
        char_varying(6)    item_no;
        char_varying(20)   description;
        int                high_bid;
        char_varying(6)    last_bidder_id;
        }         item_record;

struct{
      short int         version;
      long int          header_msg_id;
      long int          date_time_queued;
      short int         requester_priority;
      char_varying(32)  msg_subject;
      long int          requester_process_id;
      char_varying(32)  requester_user_name;
      char_varying(32)  requester_user_group;
      long int          server_process_id;
      short int         switches;
      short int         requester_maximum_priority;
      short int         requester_maximum_processes;
      short int         jiffies_queued;
      short int         reserved1[10];
      char_varying(32)  language;
      short int         character_set;
      short int         reserved2[3];
      }           msg_header;

struct msg_reply
      {
      char_varying(20)  description;
      int               high_bid;
      char_varying(70)  error_message;
      }     message_reply;

struct message_info
      {
      char_varying(1)   next_action;    /* 2 len spec + 1 data + 1 word
align*/
     char_varying(6)   your_bidder_id; /* 2 length spec + 6 data bytes
*/
     char_varying(6)   item_no;       /* 2 length spec + 6 data bytes     */
      int             your_bid;       /* 4 data bytes                     */
      }    req_message;              /* length = 24                      */
struct  msg_reply        *PTR_message_reply;
short int                 access_mode;
char                      amt_holder[6];
char_varying(32)          bidder_file_index_name;
char_varying(256)         bidder_file_path_name;
short int                 bidder_port_id;
short int                 buf_length = 38;
short int                 call_debug;
char_varying(32)          caller = "server";
short int                 chase_switch = 0;
char_varying(256)         command_line;
char_varying(256)         config_path;
int                       counter;
short int                 duration_switches;
void                      epilogue_for_program();
short int                 error_code;
short int                 file_organization;
char                      holder[38];
char                     *PTR_to_holder;
short int                 object_type;
char                      io_success;
short int                 io_type;
char_varying(32)          item_file_index_name;
char_varying(256)         item_file_path_name;
short int                 item_port_id;
short int                 locking_mode;
short int                 maximum_record_length;
long int                  msg_id;
long int                  msg_length_in;
long int                  msg_length_out;
short int                 msg_priority;
short int                 msg_selector;
char_varying(32)          port_name;
short int                 protection_switch;
short int                 record_length;
long int                  reply_length_in;
char_varying(256)         rel_path_name;
int                       save_last_bid;
char_varying(6)           save_last_bidder;
char_varying(256)         server_queue_path_name;
short int                 server_queue_port_id;
int                       status;
char                      stop_application_flag;
char_varying(32)          suffix = "";
long int                  time_period;
char                      trans_aborted;
short int                 transaction_priority;
char_varying(32)          where_occurred;
```

```
short int               e$file_in_use =  1084;
short int               e$key_in_use = 2918;
short int               e$object_not_found = 1032;
short int               e$record_not_found = 1112;
short int               e$record_in_use =  2408;
short int               e$tp_aborted  = 2931;

extern void             s$abort_transaction();
extern void             s$add_epilogue_handler();
extern void             s$attach_port();
extern void             s$close();
extern void             s$commit_transaction();
extern void             s$create_file();
extern void             s$delete_file_on_close();
extern void             s$error();
extern void             s$expand_path();
extern void             s$get_object_type();
extern void             s$keyed_read();
extern void             s$keyed_rewrite();
extern void             s$msg_open();
extern void             s$msg_send_reply();
extern void             s$msg_receive();
extern void             s$open();
extern void             s$set_transaction_file();
extern void             s$sleep();
extern void             s$start_transaction();
extern void             s$start_priority_transaction();
extern void             s$stop_program();

void                    epilogue_for_program();
void                    f100_process_new_message();
void                    f200_get_info();
void                    f210_info_transaction();
void                    f300_submit_bid();
void                    f310_file_io_1();
void                    f320_file_io_2();
void                    f330_file_io_3();
void                    f340_file_io_4();
void                    f350_file_io_5();
void                    f900_common_error_code_check();
void                    f950_unexpected_error();
void                    f999_fatal_error();
```

```
#define FIRST_MESSAGE_NON_BUSY     1
#define FIXED                      1
#define HIGH_PRIORITY              9
#define INDEXED                    3
#define MAX_RETRIES                5
#define REQ_MESSAGE_SIZE           24
#define REPLY_SIZE                 96
#define SERVER                     5
#define SERVER_IO_TYPE             6
#define TRUE                       1
#define UPDATE                     4


  main()
/*****************************************************************
*******    Set all bits in duration_switches to zero.  This variable
*******    is used by the s$attach_port subroutine.
*****************************************************************/
  {
     duration_switches = 0;
     version = 1;
/*****************************************************************
******* TP1.  Declare the Bidder file to be a transaction file.
*******       The s$set_transaction_file subroutine must be called
*******       from a privileged process.
*****************************************************************/
     rel_path_name = "bidder_file";
     s$expand_path( &rel_path_name,
                    &suffix,
                    &bidder_file_path_name,
                    &error_code);
     if (error_code != 0)
        {
           where_occurred = "s$expand_path: bidder_file";
           f999_fatal_error();
        }
     protection_switch = TRUE;
     s$set_transaction_file( &bidder_file_path_name,
                             &protection_switch,
                             &error_code);
     if (error_code != 0)
        {
          where_occurred = "error from s$set_trans_file bidder";
          f999_fatal_error();
        }
```

```
/****************************************************************
******* TP2.  Declare the Item file to be a transaction file.
****************************************************************/
      rel_path_name = "item_file";
      s$expand_path( &rel_path_name,
                     &suffix,
                     &item_file_path_name,
                     &error_code);
      if (error_code != 0)
         {
           where_occurred = "s$expand_path: item_file";
           f999_fatal_error();
         }

       s$set_transaction_file( &item_file_path_name,
                               &protection_switch,
                               &error_code);
       if (error_code != 0)
          {
             where_occurred = "error from s$set_trans_file item";
             f999_fatal_error();
          }
/****************************************************************
******* TP3. Attach ports to transaction files and open the files.
****************************************************************/
       port_name = "bidder_port";
       s$attach_port(  &port_name,
                       &bidder_file_path_name,
                       &duration_switches,
                       &bidder_port_id,
                       &error_code);
       if (error_code != 0)
          {
            where_occurred = "error from s$attach_port for bidder file";
             f999_fatal_error();
          }
       port_name = "item_port";
       s$attach_port(  &port_name,
                       &item_file_path_name,
                       &duration_switches,
                       &item_port_id,
                       &error_code);
       if (error_code != 0)
          {
             where_occurred = "error from s$attach_port for item file";
             f999_fatal_error();
          }
```

```
             file_organization = FIXED;
             io_type = UPDATE;
             access_mode = INDEXED;
             bidder_file_index_name = "bidder_id";
             s$open( &bidder_port_id,
                     &file_organization,
                     &buf_length,
                     &io_type,
                     &locking_mode,
                     &access_mode,
                     &bidder_file_index_name,
                     &error_code);
            if (error_code != 0)
               {
                 where_occurred = "error from s$open for bidder file";
                 f999_fatal_error();
               }

            item_file_index_name = "item_id";
            s$open( &item_port_id,
                    &file_organization,
                    &buf_length,
                    &io_type,
                    &locking_mode,
                    &access_mode,
                    &item_file_index_name,
                    &error_code);
            if (error_code != 0)
               {
                 where_occurred = "error from s$open for item file";
                 f999_fatal_error();
               }

      /******************************************************************
      ******* Q1.  Create a two-way server queue if one does not exist.
      *******      The test to see if a queue already exists permits
      *******      multiple server processes to simultaneously execute
      *******      this program.
      ******************************************************************/
            rel_path_name = "2-way_server_queue";
            s$expand_path( &rel_path_name,
                           &suffix,
                           &server_queue_path_name,
                           &error_code);
            if (error_code != 0)
               {
                 where_occurred = "s$expand_path: server_queue";
                 f999_fatal_error();
               }
            s$get_object_type( &server_queue_path_name,
                               &chase_switch,
                               &object_type,
                               &error_code);
            if ((error_code == 0) || (error_code == e$object_not_found))
              ;
            else
              {
                 where_occurred = "error from s$get_object_type";
                 f999_fatal_error();
              }
            if (error_code == e$object_not_found)
              {
               file_organization = SERVER;
               maximum_record_length = 96;
               s$create_file( &server_queue_path_name,
                              &file_organization,
                              &maximum_record_length,
                              &error_code);
               if (error_code != 0)
                  {
                    where_occurred = "error from s$create_file";
                    f999_fatal_error();
                  }
              }

      /*****************************************************************
      ******* Q2.  Attach a port to the queue.
      *****************************************************************/
            port_name = "server_1";
            s$attach_port(  &port_name,
                            &server_queue_path_name,
                            &duration_switches,
                            &server_queue_port_id,
                            &error_code);
            if (error_code != 0)
               {
                 where_occurred = "error from s$attach_port";
                 f999_fatal_error();
               }
      /*****************************************************************
      ******* Q3. Open the port to the queue.
      *****************************************************************/
            io_type = SERVER_IO_TYPE;
            s$msg_open( &server_queue_port_id,
                        &io_type,
                        &error_code);
            if (error_code != 0)
               {
                 where_occurred = "error from s$msg_open";
                 f999_fatal_error();
               }
      /***********************************************************************
      *******     Add program epilogue handler.
      ***********************************************************************/
             s$add_epilogue_handler(&epilogue_for_program,
                                    &error_code);
             if (error_code != 0)
```

```
                {
                   where_occurred = "error from s$add_epilogue_handler";
                   f999_fatal_error();
                }

      /**********************************************************************
      *******    Continuously process new messages from the queue. Stop when the
      *******     monitor task in the requester program sends a message saying
      *******     that the application is stopping.  The requester program
      *******    stops all user tasks before it sends this message, so this is
      *******     the last message in the queue.  The server can stop itself
      *******    immediately and will not leave any user messages unserviced.
      **********************************************************************/
             PTR_message_reply = &message_reply;
             for (stop_application_flag = 'n'; stop_application_flag != 'y';
                  f100_process_new_message());
      /**********************************************************************
      *******    Stop the progam.  The program epilogue handler will execute
      *******    after s$stop_program.
      **********************************************************************/
             error_code = 0;
             s$stop_program( &command_line,
                             &error_code);
        }

       void f100_process_new_message()
      /**********************************************************************
      ******* Q4.  Receive a message from the queue.  Since the port is in wait
      *******     mode, if there are no messages in the queue, the program will
      *******    suspend in s$msg_receive until a message is added to the queue.
      **********************************************************************/
       {
            msg_selector =  FIRST_MESSAGE_NON_BUSY;
            msg_length_in = REQ_MESSAGE_SIZE;
            s$msg_receive( &server_queue_port_id,
                           &msg_selector,
                           &msg_id,
                           &msg_header,
                           &msg_length_in,
                           &msg_length_out,
                           &req_message,
                           &error_code);
            if (error_code != 0)
               {
                  where_occurred = "error from s$msg_receive";
                  f999_fatal_error();
               }
```

```
/*****************************************************************
*******       Initialize reply fields.
*****************************************************************/
         message_reply.description = "";
         message_reply.error_message = "";
         message_reply.high_bid = 0;
/*****************************************************************
*******       Set the stop flag to 'y' if the monitor task in the
*******       requester program sent a stop message.
*****************************************************************/
         if (msg_subject == "MONITOR SAYS STOP")
            {
               message_reply.error_message =
                         "\nreceived stop message -- stopping now\n";
               stop_application_flag = 'y';
            }
         else
            if (req_message.next_action == "i")
                f200_get_info();
            else
               if (req_message.next_action == "b")
                   f300_submit_bid();

/*******************************************************************
******* Q5.  Reply to the message.  The reply contains the
*******      results of servicing the message.
*******************************************************************/
         reply_length_in = REPLY_SIZE;
         s$msg_send_reply( &server_queue_port_id,
                           &msg_id,
                           &reply_length_in,
                           &message_reply,
                           &error_code);
         if (error_code != 0)
            {
              where_occurred = "error from msg_send_reply";
              f999_fatal_error();
            }
 }

 void f200_get_info()
/*****************************************************************
******* TP4. Start a transaction that obtains information from
*******      the Item file.
*****************************************************************/
  {
       s$start_transaction( &error_code);
       if (error_code != 0)
          {
            where_occurred = "error from s$start_trans";
            f950_unexpected_error();
          }
```

```
/*********************************************************************
******* TP5. Perform a transaction that obtains information about the
*******      item number sent in the message.  The transaction
*******      reads a record in the Item file.  It keeps trying
*******      the I/O call until either the transaction is successful,
*******     the transaction is aborted, or five attempts have been made.
*********************************************************************/
      for (trans_aborted = 'n', io_success = 'n', counter = 0,
        time_period = 10; (trans_aborted == 'n') && (io_success == 'n') &&
           (counter < MAX_RETRIES); f210_info_transaction());
/*********************************************************************
******* TP8. If the read was successful, commit the transaction.
*********************************************************************/
        if ((trans_aborted == 'n') && (io_success == 'y'))
           {
              s$commit_transaction( &error_code);
              if (error_code != 0)
                 {
                   where_occurred = "error from s$commit_trans";
                   f950_unexpected_error();
                 }
              else
                 {
                   *strncpy(&message_reply.description, &holder[6], 20);
                   *strncpy(amt_holder, &holder[26], 6);
                   message_reply.high_bid = (short)atoi(amt_holder);
                 }
            }
}

void f210_info_transaction()
/*****************************************************************
******* TP6. Read the Item file.
*****************************************************************/
{
       s$keyed_read( &item_port_id,
                     &item_file_index_name,
                     &req_message.item_no,
                     &buf_length,
                     &record_length,
                      holder,
                     &error_code);
/*********************************************************************
******* TP7. Test for the e$record_not_found error message.  Then test
*******      for other errors related to locking.
*********************************************************************/
       if (error_code != 0)
          {
            if (error_code == e$record_not_found)
            {
                message_reply.error_message =
                       "You entered an invalid item id";
                s$abort_transaction( &error_code);
                if (error_code != 0)
```

```
                  {
                    where_occurred = "error from s$abort_transaction";
                    f950_unexpected_error();
                  }
                else
                  trans_aborted = 'y';
            }
            else
                   f900_common_error_code_check();
         }
      else
         io_success = 'y';
 }

 void f300_submit_bid()
/*********************************************************************
******* TP9.  Start a transaction that updates two transaction files.
*******       Assign a higher priority to transactions with dollar
*******       amounts > $10000.
*********************************************************************/
 {
    if (req_message.your_bid < 10000)
        s$start_transaction( &error_code);
    else
        {
           transaction_priority = HIGH_PRIORITY;
           s$start_priority_transaction( &transaction_priority,
                                            &error_code);
        }
    if (error_code != 0)
       {
          where_occurred = "error from submit bid trans";
          f950_unexpected_error();
       }
/*********************************************************************
******* TP10. Perform the transaction.  Retry each I/O attempt up to five
*******       times.  If the transaction is aborted, do not perform the
*******       rest of the transaction.
*********************************************************************/
     trans_aborted = 'n';

     for (counter = 0, time_period = 10, io_success = 'n';
         (trans_aborted == 'n') && (counter < MAX_RETRIES) &&
         (io_success == 'n'); f310_file_io_1());

     if (trans_aborted == 'n')
        for (counter = 0, time_period = 10, io_success = 'n';
            (trans_aborted == 'n') && (counter < MAX_RETRIES) &&
            (io_success == 'n'); f320_file_io_2());

     if (trans_aborted == 'n')
        for (counter = 0, time_period = 10, io_success = 'n';
            (trans_aborted == 'n') && (counter < MAX_RETRIES) &&
            (io_success == 'n'); f330_file_io_3());
     if (trans_aborted == 'n')
        for (counter = 0, time_period = 10, io_success = 'n';
            (trans_aborted == 'n') && (counter < MAX_RETRIES) &&
            (io_success == 'n'); f340_file_io_4());

     if (trans_aborted == 'n')
        for (counter = 0, time_period = 10, io_success = 'n';
            (trans_aborted == 'n') && (counter < MAX_RETRIES) &&
            (io_success == 'n'); f350_file_io_5());
/*********************************************************************
******* TP11. Attempt to commit the transaction if it has not been aborted.
*******      If the commit is not successful, either ask the user to try
*******      again later or perform the error routine, depending on the
*******      error code.
*********************************************************************/
      if (trans_aborted == 'n')
         {
            s$commit_transaction( &error_code);
            if (error_code != 0)
               {
                  if (error_code == e$tp_aborted)
                      message_reply.error_message =
                      "Your bid cannot be processed - please try later";
                   else
                     {
                       where_occurred = "error from s$commit_trans";
                       f950_unexpected_error();
                     }
               }
              else
                  message_reply.error_message = "Bid accepted";
         }
 } /* end f300_submit_bid */

 void f310_file_io_1()
/*********************************************************************
******* Read the Bidder file and validate information.
*********************************************************************/
 {
     bidder_record.bidder_id = req_message.your_bidder_id;
     s$keyed_read( &bidder_port_id,
                   &bidder_file_index_name,
                   &bidder_record.bidder_id,
                   &buf_length,
                   &record_length,
                    holder,
                   &error_code);
      if (error_code != 0)
         {
            if (error_code == e$record_not_found)
               {
                  message_reply.error_message =
                         "You are not an authorized bidder";
                  s$abort_transaction( &error_code);
                  trans_aborted = 'y';
               }
            else
                  f900_common_error_code_check();
         }
       else
         {
            io_success = 'y';
            *strncpy(amt_holder, &holder[26], 6);
            bidder_record.amt_approved = (short)atoi(amt_holder);
            *strncpy(amt_holder, &holder[32], 6);
            bidder_record.amt_spent = (short)atoi(amt_holder);
            if ((bidder_record.amt_approved) -
                (bidder_record.amt_spent + req_message.your_bid) < 0)
               {
                  message_reply.error_message =
                           "You do not have that much money";
                  s$abort_transaction( &error_code);
                    trans_aborted = 'y';
                }
         }
   if ((trans_aborted == 'y') && (error_code != 0))
      {  where_occurred = "error from s$abort_trans";
         f950_unexpected_error();
      }
}

 void f320_file_io_2()
/****************************************************************
*******        Update the Bidder file.
****************************************************************/
 {
    PTR_to_holder = &holder;
    amt_spent = amt_spent + req_message.your_bid;
    holder[32] = '\0';
    sprintf(PTR_to_holder, "%s%.6d", holder, amt_spent);
    s$keyed_rewrite( &bidder_port_id,
                     &bidder_record.bidder_id,
                     &buf_length,
                      holder,
                     &error_code);
    if (error_code != 0)
         f900_common_error_code_check();
    else
       io_success = 'y';
  }

 void f330_file_io_3()
```

```
/***************************************************************
*******        Read the Item file and validate information.
****************************************************************/
 {  item_record.item_no = req_message.item_no;
    s$keyed_read( &item_port_id,
                 &item_file_index_name,
                 &item_record.item_no,
                 &buf_length,
                 &record_length,
                  holder,
                 &error_code);
    if (error_code != 0)
       {
         if (error_code == e$record_not_found)
            {
               message_reply.error_message =
                         "You entered an invalid item number";
               s$abort_transaction( &error_code);
               trans_aborted = 'y';
             }
          else
               f900_common_error_code_check();
      }
   else
     {
          io_success = 'y';
          *strncpy(item_record.description, &holder[6], 20);
          *strncpy(amt_holder, &holder[26], 6);
          item_record.high_bid = (short)atoi(amt_holder);
          *strncpy(&item_record.last_bidder_id, &holder[32], 6);
          message_reply.description = item_record.description;
          message_reply.high_bid = item_record.high_bid;
          if (req_message.your_bid <= item_record.high_bid)
             {
               message_reply.error_message =
                      "You did not outbid the previous high bid";
               s$abort_transaction( &error_code);
               trans_aborted = 'y';
             }
     }
         if (trans_aborted == 'y' && error_code != 0)
            {  where_occurred = "error from s$abort_trans";
               f950_unexpected_error();
            }
 }

 void f340_file_io_4()
```

```
/****************************************************************
*******        Update the Item file.
****************************************************************/
 {
     save_last_bid = item_record.high_bid;
     save_last_bidder = item_record.last_bidder_id;
     item_record.high_bid = req_message.your_bid;
     item_record.last_bidder_id = req_message.your_bidder_id;
     holder[26] = '\0';
     sprintf(PTR_to_holder, "%s%.6d%v", holder, item_record.high_bid,
             &item_record.last_bidder_id);
     s$keyed_rewrite( &item_port_id,
                      &item_record.item_no,
                      &buf_length,
                       holder,
                      &error_code);
     if (error_code != 0)
           f900_common_error_code_check();
     else
        io_success = 'y';
 }

 void f350_file_io_5()
/************************************************************************
*******       Find the previous high bidder on the item and refund
*******       that bidder's account for the amount of the previous
*******       high bid.
************************************************************************/
 {
    s$keyed_read( &bidder_port_id,
                  &bidder_file_index_name,
                  &save_last_bidder,
                  &buf_length,
                  &record_length,
                   holder,
                  &error_code);
     if (error_code != 0)
        {
/*******************************************************************
*******       Do nothing if the previous bidder is not in the file.
*******************************************************************/
           if (error_code == e$record_not_found)
               io_success = 'y';
           else
                f900_common_error_code_check();
        }
     else
        {
          *strncpy(amt_holder, &holder[32], 6);
          bidder_record.amt_spent = (short)atoi(amt_holder);
          amt_spent = amt_spent - save_last_bid;
          holder[32] = '\0';
          sprintf(PTR_to_holder, "%s%.6d", holder, amt_spent);
```

```
                s$keyed_rewrite( &bidder_port_id,
                                 &save_last_bidder,
                                 &buf_length,
                                  holder,
                                 &error_code);
                if (error_code != 0)
                     f900_common_error_code_check();
                else
                     io_success = 'y';
            }
       }

       void f900_common_error_code_check()
      /*************************************************************************
      ******  TP12. Check for the e$record_in_use, e$key_in_use, and
      ******        e$file_in_use error messages.  Retry the I/O call if TP
      ******        locks were not obtained.  If an I/O call is not successful
      ******        after five attempts, abort the transaction.  The length of
      ******        the sleep time increases with each retry.
      ************************************************************************/
       {
          if ((error_code == e$record_in_use)  ||
               (error_code == e$key_in_use)  ||
               (error_code == e$file_in_use))
             {
                counter += 1;
                time_period += 10;
                s$sleep( &time_period, &error_code);
                if (error_code != 0)
                  {
                    where_occurred = "error from s$sleep";
                    f950_unexpected_error();
                  }
                else
                  ;
             }
           else
             f950_unexpected_error();

           if (counter == MAX_RETRIES)
              {
                 trans_aborted = 'y';
                 message_reply.error_message =
                        "The system is busy - try your bid again";
                 s$abort_transaction( &error_code);
                 if (error_code != 0)
                    {
                       where_occurred = "error from s$abort_transaction";
                       f950_unexpected_error();
                    }
               }
       }

       void f950_unexpected_error()
```

```
/************************************************************************
******* On miscellaneous errors within a transaction, try to clean
******* up the current message.  (Send a reply
******* to the message, record the error in the server.out log, and
******* abort the transaction.) If the abort is not successful,
******* call the fatal error routine.  Otherwise, the server continues
******* processing other messages.
************************************************************************/
 {
     message_reply.error_message =
            "Request not completed.  Program error.  Notify SysAdmin.";
     printf("unexpected abort");
     s$error( &error_code, &caller, &where_occurred);
     s$abort_transaction(&error_code);
     if (error_code != 0)
        {
           where_occurred = "error from unexpected abort trans";
           f999_fatal_error();
        }
     else
        trans_aborted = 'y';
 }



 void f999_fatal_error()
/************************************************************************
*******    Stop the program on serious errors.
************************************************************************/
 {
     s$error( &error_code, &caller, &where_occurred);
     error_code = 0;
     s$stop_program( &command_line, &error_code);
 }

 void  epilogue_for_program()
/************************************************************************
******* Q6. Provide the program epilogue handler.  The program
*******     epilogue handler closes the transaction files and the queue.
************************************************************************/
 {
    printf("entering program epilogue");
    caller = "server program epilogue";
    s$close( &bidder_port_id,
             &error_code);
    if (error_code != 0)
      {
        where_occurred = "bidder file close error";
        s$error( &error_code, &caller, &where_occurred);
      }
    s$close( &item_port_id,
             &error_code);
    if (error_code != 0)
```

```
       {
         where_occurred = "item file close error";
         s$error( &error_code,
                  &caller,
                  &where_occurred);
       }
   s$close( &server_queue_port_id,
            &error_code);
   if (error_code !=0)
       {
         where_occurred = "server queue close error";
         s$error( &error_code,
                  &caller,
                  &where_occurred);
       }
    exit(0);
}
```

Requester Program — C Version

```
/**********************************************************************
******* This is the requester program in a requester/server application.
******* This is a tasking program, with one monitor task and many user
******* tasks.  All user tasks have the same entry point (menu_entry).
*******
******* The user tasks display forms on the users' terminals.  Users enter
******* requests, which are changed into messages and sent to a two-way
******* server queue.  The server program receives the messages, services
******* them, and sends back replies.  The user tasks receive the replies
******* and put the results in the FMS forms displayed on the users'
******* terminals.
*******
******* The monitor task controls when the application stops.  When the
******* monitor task wants to stop, it sets a flag that causes each user
******* task to stop after its current message is completely serviced.
******* Then the monitor task stops itself.  When all tasks are stopped,
******* the program epilogue handler executes.  It sends a message to the
******* server program telling the server program to stop.  When the
******* program epilogue handler receives the reply to this message,
******* this program stops.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  The design described above
******* means that there can only be one requester process and one server
******* process.  To stop multiple servers, the requester could repeatedly
******* send the stop message until the error message
******* e$no_msg_server_for_queue is returned.  In the case of multiple
******* requesters, you would probably not want the entire application to
******* stop because one requester wants to stop.  An alternative is to
******* write an external program that controls application startup and
******* shutdown.
**********************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cmenu.incl.c"
/*
typedef struct
     {
          char              choice[1];
     }
     CMENU_TYPE;
*/
CMENU_TYPE  menu_record;

#include "cbid_sub.incl.c"
/*
typedef struct
     {
          char_varying(6)   your_bidder_id;
          char_varying(6)   item_no;
          char_varying(20)  description;
          int               high_bid;
          int               your_bid;
          char_varying(1)   next_action;
     }
     CBID_SUB_TYPE;
*/
CBID_SUB_TYPE    bid_record;

/***********************************************************************
*******  When figuring message length when the message includes data of
*******  type char_varying, you must include the two bytes in each
*******  char_varying data element that holds the length of the element.
*******  If the message is a structure, each element of the structure is
*******  word-aligned, so you must include bytes used for word alignment
*******  in the total count.
***********************************************************************/
struct
 {
  char_varying(1)      next_action;    /* length = 2 length  + 1 data
                                      byte + 1 byte for word alignment */
  char_varying(6)     your_bidder_id; /* length = 2 length + 6 data bytes
*/
  char_varying(6)     item_no;       /* length = 2 length + 6 data bytes  */
  int                your_bid;       /* length = 4 bytes                 */
 }    message_info;                 /* MESSAGE_SIZE = 24               */

struct
{
  char_varying(20)     description;
  int                  high_bid;
  char_varying(70)     error_message;
}      message_reply;
struct
 {
  char_varying(4)      task;
  short int            task_id;
 }     task_port_name;

short int              abbrev_switch;
short int              action_code;
char                   answer;
short int              bid_form_id;
short int              call_debug;
char_varying(32)       caller;
char_varying(256)      command_line;
char_varying(256)      config_path;
char_varying(32)       date_time;
short int              duration_switches;
short int              error_code;
short int              e$timeout = 1081;
short int              file_organization;
char_varying(50)       form_reply;
short int              io_type;
short int              maximum_length;
short int              maximum_record_length;
short int              menu_form_id;
int                    msg_length_in;
short int              msg_priority;
int                    msg_id;
char_varying(32)       msg_subject;
short int              port_id;
char_varying(32)       port_name;
char_varying(32)       prompt_string;
char_varying(256)      rel_path_name;
int                    reply_length_in;
int                    reply_length_out;
short int              server_queue_port_id;
int                    status;
char_varying(32)       suffix;
char                   task_holder[4];
char                   *PTR_task_holder;
short int              task_id;
short int              terminal_switch;
int                    time_period = 10240;
char_varying(50)       where_occurred;

/***********************************************************************
******* T1.  Specify variables shared by all tasks.
***********************************************************************/
ext_shared char                stop_flag = 'n';
ext_shared char_varying(256)   server_queue_path_name;

/***********************************************************************
*******      Specify FMS screen object modules.
***********************************************************************/
extern void            cmenu();
extern void            cbid_sub();

extern void            s$add_epilogue_handler();
extern void            s$add_task_epilogue_handler();
extern void            s$attach_port();
extern void            s$close();
extern void            s$control_task();
extern void            s$detach_port();
extern void            s$encode_flags();
extern void            s$error();
extern void            s$expand_path();
extern void            s$find_entry();
extern void            s$get_task_id();
extern void            s$init_task_config();
extern void            s$monitor();
extern void            s$msg_open();
extern void            s$msg_send();
extern void            s$msg_receive_reply();
extern void            s$set_process_terminal();
extern void            s$string_date_time();
```

```
void                  menu_entry();
void                  f100_start_monitor();
void                  f300_display_menu();
void                  f400_display_bid_form();
void                  f500_process_request();
void                  f900_task_error_routine();
void                  f950_stop_task();
void                  f999_error_routine();
void                  epilogue_for_tasks();
void                  (*task_epilogue_ptr)();
void                  program_epilogue_handler();

#define CONFIG_FILE        "requester_config"
#define CURRENT_TASK       0
#define FALSE              0
#define HIGH_PRIORITY      17
#define LOW_PRIORITY       10
#define MESSAGE_SIZE       24
#define REQUESTER_IO_TYPE  5
#define REPLY_SIZE         96
#define STOP_TASK          1
#define STOP_TASK_CLEANUP  7
#define TRUE               1


 main()
/************************************************************************
******* T2.  Begin execution at the program's main entry point. This is
*******      the entry for task 1 by default.
*************************************************************************
*************************************************************************
******* T3.  Initialize and start the static tasks that are listed in the
*******      task configuration table.
************************************************************************/
 {
      caller = "primary task";
      rel_path_name =  CONFIG_FILE;
      suffix = ".table";
      s$expand_path( &rel_path_name,
                     &suffix,
                     &config_path,
                     &error_code);
      if (error_code != 0)
         {
            where_occurred = "s$expand_path: requester_config.table";
            f999_error_routine();
         }
      call_debug = FALSE;
      s$init_task_config(&config_path,
                         &call_debug,
                         &error_code);
      if (error_code != 0)
         {
           where_occurred = "error from s$init_task_config";
           f999_error_routine();
         }
/*******************************************************************
******* T4.  Get the shared queue path name (shared by all tasks).
*******      The queue is created in the server program.
*******************************************************************/
      rel_path_name = "2-way_server_queue";
      suffix = "";
      s$expand_path(&rel_path_name,
                    &suffix,
                    &server_queue_path_name,
                    &error_code);
      if (error_code != 0)
         {
           where_occurred = "s$expand_path: server_queue";
           f999_error_routine();
         }

/*******************************************************************
*******     Add the program epilogue handler.  It is executed after all
*******     task epilogue handlers.
*******************************************************************/
      s$add_epilogue_handler( &program_epilogue_handler,
                              &error_code);
      if (error_code != 0)
        {
          where_occurred = "error from s$add_epilogue_handler";
          f999_error_routine();
        }
      f100_start_monitor();
 }

 void f100_start_monitor()
/*****************************************************************
******* MT1. Call s$monitor.  Task 1 becomes the monitor task.
*****************************************************************/
 {
      prompt_string = "MONITOR:";
      abbrev_switch = TRUE;
      s$monitor(&caller,
                &prompt_string,
                &abbrev_switch,
                &error_code);
      if (error_code != 0)
         {
            where_occurred = "error from  s$monitor";
            f999_error_routine();
         }
```

```
/***********************************************************************
******* MT2. Respond to the quit monitor task request, which causes
*******     s$monitor to return with a zero error code.  Offer a chance
*******      to retract the quit request.  If the monitor task user
*******      wants to stop the application,  move 'y' to stop_flag.
*******      This is a shared variable.  Other tasks will read this
*******      value and stop themselves.  Stop task 1.
***********************************************************************/
      printf("Do you really want to stop the process?(y or n)");
      scanf("%s", &answer);
      if (answer != 'y')
         f100_start_monitor();
      else
        {
          stop_flag = 'y';
          printf("calling stop task");
          task_id = CURRENT_TASK;
          action_code = STOP_TASK;
          s$control_task( &task_id,
                          &action_code,
                          &error_code);
          if (error_code != 0)
             {
                where_occurred = "error from s$control_task";
                f900_task_error_routine();
             }
        }
     exit(0);
 }

 void menu_entry()
/***********************************************************************
******* T5.  Specify a common entry point for static user tasks.
***********************************************************************/
/***********************************************************************
******* T6.  Prepare to use s$set_process_terminal later in the task.
***********************************************************************/
 {
      terminal_switch = 1;
/*****************************************************************
******* T7.  Add a task epilogue handler.
*****************************************************************/
      task_epilogue_ptr = &epilogue_for_tasks;
      s$add_task_epilogue_handler(task_epilogue_ptr,
                                  &error_code);
      if (error_code != 0)
         {
           where_occurred = "error from s$add_task_epilogue_handler";
           f900_task_error_routine();
          }
```

```
/**********************************************************************
******* Q1.  Attach a port to the two-way server queue and open the port.
**********************************************************************
******* T8.  Get the task's task ID.  This value is used at various places
*******      in the program to identify the running task. Form a queue port
*******      name containing the task ID.  Each task attaches a unique port
*******      to the queue.  All bits in duration switches are set to zero.
*********************************************************************/
     s$get_task_id(&task_id);
     task_port_name.task = "task";
     task_port_name.task_id = task_id;
     duration_switches = 0;
     s$attach_port(&task_port_name,
                   &server_queue_path_name,
                   &duration_switches,
                   &server_queue_port_id,
                   &error_code);
      if (error_code != 0)
         {
            where_occurred = "error from s$attach_port";
            f900_task_error_routine();
         }

      io_type = REQUESTER_IO_TYPE;
      s$msg_open(&server_queue_port_id,
                 &io_type,
                 &error_code);
      if (error_code != 0)
         {
            where_occurred =  "error from s$msg_open";
            f900_task_error_routine();
         }

/*******************************************************************
****      Initialize screens and save formats in the user heap.
*******************************************************************/
    initialize $screen form(cmenu) into (menu_record)
                         formid (menu_form_id)
                         status (error_code);
    if (error_code != 0)
      {   where_occurred = "error from menu screen init";
         f900_task_error_routine();
      }
     save $screen formid (menu_form_id)
                  status (error_code);
     if (error_code != 0)
        { where_occurred = "error from menu screen save";
          f900_task_error_routine();
        }
     initialize $screen form (cbid_sub) into (bid_record)
                formid (bid_form_id)
                status (error_code);
     if (error_code != 0)
        { where_occurred = "error from bid screen init";
          f900_task_error_routine();
        }
     save $screen formid (bid_form_id)
                  status (error_code);
     if (error_code != 0)
        { where_occurred = "errror in bid screen save";
          f900_task_error_routine();
        }
     form_reply = "";
/*************************************************************************
******* T9.  Display the menu form and accept the user's input.
*************************************************************************/
     while (stop_flag != 'y')
        f300_display_menu();
     f950_stop_task();
}

void f300_display_menu()
{
   input $screen formid (menu_form_id) update (menu_record)
                 message (form_reply)
                 timeout (time_period)
                 status (error_code);
   if (error_code != 0)
       {
          if (error_code == e$timeout)
             return;
          else
           {
              where_occurred =  "error from menu screen input";
              f900_task_error_routine();
           }
       }
    if (choice[0] == '1')
       form_reply = "Function 1 is not operational.  Choose Number 3.";
    else
        if (choice[0] == '2')
          form_reply = "Function 2 is not operational.  Choose Number 3.";
        else
            if (choice[0] == '3')
            {
               bid_record.next_action = "";
               form_reply =  "";
               while (stop_flag != 'y' && bid_record.next_action != "q")
                     f400_display_bid_form();
            }
 }
 void  f400_display_bid_form()
/*****************************************************************
******* T10. Stop the task if the monitor task has initiated the
*******      stop procedure.  Otherwise, display the bid form and
*******      accept the user's input.
*****************************************************************/
 {
    bid_record.next_action = "";
    input $screen formid(bid_form_id) update (bid_record)
                                  message (form_reply)
                                  status (error_code);
    if (error_code != 0)
       {
         where_occurred = "error from bid screen input";
         f900_task_error_routine();
       }
    if ((bid_record.next_action == "i") ||
        (bid_record.next_action == "b"))
         f500_process_request();
    else if (bid_record.next_action == "n")
          {
             bid_record.your_bidder_id = "";
             bid_record.item_no = "";
             bid_record.description = "";
             bid_record.high_bid = 0;
             bid_record.your_bid = 0;
             bid_record.next_action = "";
             form_reply = "";
          }
    else if (bid_record.next_action == "q")
            form_reply = "";
         else
           form_reply = "invalid action code";
}

void  f500_process_request()
/***********************************************************************
******* Q2.  Assign a high priority (17) to messages containing bids
*******      greater than 1000.  Other messages have the lower priority
*******     of 10.  The s$msg_send subroutine will return an error message
*******      if no server program is active when a message is sent.  The
*******      highest priority (19) is saved for future use.
***********************************************************************/
{
   if ((bid_record.next_action == "b") && (bid_record.your_bid  > 1000))
          msg_priority =  HIGH_PRIORITY;
   else
          msg_priority = LOW_PRIORITY;
/***********************************************************************
******* Q3.  Send a message to the queue.
***********************************************************************/
   msg_length_in = MESSAGE_SIZE;
   message_info.next_action = bid_record.next_action;
   message_info.your_bidder_id =  bid_record.your_bidder_id;
   message_info.item_no =  bid_record.item_no;
   if (bid_record.next_action == "b")
         message_info.your_bid = bid_record.your_bid;
   s$msg_send(&server_queue_port_id,
              &msg_priority,
              &msg_subject,
              &msg_length_in,
              &message_info,
              &msg_id,
              &error_code);

     if (error_code != 0)
        {
          where_occurred = "error from msg_send";
          f900_task_error_routine();
        }

/*****************************************************************
******* Q4.  Receive a reply from the queue.
*****************************************************************/
     reply_length_in = REPLY_SIZE;
     s$msg_receive_reply(&server_queue_port_id,
                         &msg_id,
                         &reply_length_in,
                         &reply_length_out,
                         &message_reply,
                         &error_code);
     if (error_code != 0)
        {
          where_occurred = "error from s$msg_receive_reply";
          f900_task_error_routine();
        }
/***********************************************************************
******* T11. Redisplay the form with information obtained from the reply.
*******     User error messages are displayed on the last line of the form.
***********************************************************************/
     bid_record.description = message_reply.description;
     bid_record.high_bid = message_reply.high_bid;
     if (message_reply.error_message == "")
        form_reply =  "Enter your bid now";
     else
        form_reply = message_reply.error_message;
     if (bid_record.next_action == "i")
         bid_record.your_bid = 0;
     bid_record.next_action = "";
}

void  f900_task_error_routine()
```

```
/**********************************************************************
******* T12. Handle task errors.  If an error occurs in a task, the task
*******      disables tasking and displays its error message on the
*******     process terminal.  The task then reenables tasking and stops
*******      itself.  Other tasks continue to run.
**********************************************************************/
 {
     s$set_process_terminal(&terminal_switch);
     s$get_task_id(&task_id);
     PTR_task_holder = &task_holder;
     sprintf(PTR_task_holder, "%d", task_id);
     caller = "error from requester task #  ";
     *strcat(caller, task_holder);
     s$error( &error_code,
              &caller,
              &where_occurred);
     s$set_process_terminal(&terminal_switch);
     f950_stop_task();
}
void  f950_stop_task()
/**********************************************************************
******* T13. Stop the current task.  When a task stops, the task epilogue
*******      handler runs.
**********************************************************************/
{
     task_id = CURRENT_TASK;
     action_code = STOP_TASK_CLEANUP;
     s$control_task( &task_id,
                     &action_code,
                     &error_code);
     if (error_code != 0)
       {
        where_occurred =  "error from s$control_task";
        f999_error_routine();
       }
     exit(0);
}
void f999_error_routine()
/**********************************************************************
*******      End the program (all tasks) on miscellaneous errors.
**********************************************************************/
{
     s$error(&error_code, &caller, &where_occurred);
     exit(0);
 }
```

```
void  epilogue_for_tasks()
/***********************************************************************
******* T14. Specify the task epilogue entry point.  The task epilogue
*******      handler closes the queue for each task.
***********************************************************************/
  {
     printf("Sorry - application stopping now");
     s$close( &server_queue_port_id,
              &error_code);
     s$detach_port( &server_queue_port_id,
                    &error_code);
     exit(0);
  }

void   program_epilogue_handler()
/***********************************************************************
******* T15. Specify the program epilogue entry point.
*******     When all tasks are stopped, the program epilogue handler runs.
*******      The operating system restarts task 1 to execute the program
*******      epilogue handler.
************************************************************************
************************************************************************
******* Q5.  The program epilogue handler attaches a port to
*******      the server queue and opens the port.
***********************************************************************/
 {
     s$set_process_terminal(&terminal_switch);
     printf("\nentering program epilogue");
     duration_switches = 0;
     port_name = "cleanup_port";
     s$attach_port(&port_name,
                   &server_queue_path_name,
                   &duration_switches,
                   &server_queue_port_id,
                   &error_code);
     if (error_code != 0)
        {
           where_occurred = "error from s$attach_port in prog epilogue";
           f999_error_routine();
        }
     io_type = REQUESTER_IO_TYPE;
     s$msg_open(&server_queue_port_id,
                &io_type,
                &error_code);
     if (error_code != 0)
        {
           where_occurred =  "error from s$msg_open in prog epilogue";
           f999_error_routine();
        }
/***********************************************************************
******* Q6.  Send a message to the queue.  The program epilogue handler
*******      tells the server program to stop itself.
***********************************************************************/
      message_info.item_no = "000000";
      msg_priority = 19;
      msg_subject = "MONITOR SAYS STOP";
      msg_length_in = MESSAGE_SIZE;
      s$msg_send(&server_queue_port_id,
                 &msg_priority,
                 &msg_subject,
                 &msg_length_in,
                 &message_info,
                 &msg_id,
                 &error_code);
      if (error_code != 0)
         {   where_occurred = "error from s$msg_send in prog epilogue";
              f999_error_routine();
          }
/*****************************************************************
******* Q7.  Receive a reply from the queue.  When the program
*******      epilogue handler receives the reply from the server
*******      program, this program ends.
*****************************************************************/
      reply_length_in = REPLY_SIZE;
      s$msg_receive_reply(&server_queue_port_id,
                          &msg_id,
                          &reply_length_in,
                          &reply_length_out,
                          &message_reply,
                          &error_code);
      if (error_code != 0)
        { where_occurred = "error from s$msg_receive_reply";
          f999_error_routine();
        }
      else
          printf("%v", &message_reply.error_message);
 }
/*****************************************************************
*******      The operating system closes and detaches ports
*******      because the hold_attached and hold_open bits were
*******      set to 0 when s$attach_port was called.
*****************************************************************/
```

Server Program — PL/I Version

```
server:   procedure;
/***********************************************************************
******* This is the server program in a requester/server application.
******* It reads messages from a two-way server queue, services the
******* messages, and sends back replies.  Two types of messages are
******* serviced.  One type is a simple request for information about an
******* auction item.  To service this type of request, the program reads a
******* transaction file to obtain the information and sends the
******* information to the requester in the reply to the message.
******* The other type of message serviced by this program is bid
******* submissions.  For these types of messages, the program
******* validates the bid submission, and, if the request is valid, the
******* program updates two transaction files.  The program tells the
******* requester whether the bid was accepted in the reply to the message.
*******
******* If the bid was not accepted, the reply is an appropriate error
******* message.
*******
******* This program stops when a stop message is received from the
******* monitor task in the requester program.
*******
******* This program must be executed in a privileged process.  It can
******* be executed in a background process.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  This program starts two types
******* of transactions - one is a request for information and the other is
******* a bid submission.  Suppose that a user in a task asks for
******* information about an item (one transaction) and then submits a bid
******* on the item (a second transaction).  It is possible that, in
******* between these two transactions, a user in another task gets a bid
******* accepted on the same item.  This would make the information that
******* the first user received as a result of the first transaction
******* out of date.
*******
******* This situation can be avoided if the two types of transactions are
******* combined into a single transaction.  This change could be
******* efficiently implemented by having the requester program start
******* and end the transaction, based on the user's input.  The server
******* program would process information on behalf of the requester's
******* transactions, and this processing would be transaction-protected.
******* The server program can also abort the transaction.
******* See the section called "Transaction Protection and Two-Way Server
******* Queues" in Chapter 5, "Queues," of this manual for a description
******* of this capability.  The section called "Server Queue Scenarios,"
******* also in Chapter 5, illustrates how to use the feature.
**********************************************************************/
```

```
declare bidder_file                  file;
declare item_file                    file;

declare 1  bidder_record ,
           2  bidder_id              char (6) ,
           2  name                   char (20),
           2  amt_approved           picture '999999',
           2  amt_spent              picture '999999';

declare 1  item_record ,
           2  item_no                char (6) ,
           2  description            char (20),
           2  high_bid               picture '999999' ,
           2  last_bidder_id         char (6) ;

declare  bidder_buffer               char (38) defined (bidder_record);
declare  bidder_buffer_length        fixed bin (15);
declare  bidder_file_path_name       char (256) varying;
declare  bidder_port_id              fixed bin (15) external static;
declare  caller                      char (32) varying;
declare  command_line                char (256) varying;
declare  counter                     fixed bin (15);
declare  entry_value                 entry variable;
declare  error_code                  fixed bin (15);
declare  e$file_in_use             fixed bin (15) external;    /* (1084)*/
declare  e$key_in_use              fixed bin (15) external;    /* (2918)*/
declare  e$object_not_found        fixed bin (15) external;    /* (1032)*/
declare  e$record_not_found        fixed bin (15) external;    /* (1112)*/
declare  e$record_in_use           fixed bin (15) external;    /* (2408)*/
declare  e$tp_aborted              fixed bin (15) external;    /* (2931)*/
declare  io_success                  char (1);
declare  item_buffer                 char(38) defined(item_record);
declare  item_buffer_length          fixed bin (15);
declare  item_file_path_name         char (256) varying;
declare  item_port_id                fixed bin (15) external static;
declare  maximum_record_length       fixed bin (15);
declare  1 message_reply ,
           2  description            char (20) ,
           2  high_bid               picture '999999',
           2  error_message          char (70) varying;

declare  1 msg_header ,
           2  version                fixed bin (15),
           2  header_msg_id          fixed bin (31),
           2  date_time_queued       fixed bin (31),
           2  requester_priority     fixed bin (15),
           2  msg_subject            char (32) varying,
           2  requester_process_id   fixed bin (31),
           2  requester_user_name    char (32) varying,
           2  requester_user_group   char (32) varying,
           2  server_process_id      fixed bin (31),
           2  switches               fixed bin (15),
           2  requester_maximum_priority  fixed bin (15),
           2  requester_maximum_processes fixed bin (15),
           2  jiffies_queued        fixed bin (15),
           2  reserved1             (10) fixed bin (15),
           2  language              char (32) varying,
           2  character_set         fixed bin (15),
           2  reserved2             (3) binary(15);
declare  msg_id                       fixed bin (31);
declare  msg_length_in                fixed bin (31);
declare  msg_length_out               fixed bin (31);
declare  1 req_message ,
           2  next_action           char (1),
           2  your_bidder_id        char (6),
           2  item_no               char (6),
           2  your_bid              picture '999999';
declare  record_length                fixed bin (15);
declare  reply_length_in              fixed bin (31);
declare  save_last_bid                picture '999999';
declare  save_last_bidder             char (6) ;
declare  server_queue_path_name       char (256) varying;
declare  server_queue_port_id         fixed bin (15) external static;
declare  stop_application_flag        char (1);
declare  time_period                  fixed bin (31);
declare  trans_aborted                char (1);

%replace FIRST_MESSAGE_NON_BUSY by       1;
%replace FIXED by                        1;
%replace HIGH_PRIORITY by                9;
%replace IMPLICIT by                     4;
%replace INDEXED by                      3;
%replace MAX_RETRIES by                  5;
%replace SERVER by                       5;
%replace SERVER_IO_TYPE by               6;
%replace TRUE by                         1;
%replace UPDATE by                       4;


declare s$abort_transaction entry(fixed bin(15));
declare s$add_epilogue_handler entry(entry, fixed bin(15));
declare s$attach_port entry(char(32)varying, char(256)varying,
    fixed bin(15), fixed bin(15), fixed bin(15));
declare s$close entry(fixed bin(15), fixed bin(15));
declare s$commit_transaction entry(fixed bin(15));
declare s$create_file entry(char(256)varying, fixed bin(15), fixed
bin(15),
    fixed bin(15));
declare s$delete_file_on_close entry(char(256)varying, fixed bin(15));
declare s$error entry(fixed bin(15), char(32)varying, char(*)varying);
declare s$expand_path entry(char(256)varying, char(32)varying,
    char(256)varying, fixed bin(15));
declare s$find_entry entry(pointer, char(32) varying, entry, fixed
bin(15));
declare s$get_object_type entry(char(256) varying, fixed bin(15),
    fixed bin(15), fixed bin(15));
```

```
declare s$keyed_read entry(fixed bin(15), char(32)varying,
    char(64)varying, fixed bin(15), fixed bin(15), char(*),
    fixed bin(15));
declare s$keyed_rewrite entry(fixed bin(15), char(64)varying,
    fixed bin(15), char(*), fixed bin(15));
declare s$msg_open entry(fixed bin(15), fixed bin(15), fixed bin(15));
declare s$msg_send_reply entry(fixed bin(15), fixed bin(31),
    fixed bin(31), 1 like message_reply, fixed bin(15));
declare s$msg_receive entry(fixed bin(15), fixed bin(15),
    fixed bin(31), 1 like msg_header, fixed bin(31), fixed bin(31),
    1 like req_message, fixed bin(15));
declare s$open entry(fixed bin(15), fixed bin(15), fixed bin(15),
    fixed bin(15), fixed bin(15), fixed bin(15), char(32)varying,
    fixed bin(15));
declare s$set_transaction_file entry(char(256) varying, fixed bin(15),
    fixed bin(15));
declare s$sleep entry(fixed bin(31), fixed bin(15));
declare s$start_transaction entry(fixed bin(15));
declare s$start_priority_transaction entry(fixed bin(15), fixed bin(15));
declare s$stop_program entry(char(256)varying, fixed bin(15));

declare epilogue_for_program               entry variable;


 bidder_buffer_length = 38;
 caller = 'server';
 item_buffer_length = 38;
 msg_header.version = 1;
 msg_length_in = bytesize(req_message);
 reply_length_in = bytesize(message_reply);
/*****************************************************************
******* TP1.  Declare the Bidder file to be a transaction file.
*******       The s$set_transaction_file subroutine must be called
*******       from a privileged process.
*****************************************************************/
 call f600_set_transaction_file_on ('bidder_file',
                                    bidder_file_path_name);
/*****************************************************************
******* TP2.  Declare the Item file to be a transaction file.
*****************************************************************/
 call f600_set_transaction_file_on ('item_file',
                                    item_file_path_name);
/*****************************************************************
******* TP3. Attach ports to transaction files and open the files.
*****************************************************************/
 call f650_attach_and_open ('bidder_port',
                           bidder_file_path_name,
                           bidder_port_id,
                           'bidder_id');
 call f650_attach_and_open ('item_port',
                           item_file_path_name,
                           item_port_id,
                           'item_id');
```

```
/******************************************************************
******* Q1.  Create a two-way server queue if one does not exist.
*******     The test to see if a queue already exists permits
*******     multiple server processes to simultaneously execute
*******     this program.
******************************************************************/
 call s$expand_path('2-way_server_queue',
                    '',
                    server_queue_path_name,
                    error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            's$expand_path: server_queue');

 call s$get_object_type( server_queue_path_name,
                        0,
                        0,
                        error_code);
 if ((error_code ^= 0) & (error_code ^= e$object_not_found))
 then call f999_fatal_error (error_code,
                            'error from s$get_object_type');

 if (error_code = e$object_not_found)
 then do;
      call s$create_file( server_queue_path_name,
                         SERVER,           /*file_organization*/
                         96,               /*maximum_record_length*/
                         error_code);
      if (error_code ^= 0)
      then call f999_fatal_error (error_code,
                                 'error from s$create_file');
 end;
/******************************************************************
******* Q2.  Attach a port to the queue.
******************************************************************/
 call s$attach_port(  'server_1',      /*port_name*/
                     server_queue_path_name,
                     0,                /*duration_switches*/
                     server_queue_port_id,
                     error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            'error from s$attach_port');

/******************************************************************
******* Q3. Open the port to the queue.
******************************************************************/
 call s$msg_open( server_queue_port_id,
                 SERVER_IO_TYPE,      /* io_type*/
                 error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            'error from s$msg_open');
```

```
/**********************************************************************
*******     Add program epilogue handler.
**********************************************************************/
 call s$find_entry ( null(),                /*null_pointer*/
                    'epilogue_for_program', /*entry_name  */
                     entry_value,
                     error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                             'error from s$find_entry');
 call s$add_epilogue_handler( (entry_value),
                               error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                             'error from s$add_epilogue_handler');
/**********************************************************************
*******    Continuously process new messages from the queue. Stop when the
*******     monitor task in the requester program sends a message saying
*******     that the application is stopping.  The requester program
*******    stops all user tasks before it sends this message, so this is
*******     the last message in the queue.  The server can stop itself
*******    immediately and will not leave any user messages unserviced.
**********************************************************************/
 stop_application_flag = 'n';
 do while (stop_application_flag ^= 'y');
    call f100_process_new_message;
 end;
/**********************************************************************
*******    Stop the progam.  The program epilogue handler will execute
*******    after s$stop_program.
**********************************************************************/
 call s$stop_program( command_line,
                      (0));
stop;
/*end server*/

f100_process_new_message: procedure;
/**********************************************************************
******* Q4.  Receive a message from the queue.  Since the port is in wait
*******     mode, if there are no messages in the queue, the program will
*******      suspend in s$msg_receive until a message is added to the
*******      queue.
**********************************************************************/
 call s$msg_receive( server_queue_port_id,
                     FIRST_MESSAGE_NON_BUSY,  /* msg_selector */
                     msg_id,
                     msg_header,
                     msg_length_in,
                     msg_length_out,
                     req_message,
                     error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                             'error from s$msg_receive');
```

```
/*****************************************************************
*******       Initialize reply fields.
*****************************************************************/
 message_reply.description = '';
 message_reply.high_bid = '';
 message_reply.error_message = '';
/*****************************************************************
*******       Set the stop flag to 'y' if the monitor task in the
*******       requester program sent a stop message.
*****************************************************************/
 if (msg_header.msg_subject = 'MONITOR SAYS STOP')
 then do;
      message_reply.error_message =
                      'received stop message -- stopping now';
      stop_application_flag = 'y';
 end;
 else
    if (req_message.next_action = 'i')
    then call f200_get_info;
    else
      if (req_message.next_action = 'b')
      then call f300_submit_bid;

/***********************************************************************
******* Q5.  Reply to the message.  The reply contains the
*******      results of servicing the message.
***********************************************************************/
 call s$msg_send_reply( server_queue_port_id,
                        msg_id,
                        reply_length_in,
                        message_reply,
                        error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                             'error from s$msg_send_reply');

end f100_process_new_message;

f200_get_info: procedure;
/*****************************************************************
******* TP4. Start a transaction that obtains information from
*******      the Item file.
*****************************************************************/
 call s$start_transaction( error_code);
 if (error_code ^= 0)
 then call f950_unexpected_error (error_code,
                                  'error from s$start_trans');
```

```
/**********************************************************************
******* TP5. Perform a transaction that obtains information about the
*******      item number sent in the message.  The transaction
*******      reads a record in the Item file.  It keeps trying
*******      the I/O call until either the transaction is successful,
*******     the transaction is aborted, or five attempts have been made.
**********************************************************************/
 trans_aborted = 'n';
 io_success = 'n';
 counter = 0;
 time_period = 10;
 do while ((trans_aborted = 'n') & (io_success = 'n') &
          (counter < MAX_RETRIES));
        call f210_info_transaction;
      end;
/**********************************************************************
******* TP8. If the read was successful, commit the transaction.
**********************************************************************/
 if ((trans_aborted = 'n') & (io_success = 'y'))
 then do;
      call s$commit_transaction( error_code);
      if (error_code ^= 0)
      then call f950_unexpected_error (error_code,
                                       'error from s$commit_trans');
      else do;
           message_reply.high_bid = item_record.high_bid;
           message_reply.description = item_record.description;
      end;
 end;
end f200_get_info;


f210_info_transaction: procedure;
/****************************************************************
******* TP6. Read the Item file.
****************************************************************/
 call s$keyed_read( item_port_id,
                   'item_id',
                   (req_message.item_no),
                   item_buffer_length,
                   record_length,
                   item_buffer,
                   error_code);
/**********************************************************************
******* TP7. Test for the e$record_not_found error message.  Then test
*******      for other errors related to locking.
**********************************************************************/
 if (error_code ^= 0)
 then do;
      if (error_code = e$record_not_found)
      then do;
           message_reply.error_message =
                         'You entered an invalid item id';
           call s$abort_transaction( error_code);
```

```
                 if (error_code ^= 0)
                 then call f950_unexpected_error (error_code,
                                            'error from s$abort_transaction');
                 else trans_aborted = 'y';
            end;
            else
                 call f900_common_error_code_check (error_code,
                                                  'error from s$keyed_read');
       end;
       else io_success = 'y';
      end f210_info_transaction;


      f300_submit_bid: procedure;
      /************************************************************************
      ******* TP9.  Start a transaction that updates two transaction files.
      *******       Assign a higher priority to transactions with dollar
      *******       amounts > $10000.
      ************************************************************************/
       if (req_message.your_bid < 10000)
       then call s$start_transaction( error_code);
       else call s$start_priority_transaction( HIGH_PRIORITY,
                                               error_code);
       if (error_code ^= 0)
       then call f950_unexpected_error (error_code,
                                     'error from submit bid trans');
      /************************************************************************
      ******* TP10. Perform the transaction.  Retry each I/O attempt up to five
      *******       times.  If the transaction is aborted, do not perform the
      *******       rest of the transaction.
      ************************************************************************/
       trans_aborted = 'n';
       counter = 0;
       time_period = 10;
       io_success = 'n';
       do while ((trans_aborted = 'n') & (counter < MAX_RETRIES) &
                 (io_success = 'n'));
          call f310_file_io_1;
       end;
       if (trans_aborted = 'n')
       then do;
            counter = 0;
            time_period = 10;
            io_success = 'n';
            do while ((trans_aborted = 'n') & (counter < MAX_RETRIES) &
                      (io_success = 'n'));
               call f320_file_io_2;
            end;
       end;
```

```
 if (trans_aborted = 'n')
 then do;
      counter = 0;
      time_period = 10;
      io_success = 'n';
      do while ((trans_aborted = 'n') & (counter < MAX_RETRIES) &
                (io_success = 'n'));
         call f330_file_io_3;
      end;
 end;

if (trans_aborted = 'n')
then do;
     counter = 0;
     time_period = 10;
     io_success = 'n';
     do while ((trans_aborted = 'n') & (counter < MAX_RETRIES) &
               (io_success = 'n'));
        call f340_file_io_4;
     end;
 end;

 if (trans_aborted = 'n')
 then do;
      counter = 0;
      time_period = 10;
      io_success = 'n';
      do while ((trans_aborted = 'n') & (counter < MAX_RETRIES) &
                (io_success = 'n'));
         call f350_file_io_5;
      end;
 end;
/************************************************************************
******* TP11. Attempt to commit the transaction if it has not been aborted.
*******      If the commit is not successful, either ask the user to try
*******      again later or perform the error routine, depending on the
*******      error code.
************************************************************************/
 if (trans_aborted = 'n')
 then do;
      call s$commit_transaction( error_code);
      if (error_code ^= 0)
      then do;
           if (error_code = e$tp_aborted)
           then message_reply.error_message =
                     'Your bid cannot be processed - please try later';
           else call f950_unexpected_error (error_code,
                                       'error from s$commit_trans');

       end;
       else message_reply.error_message = 'Bid accepted';
 end;
end f300_submit_bid;
```

```
f310_file_io_1: procedure;
/************************************************************************
*******       Read the Bidder file and validate information.
************************************************************************/
 bidder_record.bidder_id = req_message.your_bidder_id;
 call s$keyed_read( bidder_port_id,
                    'bidder_id',
                    (bidder_record.bidder_id),
                    bidder_buffer_length,
                    record_length,
                    bidder_buffer,
                    error_code);
 if (error_code ^= 0)
 then do;
     if (error_code = e$record_not_found)
     then do;
          message_reply.error_message =
                    'You are not an authorized bidder';
          call s$abort_transaction( error_code);
          trans_aborted = 'y';
     end;
     else
          call f900_common_error_code_check (error_code,
                                        'error from s$keyed_read');
 end;
 else do;
      io_success = 'y';
      if (bidder_record.amt_approved -
              (bidder_record.amt_spent + req_message.your_bid) < 0)
      then do;
           message_reply.error_message =
                       'You do not have that much money';
           call s$abort_transaction( error_code);
           trans_aborted = 'y';
      end;
 end;
 if (trans_aborted = 'y' & error_code ^= 0)
 then call f950_unexpected_error (error_code,
                                  'error from s$abort_trans');

end f310_file_io_1;

f320_file_io_2: procedure;
/****************************************************************
*******       Update the Bidder file.
****************************************************************/
 bidder_record.amt_spent = bidder_record.amt_spent +
req_message.your_bid;
 call f800_rewrite_bidder_record ((bidder_record.bidder_id));
end f320_file_io_2;
```

```
f330_file_io_3: procedure;
/****************************************************************
*******        Read the Item file and validate information.
****************************************************************/
 item_record.item_no = req_message.item_no;
 call s$keyed_read( item_port_id,
                    'item_id',
                    (item_record.item_no),
                    item_buffer_length,
                    record_length,
                    item_buffer,
                    error_code);
 if (error_code ^= 0)
 then do;
      if (error_code = e$record_not_found)
      then do;
           message_reply.error_message =
                         'You entered an invalid item number';
           call s$abort_transaction( error_code);
           trans_aborted = 'y';
       end;
       else call f900_common_error_code_check (error_code,
                                              'error from s$keyed_read');
  end;
  else do;
       io_success = 'y';
       message_reply.description = item_record.description;
       message_reply.high_bid = item_record.high_bid;
       if (req_message.your_bid <= item_record.high_bid)
       then do;
            message_reply.error_message =
                     'You did not outbid the previous high bid';
            call s$abort_transaction( error_code);
            trans_aborted = 'y';
       end;
  end;
  if (trans_aborted = 'y' & error_code ^= 0)
  then call f950_unexpected_error (error_code,
                                   'error from s$abort_trans');
end f330_file_io_3;


f340_file_io_4: procedure;
/****************************************************************
******* Update the Item file.
****************************************************************/
 save_last_bid = item_record.high_bid;
 save_last_bidder = item_record.last_bidder_id;
 item_record.high_bid = req_message.your_bid;
 item_record.last_bidder_id = req_message.your_bidder_id;
 call s$keyed_rewrite( item_port_id,
                       (item_record.item_no),
                       item_buffer_length,
                      item_buffer,
                      error_code);
 if (error_code ^= 0)
 then call f900_common_error_code_check (error_code,
                                       'error from s$keyed_rewrite');
 else io_success = 'y';
end f340_file_io_4;


f350_file_io_5: procedure;
/*************************************************************************
*******       Find the previous high bidder on the item and refund
*******       that bidder's account for the amount of the previous
*******       high bid.
*************************************************************************/
 call s$keyed_read( bidder_port_id,
                    'bidder_id',
                    (save_last_bidder),
                    bidder_buffer_length,
                    record_length,
                    bidder_buffer,
                    error_code);
 if (error_code ^= 0)
 then do;
/*********************************************************************
*******       Do nothing if the previous bidder is not in the file.
*********************************************************************/
      if (error_code = e$record_not_found)
      then io_success = 'y';
      else call f900_common_error_code_check (error_code,
                                            'error from s$keyed_read');
end;
else do;
     bidder_record.amt_spent = bidder_record.amt_spent -
                                     save_last_bid;
     call f800_rewrite_bidder_record ((save_last_bidder));
 end;
end f350_file_io_5;


f600_set_transaction_file_on: procedure(p_rel_file_name, p_path_name);

 /* parameters */

  declare    p_rel_file_name           char (256) varying,
             p_path_name               char (256) varying;

  /* automatic */

   declare   error_code                fixed bin(15);

 call s$expand_path( p_rel_file_name,
                     '',
                     p_path_name,
                     error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            'call s$expand_path:' || p_rel_file_name);

 call s$set_transaction_file( p_path_name,
                              TRUE,           /*protection_switch*/
                              error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                      'error from s$set_trans_file' || p_rel_file_name);
end f600_set_transaction_file_on;


f650_attach_and_open: procedure(p_port_name, p_path_name, p_port_id,
                                p_index_name);
  /* parameters */

 declare p_port_name           char (32) varying,
         p_path_name           char (256) varying,
         p_port_id             fixed bin (15),
         p_index_name          char (32) varying;

  /* automatic */

 declare error_code            fixed bin (15),
         maximum_length        fixed bin (15);
/*****************************************************************
*******    Set all bits in duration_switches to zero.
*****************************************************************/
 call s$attach_port(  p_port_name,
                      p_path_name,
                      0,             /*duration_switches*/
                      p_port_id,
                      error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            'error from s$attach_port' || p_port_name);

 call s$open( p_port_id,
              FIXED,                /*file_organization */
              maximum_length,
              UPDATE,               /*io_type           */
              IMPLICIT,             /*locking_mode      */
              INDEXED,              /*access_mode       */
              p_index_name,
              error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code, 'error from s$open' ||
p_port_name);
end f650_attach_and_open;
```

```
f800_rewrite_bidder_record: procedure (p_bidder_id);

/*  parameters */

declare   p_bidder_id              char (64) varying in;

/*  execution */

 call s$keyed_rewrite( bidder_port_id, p_bidder_id,
                       bidder_buffer_length, bidder_buffer,
                       error_code);

 if (error_code ^= 0)
 then call f900_common_error_code_check (error_code,
                                         'error from s$keyed_rewrite');
 else io_success = 'y';
 end f800_rewrite_bidder_record;


f900_common_error_code_check: procedure (p_code, p_text);
/*************************************************************************
******  TP12. Check for the e$record_in_use, e$key_in_use, and
******        e$file_in_use error messages.  Retry the I/O call if TP
******        locks were not obtained.  If an I/O call is not successful
******        after five attempts, abort the transaction.  The length of
******        the sleep time increases with each retry.
*************************************************************************/
/* parameters */

declare   p_code              fixed bin (15) in,
          p_text              char (*) var in;

/* automatic */

declare   error_code          fixed bin (15);

/* execution */

 if ((p_code = e$record_in_use)  |
         (p_code = e$key_in_use) |
         (p_code = e$file_in_use))
 then do;
      counter = counter + 1;
      time_period = time_period + 10;
      call s$sleep( time_period, error_code);
      if (error_code ^= 0)
     then call f950_unexpected_error (error_code, 'error from s$sleep');
 end;
else call f950_unexpected_error (p_code, p_text);

if (counter = MAX_RETRIES)
then do;
     trans_aborted = 'y';
     message_reply.error_message =
               'The system is busy - try your bid again';
     call s$abort_transaction( error_code);
     if (error_code ^= 0)
     then call f950_unexpected_error (error_code,
                                   'error from s$abort_transaction');
 end;
end f900_common_error_code_check;


f950_unexpected_error: procedure (p_code, p_text);
/************************************************************************
******* On miscellaneous errors within a transaction, try to clean
******* up the current message.  (Send a reply
******* to the message, record the error in the server.out log, and
******* abort the transaction.) If the abort is not successful,
******* call the fatal error routine.  Otherwise, the server continues
******* processing other messages.
************************************************************************/
/* parameters */

declare   p_code              fixed bin (15) in,
          p_text              char (*) var in;

/* automatic */

declare   error_code          fixed bin (15);

/* execution */

 message_reply.error_message =
           'Request not completed.  Program error.  Notify SysAdmin.';
 put skip list('unexpected abort');
 call s$error( p_code, caller, p_text);
 call s$abort_transaction(error_code);
 if (error_code ^= 0)
 then call f999_fatal_error (error_code,
                            'error from unexpected abort trans');
 else trans_aborted = 'y';

end f950_unexpected_error;


f999_fatal_error: procedure (p_code, p_text);
/************************************************************************
*******
**********    Stop the program on serious errors.
************************************************************************
******/
/* parameters */

declare   p_code              fixed bin (15) in,
          p_text              char (*) var in;

/*  execution */
```

```
 call s$error( p_code, caller, p_text);

 call s$stop_program(command_line, (0));
 stop;
end f999_fatal_error;


epilogue_for_program: entry;
/***********************************************************************
*******
********** Q6. Provide the program epilogue handler.  The program
**********    epilogue handler closes the transaction files and the queue.
************************************************************************
******/
 put skip list('entering program epilogue');
 caller = 'server program epilogue';
 call s$close( bidder_port_id,
              error_code);
 if (error_code ^= 0)
 then call s$error( error_code,
                   caller,
                   'bidder file close error');

 call s$close( item_port_id,
              error_code);
 if (error_code ^= 0)
 then call s$error( error_code,
                   caller,
                   'item file close error');

 call s$close( server_queue_port_id,
              error_code);
 if (error_code ^=0)
 then call s$error( error_code,
                   caller,
                   'server queue close error');

 return;
end server;
```

Requester Program — PL/I Version

```
requester:
      procedure;
/************************************************************************
******* This is the requester program in a requester/server application.
******* This is a tasking program, with one monitor task and many user
******* tasks.  All user tasks have the same entry point (menu_entry).
*******
******* The user tasks display forms on the users' terminals.  Users enter
******* requests, which are changed into messages and sent to a two-way
******* server queue.  The server program receives the messages, services
******* them, and sends back replies.  The user tasks receive the replies
******* and put the results in the FMS forms displayed on the users'
******* terminals.
*******
******* The monitor task controls when the application stops.  When the
******* monitor task wants to stop, it sets a flag that causes each user
******* task to stop after its current message is completely serviced.
******* Then the monitor task stops itself.  When all tasks are stopped,
******* the program epilogue handler executes.  It sends a message to the
******* server program telling the server program to stop.  When the
******* program epilogue handler receives the reply to this message,
******* this program stops.
*******
******* A significant design improvement to this program would be
******* desirable in an actual application.  The design described above
******* means that there can only be one requester process and one server
******* process.  To stop multiple servers, the requester could repeatedly
******* send the stop message until the error message
******* e$no_msg_server_for_queue is returned.  In the case of multiple
******* requesters, you would probably not want the entire application to
******* stop because one requester wants to stop.  An alternative is to
******* write an external program that controls application startup and
******* shutdown.
************************************************************************/
declare  1 menu_record ,
%include 'menu.incl.pl1';;
/*            20 choice              char(1)                /* 07, 69 */


declare  1 bid_record,
%include 'bid_sub.incl.pl1';;
/*          20 your_bidder_id      char(6),              /* 04, 19
            20 item_no             char(6),              /* 11, 05
            20 description         char(20),             /* 11, 15
            20 high_bid            char(6),              /* 11, 42
            20 your_bid            char(6),              /* 11, 60
            20 next_action         char(1)               /* 15, 29
```

```
*/
declare answer                  char (1);
declare bid_form_id             fixed bin(15);
declare caller                  char (32) varying;
declare char_string             char (16);
declare command_line            char (256) varying;
declare config_path             char (256) varying;
declare date_time               char (32) varying;
declare error_code              fixed bin(15) static external;
declare e$timeout               fixed bin(15) external;  /* (1081)*/
declare file_organization       fixed bin(15);
declare form_reply              char (50) varying;
declare maximum_length          fixed bin(15);
declare maximum_record_length   fixed bin(15);
declare menu_form_id            fixed bin(15);
declare  1 message_info ,
            2  next_action          char (1) ,
            2  your_bidder_id       char (6) ,
            2  item_no              char (6) ,
            2  your_bid             picture '999999' ;
 declare  1 message_reply ,
            2  description          char (20),
            2  high_bid             picture '999999',
            2  error_message        char (70) varying;
declare msg_id                  fixed bin(31);
declare msg_length_in           fixed bin(31);
declare msg_subject             char (32) varying;
declare msg_priority            fixed bin(15);
declare port_name               char (32) varying;
declare port_id                 fixed bin(15);
declare reply_length_in         fixed bin(31);
declare reply_length_out        fixed bin(31);
declare server_queue_port_id    fixed bin(15) static external;
declare task_id                 fixed bin(15);
declare terminal_switch         fixed bin(15);
declare time_period             fixed bin(15);

/************************************************************************
******* T1.  Specify variables shared by all tasks.
***********************************************************************/
declare stop_flag              char (1) varying static external shared;
declare server_queue_path_name   char (256) varying static external
shared;

%replace CONFIG_FILE by '<tables>requester_config';
%replace FALSE by 0;
%replace HIGH_PRIORITY by 17;
%replace LOW_PRIORITY by 10;
%replace TRUE by 1;
%replace STOP_TASK by 1;
%replace CURRENT_TASK by 0;
%replace STOP_TASK_CLEANUP by 7;
%replace REQUESTER_IO_TYPE by 5;
```

```
declare s$add_epilogue_handler entry(entry, fixed bin(15));
declare s$add_task_epilogue_handler entry(entry, fixed bin(15));
declare s$attach_port entry(char(32)varying, char(256) varying, fixed
bin(15),
      fixed bin(15), fixed bin(15));
declare s$close entry(fixed bin(15), fixed bin(15));
declare s$control_task entry(fixed bin(15), fixed bin(15), fixed
bin(15));
declare s$detach_port entry(fixed bin(15), fixed bin(15));
declare s$encode_flags entry(char(16), fixed bin(15));
declare s$error entry(fixed bin(15), char(32) varying, char(*) varying);
declare s$expand_path entry(char(256) varying, char(32) varying,
       char(256) varying, fixed bin(15));
declare s$find_entry entry(pointer, char(32) varying, entry, fixed
bin(15));
declare s$get_task_id entry(fixed bin(15));
declare s$init_task_config entry(char(256) varying, fixed bin(15),
        fixed bin(15));
declare s$monitor entry(char(32) varying, char(32) varying, fixed
bin(15),
        fixed bin(15));
declare s$msg_open entry(fixed bin(15), fixed bin(15), fixed bin(15));
declare s$msg_receive_reply entry(fixed bin(15), fixed bin(31), fixed
bin(31),
        fixed bin(31), 1 like message_reply, fixed bin(15));
declare s$msg_send entry(fixed bin(15),fixed bin(15), char(32)varying,
       fixed bin(31), 1 like message_info, fixed bin(31), fixed bin(15));
declare s$set_process_terminal entry(fixed bin(15));
declare s$string_date_time entry(char(32) varying);
declare menu_entry                     entry variable;
declare epilogue_for_tasks             entry variable;
declare program_epilogue               entry variable;
/*   FMS external entries  */
declare bid_sub                        entry;
declare menu                           entry;

/************************************************************************
******* T2.  Begin execution at the program's main entry point. This is
*******      the entry for task 1 by default.
*************************************************************************
 stop_flag = n;
*************************************************************************
******* T3.  Initialize and start the static tasks that are listed in the
*******      task configuration table.
************************************************************************/
 caller = 'primary task';
 call f750_expand_path (CONFIG_FILE,
                       '.table',
                       config_path);

 call s$init_task_config (config_path,
                         FALSE /* debug switch */,
                         error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                              'error from s$init_task_config');
/****************************************************************
******* T4.  Get the shared queue path name (shared by all tasks).
*******      The queue is created in the server program.
****************************************************************/
 call f750_expand_path ('2-way_server_queue',
                        '',
                        server_queue_path_name);
/**********************************************************************
*******     Add the program epilogue handler.  It is executed after all
*******     task epilogue handlers.
**********************************************************************/
 call f700_find_entry ('program_epilogue',
                       s$add_epilogue_handler,
                       f999_error_routine);

 do while (stop_flag ^= 'y');
    call f100_start_monitor;
 end;

 put list ('calling stop task');
 call s$control_task(CURRENT_TASK,
                     STOP_TASK,
                     error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code, 'error from
s$control_task');
return;
/* end requester task 1 (the monitor task) */

 f100_start_monitor: procedure;
/*******************************************************************
********** MT1. Call s$monitor.  Task 1 becomes the monitor task.
*******************************************************************/
  call s$monitor( caller,
                 'MONITOR:',
                  TRUE,
                  error_code);
  if (error_code ^= 0)
  then call f999_error_routine (error_code, 'error from  s$monitor');
/**********************************************************************
******* MT2. Respond to the quit monitor task request, which causes
*******      s$monitor to return with a zero error code.  Offer a chance
*******      to retract the quit request.  If the monitor task user
*******      wants to stop the application,  move 'y' to stop_flag.
*******      This is a shared variable.  Other tasks will read this
*******      value and stop themselves.  Stop task 1.
**********************************************************************/
 put list('Do you really want to stop the process?(y or n)');
 get list(answer);
 if (answer = 'y')
 then stop_flag = 'y';
end f100_start_monitor;
```

```
menu_entry: entry;
/************************************************************************
******* T5.  Specify a common entry point for static user tasks.
************************************************************************/
 time_period = 10240;      /* ten seconds */
/************************************************************************
******* T6.  Prepare to use s$set_process_terminal later in the task.
************************************************************************/
 terminal_switch = 1;
/*****************************************************************
******* T7.  Add a task epilogue handler.
*****************************************************************/
 call f700_find_entry ('epilogue_for_tasks',
                      s$add_task_epilogue_handler,
                      f900_task_error_routine);
/************************************************************************
******* Q1.  Attach a port to the two-way server queue and open the port.
************************************************************************
******* T8.  Get the task's task ID.  This value is used at various places
*******      in the program to identify the running task. Form a queue port
*******      name containing the task ID.  Each task attaches a unique port
*******      to the queue.  All bits in duration switches are set to zero.
************************************************************************/
 call s$get_task_id(task_id);
 port_name = 'task' || character(task_id);
 call f650_open_queue (port_name,
                      server_queue_path_name,
                      server_queue_port_id);
/************************************************************************
*******      Initialize screens and save formats in the user heap.
************************************************************************/
 initialize screen form(menu) into (menu_record)
                   formid (menu_form_id)
                   status (error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code,
                                   'error from menu screen init');

 save screen formid (menu_form_id)
             status (error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code,
                                   'error from menu screen save');

 initialize screen form (bid_sub) into (bid_record)
                   formid (bid_form_id)
                   status (error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code,
                                   'error from bid screen init');

 save screen formid (bid_form_id)
             status (error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code,
                                   'error in bid screen save');

 form_reply = '';
/************************************************************************
******* T9.  Display the menu form and accept the user's input.
************************************************************************/
 do while (stop_flag ^= 'y');
   call f300_display_menu;
 end;
 call f950_stop_task;
return;

/*end menu_entry*/


f300_display_menu: procedure;

 input screen formid (menu_form_id) update (menu_record)
              message (form_reply)
              timeout (time_period)
              status (error_code);
 if (error_code ^= 0)
 then do;
     if (error_code = e$timeout)
     then return;
     else call f900_task_error_routine (error_code,
                                   'error from menu screen input');
 end;
 if (menu_record.choice = '1')
 then form_reply = 'Function 1 is not operational.  Choose Number 3.';
 else do;
      if (menu_record.choice = '2')
     then form_reply = 'Function 2 is not operational.  Choose Number 3.';
      else do;
           if (menu_record.choice = '3')
           then do;
               bid_record.next_action = '';
               form_reply =  '';
             do while (stop_flag ^= 'y' & bid_record.next_action ^= 'q');
                    call f400_display_bid_form;
               end;
          end;
      end;
 end;
end f300_display_menu;
```

```
f400_display_bid_form: procedure;
/*****************************************************************
******* T10. Stop the task if the monitor task has initiated the
*******      stop procedure.  Otherwise, display the bid form and
*******      accept the user's input.
*****************************************************************/
 bid_record.next_action = '';
 input screen formid(bid_form_id) update (bid_record)
                                  message (form_reply)
                                  status (error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code, 'error from bid screen
input');

 if ((bid_record.next_action = 'i') ||
     (bid_record.next_action = 'b'))
 then call f500_process_request;
 else if (bid_record.next_action = 'n')
      then do;
           bid_record.your_bidder_id = '';
           bid_record.item_no = '';
           bid_record.description = '';
           bid_record.high_bid = char (0);
           bid_record.your_bid = char (0);
           bid_record.next_action = '';
           form_reply = '';
       end;
       else if (bid_record.next_action = 'q')
            then form_reply = '';
            else form_reply = 'invalid action code';
end f400_display_bid_form;


f500_process_request: procedure;
/***********************************************************************
******* Q2.  Assign a high priority (17) to messages containing bids
*******      greater than 1000.  Other messages have the lower priority
*******     of 10.  The s$msg_send subroutine will return an error message
*******      if no server program is active when a message is sent.  The
*******      highest priority (19) is saved for future use.
***********************************************************************/
  if (bid_record.next_action = 'b') & (binary(bid_record.your_bid)  >
1000)
  then msg_priority =  HIGH_PRIORITY;
  else msg_priority = LOW_PRIORITY;
/***********************************************************************
******* Q3.  Send a message to the queue.
***********************************************************************/
  msg_length_in = bytesize(message_info);
  message_info.next_action = bid_record.next_action;
  message_info.your_bidder_id =  bid_record.your_bidder_id;
  message_info.item_no =  bid_record.item_no;
  if (bid_record.next_action = 'b')
  then message_info.your_bid = bid_record.your_bid;
  call s$msg_send(server_queue_port_id,
                  msg_priority,
                  msg_subject,
                  msg_length_in,
                  message_info,
                  msg_id,
                  error_code);

  if (error_code ^= 0)
  then call f900_task_error_routine (error_code, 'error from msg_send');

/*****************************************************************
******* Q4.  Receive a reply from the queue.
*****************************************************************/
  reply_length_in = bytesize(message_reply);
  call  s$msg_receive_reply(server_queue_port_id,
                           msg_id,
                           reply_length_in,
                           reply_length_out,
                           message_reply,
                           error_code);
 if (error_code ^= 0)
 then call f900_task_error_routine (error_code,
                                    'error from s$msg_receive_reply');

/***********************************************************************
******* T11. Redisplay the form with information obtained from the reply.
*******     User error messages are displayed on the last line of the form.
***********************************************************************/
 bid_record.description = message_reply.description;
 bid_record.high_bid = message_reply.high_bid;
 if (message_reply.error_message = '')
 then form_reply =  'Enter your bid now';
 else form_reply = message_reply.error_message;
 if (bid_record.next_action = 'i')
 then bid_record.your_bid = char (0);
 bid_record.next_action = '';
end f500_process_request;


f650_open_queue: procedure (p_port_name, p_path_name, p_port_id);

/* parameters */

declare   p_port_name         char (32) var in,
          p_path_name         char (256) var in,
          p_port_id           fixed bin (15);

/* automatic */

declare   error_code          fixed bin (15);

/*  execution */
```

```
 call s$attach_port(p_port_name, p_path_name, 0, p_port_id, error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                              'error from s$attach_port');

 call s$msg_open(p_port_id, REQUESTER_IO_TYPE, error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                              'error from s$msg_open');

 end f650_open_queue;


f700_find_entry:procedure (p_entry_name, p_add_epilogue,
p_error_routine);

/* parameters */

declare   p_entry_name             char (32) varying in,
          p_add_epilogue           entry (entry, fixed bin (15)),
          p_error_routine          entry (fixed bin (15), char (*) var);

/* automatic */

declare   entry_value              entry variable,
          error_code               fixed bin (15),
          null_pointer             pointer;


/* execution */

 null_pointer = null();
 call s$find_entry (null_pointer,
                   p_entry_name,
                   entry_value,
                   error_code);
 if (error_code ^= 0)
    then call p_error_routine (error_code, 'error from s$find_entry');

 call p_add_epilogue ((entry_value), error_code);
 if (error_code ^= 0)
 then call p_error_routine (error_code,
                 'error from adding epilogue handler ' || p_entry_name);

 end f700_find_entry;


f750_expand_path: procedure (p_path, p_suffix, p_exp_path);

/* parameters */

declare   p_path              char(256) varying in,
          p_suffix            char(32) varying in,
          p_exp_path          char(256) varying;
```

```
/* automtic */

declare   error_code          fixed bin (15);

/* execution */

 call s$expand_path (p_path, p_suffix, p_exp_path, error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                               's$expand_path: ' || p_path);
 end f750_expand_path;


f900_task_error_routine: procedure (p_code, p_text);

/* parameters */

declare   p_code              fixed bin (15) in,
          p_text              char (*) var in;

/*********************************************************************
******* T12. Handle task errors.  If an error occurs in a task, the task
*******      disables tasking and displays its error message on the
*******      process terminal.  The task then reenables tasking and stops
*******      itself.  Other tasks continue to run.
*********************************************************************/
 call s$set_process_terminal(terminal_switch);
 call s$get_task_id(task_id);
 caller = 'error from requester task #' || ltrim(char(task_id));
 call s$error(p_code, caller, p_text);
 call s$set_process_terminal(terminal_switch);
 call f950_stop_task;
end f900_task_error_routine;


f950_stop_task: procedure;
/*********************************************************************
******* T13. Stop the current task.  When a task stops, the task epilogue
*******      handler runs.
*********************************************************************/
 call s$control_task(CURRENT_TASK, STOP_TASK_CLEANUP, error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code, 'error from s$control_task');

end f950_stop_task;
```

```
f999_error_routine: procedure (p_code, p_text);

/* parameters */

declare   p_code         fixed bin (15) in,
          p_text         char (*) var in;

/*********************************************************************
*******      End the program (all tasks) on miscellaneous errors.
*********************************************************************/
 call s$error(p_code, caller, p_text);
 stop;
end f999_error_routine;


epilogue_for_tasks: entry;
/*********************************************************************
******* T14. Specify the task epilogue entry point.  The task epilogue
*******      handler closes the queue for each task.
*********************************************************************/
/*********************************************************************
******* T15. Specify the program epilogue entry point.
*******     When all tasks are stopped, the program epilogue handler runs.
*******      The operating system restarts task 1 to execute the program
*******      epilogue handler.
*********************************************************************/
 put list('Sorry - application stopping now');
 call s$close(server_queue_port_id,
             error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                               'error from s$close in task epilogue');

 call s$detach_port(server_queue_port_id,
                   error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code,
                           'error from s$detach_port in task epilogue');

 return;
 /* end epilogue_for_tasks; */


program_epilogue:    entry;
/*********************************************************************
******* Q5.  The program epilogue handler attaches a port to
*******      the server queue and opens the port.
*********************************************************************/
 call s$set_process_terminal(terminal_switch);
 put list('entering program epilogue');
 caller = 'program epilogue handler';
 call f650_open_queue ('cleanup_port', server_queue_path_name,
                      server_queue_port_id);
```

```
/*********************************************************************
******* Q6.  Send a message to the queue.  The program epilogue handler
*******     tells the server program to stop itself.
*********************************************************************/
 message_info.item_no = '000000';
 msg_subject = 'MONITOR SAYS STOP';
 msg_length_in = bytesize(message_info);
 call s$msg_send(server_queue_port_id,
                19,           /*msg_priority*/
                msg_subject,
                msg_length_in,
                message_info,
                msg_id,
                error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code, 's$msg_send in prog epilogue');
/*****************************************************************
******* Q7.  Receive a reply from the queue.  When the program
*******     epilogue handler receives the reply from the server
*******     program, this program ends.
*****************************************************************/
 reply_length_in = bytesize(message_reply);
 call s$msg_receive_reply(server_queue_port_id,
                         msg_id,
                         reply_length_in,
                         reply_length_out,
                         message_reply,
                         error_code);
 if (error_code ^= 0)
 then call f999_error_routine (error_code, 'error from
s$msg_receive_reply');
return;     /* end program_epilogue; */
/*****************************************************************
*******     The operating system closes and detaches ports
*******     because the hold_attached and hold_open bits were
*******     set to 0 when s$attach_port was called.
*****************************************************************/
end requester;
```

Glossary

abort

In transaction processing, to cancel a transaction. All transaction files involved in the transaction are guaranteed to remain exactly as they were before the transaction was started.

argument

A character string that specifies how a command, request, subroutine, or function is to be executed.

attach

1. To associate a port with a file or device, creating the port, if necessary, and generating a port ID. The port ID can then be used to refer to the file or device.

2. To associate an event with a process.

back up

To copy onto disk or tape some or all of the data stored on a system's disks. If data is later damaged or accidentally deleted, it can be retrieved from the backup copy in the state it was in at an earlier time. See also **restore**.

batch process

A noninteractive process that executes a program, command or command macro. When you request a batch process, the operating system puts the batch request into a specified queue and starts a batch process to execute the command whenever resources become available. The batch process runs independently of the process that issued the batch request. See also **process** and **subprocess**.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding compacts the code and resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the `bind` command.

binder control file

A text file containing directives for the binder.

break

A signal (or to send a signal) that interrupts a program being executed and places the process executing the program at break level.

busy

In queue processing, a message in a queue that has been received.

cache

A place in memory where data is put as it is read from disk. Data remains in cache until the user no longer needs it. Even then, the data may remain in cache until the cache page is needed by another process. Cache pages are reused based on a least-recently-used algorithm.

clean up

In a tasking program, to close and detach a task's terminal device and prepare the task to be reinitialized.

command

A program invoked from command level, either interactively or as a statement in a command macro.

command function

A function that can be invoked when a process is at command level. An example of a VOS command function is `(time)`, which is replaced with the current time. A command function must be enclosed in parentheses.

command level

The state at which you can issue commands to the command processor or run programs or macros.

command line

A set of one or more commands, separated by semicolons. Pressing one of the following keys terminates a command line: RETURN, ENTER, CANCEL, and DISPLAYFORM .

command name

The name of a program, command, or command macro that can be called or invoked as a command.

commit

In transaction processing, to make permanent and visible to other transactions all updates to a set of transaction files accumulated since the transaction was started. This is done by calling `s$commit_transaction`. When `s$commit_transaction` returns to its caller without error, all updates are guaranteed.

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

configuration table

One of the table files that the operating system uses to identify the elements of a system or network. For example, the file `devices.table` contains information about each device present in the system, the file `disks.table` contains information about each disk present in the system, and the file `modules.table` contains information about each module in the system. See also **task configuration table**.

continue

In a tasking program, to allow a paused task to continue waiting or running.

current module

The module associated with a process; the module on which the process is executing. You cannot change the current module of a process.

current position

1. For files other than pipe files, the file's current position is the location in a file at which the next operation will be performed.

   Pipe files have two current positions. These are: (1) the position in the file at which data will be written by the next output operation, and (2) the position in the file at which data will be read by the next input operation. The two current positions of a pipe file are the same for all processes attached to the file.

2. In a record file, the current record; in a stream file, the current line and column.

current record

The record to which the current file position is set.

current system

The Stratus system associated with a process; the system containing the current module of the processes.

deadlock

In transaction processing, a situation in which a lock contention cannot be resolved.

default value

The value for an argument or parameter that a program or operating system uses if a specific value is not supplied.

detach

1. To dissociate a port from a file or device.

2. To dissociate an event from a process.

direct queue

In queue processing, a fast, memory-resident, one- or two-way queue whose depth is limited to one message per attached port, and whose message length is limited to 3,072 bytes. This queue cannot be transaction protected.

dynamic task

In a tasking program, a task that can be created or deleted at run time.

entry point

A statement in a program at which execution of the program can begin. An entry point can be the target of a calling statement, and it can be specified in a binder control file.

entry variable

A variable that can be used in a program to hold the value of an entry point.

error code

An arithmetic value (usually a two-byte integer representing a VOS status code) indicating what, if any, error has occurred. An error code argument is often included in subroutines.

error message

A character string that is associated with an error code.

event

A data structure, associated with a file or device, that is used by processes to communicate with one another. When one process notices that some action has occurred that is of potential interest to one or more other processes, the first process notifies the event. This notification causes the operating system to inform all processes that have been waiting for the action that the action has occurred. (These processes are said to be *waiting on the event*.) Each time an event is notified, its event count is incremented.

event attachment

The association of an event with a process.

event count

An integer value that is associated with an event. Each time the event is notified, its event count is incremented.

event status

An integer value, associated with an event, that can be used to tell waiting processes the reason for the event's notification.

executable image

The object created from a program module by the loader. The modifiable part of the executable image does not reside in the file hierarchy, but rather in temporary paging storage that is shared by many processes. Parts of an executable image can be swapped in and out of main storage.

fault tolerance

A system containing duplexed hardware components that operate in lock-step simultaneously. Processing continues even if a component fails.

fence

A portion of the user's virtual address space adjacent to the process or task user stack. The fence allows the operating system to detect, without appreciable overhead, most references to addresses beyond the end of the stack. A fence prevents data corruption in adjacent regions.

full path name

For a file, directory, or link, a name consisting of the system name, the disk name, the names of the directories that contain the object, and the name of the file, directory, or link.

For a device, a name consisting of the system name and the device name.

A full path name refers to only one object; an object has only one full path name. (However, many links can refer to the same object.)

heap

A collection of randomly accessible memory associated with a process and available for allocation. See also **user heap**.

implicit locking

A locking mode in which the operating system does not lock a file or record for either reading or writing when it opens the file; instead, the operating system locks the file for the appropriate access type each time a process performs an I/O operation on the file or record. At the end of each I/O operation, the file is unlocked.

interactive process

A process that is started with the `login` command. A user working in an interactive process is engaged in a dialogue with the operating system, issuing commands and receiving responses, normally from a CRT terminal.

internal command

A command that is built into the operating system. For a list of VOS internal commands, issue the `help` command with the argument `-type internal`.

jiffy

A unit of time equal to 1/65,536 of a second.

kernel

The part of the operating system that performs all privileged system operations. The command processor and the debugger are included in the kernel.

link

An object contained in a directory that directs all references to itself onward to a file, a directory, or another link. Like many other objects, a link has a path name that identifies it as a unique entity in the system directory hierarchy.

lock

A system data structure associated with a file, file record, or device that can be set to restrict the use of the object; the restriction remains in effect until the lock is released.

lock arbitration

In transaction processing, the scheme used by the system to resolve lock contention and award the lock to one of the conflicting transactions. The arbitration algorithm is partially controlled by user-provided parameters.

lock contention

In transaction processing, a situation that occurs when two or more transactions need a conflicting lock on an object. For example, lock contention exists if two transactions require write locks on the same record.

lock contention parameters

In transaction processing, parameters that determine the outcome when one or more tasks or processes simultaneously attempt to set conflicting locks on the same object.

lock wait time

The maximum time that a task, process, or transaction will wait to acquire a lock during any I/O operation. If the lock cannot be obtained before the lock wait time is exceeded, the I/O request returns with an error message stating that the required object is in use.

log

To record system activity in a file, called a log file. The operating system maintains certain log files, such as `syserr_log.`*`date`* and `remote_maint_log.`*`date`*. You can log I/O activity through a given port with the `start_logging` command or the `s$start_logging` subroutine.

login terminal

A terminal from which a valid user can execute, interrupt, or stop execution of VOS commands, command macros, or programs.

macro event

An event_id used to collect other event_ids. A macro event is a two-level hierarchy of events.

message

In queue processing, an entry in a queue.

message identifier

In queue processing, a number assigned by the system to a message when the message is added to a queue.

message priority

In queue processing, a message attribute that affects the order in which the messages in the queue are processed. The priority is an integer in the range 0 to 19, inclusive. The higher number indicates greater urgency.

message queue

In queue processing, a disk-resident, one-way only queue whose contents may be transaction protected. The message length is limited to 2^{23} bytes when the queue and the requester are on the same module, and to 2^{20} bytes when the queue is remote to the requester.

module

A single Stratus computer. A module is the smallest hardware unit of a system capable of executing a user's process.

module name

The name of a module. Module names are specified on the module's disk label and by a system administrator in the module's configuration table. (See **configuration table**.) The path name of a module has two components: (1) the name of the system, prefixed by a percent sign, and (2) the name of the module, prefixed by a number sign. For example: `%s1#m5`. Note that device names and disk names have the same form as module names.

monitor task

In a tasking process, a task executing either the `s$monitor` or `s$monitor_full` subroutine. The task executes interactive requests that control or display information about the tasks in the process.

monitor task request

In a tasking program, a command that is executed from the monitor task. TPF provides a standard set of monitor requests. User-written programs can be added to this set. Requests are not executable unless they are retained in the program's binder control file.

notify

To increment the event count of an event. When a process notifies an event, the operating system informs all of the processes that are waiting on the event.

no-wait mode

A port setting that causes a task or process to continue running if a requested I/O operation cannot be completed immediately.

object

In the VOS operating system, any data structure or device in the system that you can refer to by name or by some other identifier. For example, all of the following are objects: directories, files, links, systems, modules, devices, groups, persons, ports, queues, locks, file indexes, and file records.

object module

A file produced by a compiler, containing the machine code version of one or more procedures. This file usually contains symbolic references to external variables and programs. An object module must be processed by the binder to produce a program module, and then must be loaded by the loader.

object name

A character string identifying an object. The maximum length of an object name is 32 characters. Examples of objects that have object names are files, directories, and links. The object name is the last component of the full path name. For example, `file1` is the object name in the full path name `%sys1#m30>prod>data>file1`.

one-way queue

In queue processing, a queue that handles messages, but not replies to messages.

open

To prepare a file or device for a particular type of access. The file or device must be attached to a port in the process before opening.

organization

The way records in a file are stored on a disk. Valid VOS organizations are sequential, relative, fixed, and stream. In queue processing, additional valid organizations are message queue, server queue, and one-way server queue.

page

A unit of virtual storage containing 4,096 bytes. A page is the smallest unit of storage that the operating system moves between main storage and secondary storage on a disk.

paging

Dividing a virtual address space into pages and managing pages (reading in from the disk to main storage and writing out to disk, when necessary) when a process refers to virtual addresses in the pages. Paging is invisible to users' programs.

path

An attribute of an object. The object's path is the sequence of objects (system, disk, directory) that are superior to the object in the directory hierarchy.

path name

A unique name that identifies a device or locates an object in the directory hierarchy. See also **full path name** and **relative path name**.

pause

In a tasking program, to suspend the waiting or running of a task.

pipe file

A file that is used to connect processes, so that one process puts data in the file and another process takes data from the file. The command `set_pipe_file` designates a file to be a pipe file.

port

A data structure, identified by a name and ID, attached to a file or device. An executing program accesses a file or device using the port name or ID. A port is created when it is attached, and it is destroyed when it is detached. A port is unique to a process.

port attachment

The creation of a port in order to access a file or device.

port name

The character-string name of a port.

position

The location of a record or character in a file.

primary index

A file's index that was defined when the file was opened. When no other index is defined for the current keyed operation on the file, the operating system uses the primary index.

primary task

In a tasking program, the task that is executing by default when the program begins executing. The task ID of the primary task is 1.

privileged

A user attribute allowing the user to invoke certain commands, requests, and subroutines. A user is privileged or not, depending on the user's status as defined in the registration databases and the arguments specified with the `login` command.

privileged process

A process of a user who is logged in as privileged.

process

A collection of system resources whose purpose is the execution of system programs (such as the Overseer) or user programs, commands, and macros. The `login` command creates an interactive user process. An interactive user process can create subprocesses and batch processes. See also **subprocess** and **batch process**.

process terminal

The terminal of the primary task (task 1).

program entry

See **entry point**.

queue

A logical collection of messages awaiting transmission or processing. Messages can be given priorities within a queue. The operating system uses queues to record jobs that are waiting to be processed.

queue depth

In queue processing, the number of messages in a queue.

read

In queue processing, to examine a message without receiving it.

receive

In queue processing, to start processing a message by marking it as busy.

record

The data structure that the operating system uses to manage data in a file. For files other than stream files, a record is the smallest unit of data that the operating system I/O routines can access when performing I/O operations on files or devices. For stream files, a record consists of unstructured data.

relative path name

A name that identifies a device or an object in the directory hierarchy without specifying its full path name.

reply

In two-way queue processing, text that replaces the message text in a two-way queue. A server program must send a reply to a message that it previously received and processed.

request

See **monitor task request**.

requester

In queue processing, a process or task that sends messages to a queue and, for two-way queues, waits for replies.

requester/server model

A typical online transaction-processing design model that consists of requester and server programs communicating through a queue. The requester program interfaces with users and sends user requests to the queue. The server program takes the requests from the queue and services them. The server program usually interfaces with the application's data files.

restore

To retrieve data from one or more backup disks or tapes when current data on a disk is damaged or accidentally deleted. The data is retrieved in the state it was in at the time of the backup. See also **back up**, **roll forward**, and **tp_restore**.

roll forward

To reapply transactions to a backup version of a data file. Transactions are reapplied in the same order as they originally occurred.

send

In queue processing, to add a new message to a queue.

server

In queue processing, a process or task that waits for messages in a queue, receives them, processes them, and, for two-way queues, sends replies to them.

server queue

In queue processing, a one-way or two-way queue whose message length is limited to 2^{23} bytes if the queue resides on the same module as the requester program, and to 2^{20} bytes if the queue is remote to the requester. A two-way server queue can transmit transaction protection.

shared variable

1. A static external variable that can be shared by more than one object module simultaneously. Shared variables are allocated in the virtual address space of a process. The same variable in a different process is a distinct entity.

2. In a tasking program, a variable allocated only once for all tasks. Unshared variables are allocated on a per-task basis.

single-event notify

A feature available with server queues that affects the number of server processes notified when new messages are added to a specific queue. The feature limits notification of a new message to only one server process, instead of notifying all available server processes and letting them compete for the new message. Since only one process can service a message, this feature conserves system resources when there are more server processes waiting to service messages than there are messages to be serviced.

slave state

A non-login state for a terminal from which the user cannot break out of the program, terminate the program, or do anything not explicitly allowed by the program or task controlling the terminal.

source module

A text file (single source program) containing language statements, compile-time statements, and comments that can be compiled to produce an object module.

stack

An area of storage consisting of an ordered series of stack frames associated with the execution of a program.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects. Star names function in the following manner.

- An asterisk can be in any position in a star name.
- In a path name, a star name can be in the final object-name position only.
- When the operating system matches nonstar names to a star name, each asterisk represents zero or more characters.

static data

Per-process temporary data used by a program. External static data can be shared among several object modules (in which case it is called a shared variable); internal static data can be referenced only by the program that declared it.

static region

The region of an object module that contains external references and internal static data.

static task

A task that is defined at bind time and remains defined for the duration of the program run.

status code

A code with an associated name and often an associated line of text. There are four types of status code:

- error codes, whose names begin with the prefix `e$`
- message codes, whose names begin with the prefix `m$`
- query codes, whose names begin with the prefix `q$`
- request codes, whose names begin with the prefix `r$`

subprocess

An additional user process that begins while the parent process is suspended but not terminated. The new subprocess inherits several attributes from the parent process, such as priority and access privileges, and the current directory. Each successive subprocess that you start is a child of the preceding process. Therefore, you must log out of each new subprocess to return to its parent. See also **process** and **batch process**.

suffix

A character string that begins with a period and is added to an object name to indicate the type of the object.

system

Either a single module or a group of up to 32 modules located at the same site (the same building or neighboring buildings), connected in a local network and defined as a single logical entity.

table file

A file that the operating system creates from a `.tin` file when the `create_table` command is issued. All of the records in a table file have the same format and are subdivided into fixed-length fields. The name of a table file always ends in the suffix `.table`.

table input (`.tin`) file

One of the two input files required to execute the `create_table` command. The table input file, known as the `.tin` file, contains the data for the table file to be created.

task

A process subdivision that shares many of the characteristics of the process itself but shares some of the process resources with other tasks. The same section or different sections of program code may be executed in each task, with different file and device attachments. Because some process resources are shared among tasks, it may be more efficient to have a given application program executing in multiple tasks of the same process rather than in multiple processes.

task configuration table

A configuration table required by the `s$init_task_config` subroutine. The table contains the information required to associate specific terminals to static tasks in a tasking process. See also **configuration table**.

task data region (TDR)

A data structure maintained only for tasking programs. It is located in the user heap, and contains information about a task such as state, priority, terminal port ID, CPU time consumed, and stack length.

task identifier (task ID)

In a tasking program, the integer assigned by the binder or by the operating system that identifies a particular task within a process.

task priority

In a tasking program, a task attribute that affects the order in which tasks are scheduled to run and the order in which tasks waiting on events are rescheduled. The priority is an integer value in the range 0 to 255, inclusive. Tasks with higher-numbered priorities have higher precedence.

task scheduling

In a tasking program, the order in which tasks are selected for execution.

task state

In a tasking program, one of eight possible task states: uninitialized, initialized, ready, running, waiting, paused waiting, paused ready, and stopped.

task switch

In a tasking program, the transfer of control within a process from one task to another. This occurs when the currently executing task must wait on some event is paused or stopped, or explicitly calls `s$reschedule_task`. The task stops running, and the next task is chosen according to task priority from those tasks that are ready to run.

TDR

See **task data region.**

TID

See **transaction identifier.**

`tp_restore`

In transaction processing, a command that rolls a transaction file forward from a previous state using a backup version of the transaction file and saved transaction logs.

TPOverseer

In transaction processing, a background process that performs transaction-related work. The TPOverseer must be executing on all modules involved in transaction protection.

transaction

In transaction processing, a unit of work defined within an application program in the form of executable statements. While any kind of work can be performed within a transaction, only I/O operations on transaction files are bound and protected by the transaction protection capability. A transaction changes a transaction file or a set of transaction files from one consistent state to another consistent state. This means that one of two things **always** happens: within a transaction, either all of the operations requested on transaction files are performed, or none are performed.

transaction file

In transaction processing, a VOS data file for which all I/O is transaction protected. The file can be accessed only after a call to `s$start_transaction` or `s$start_priority_transaction` and before a call to `s$commit_transaction` or `s$abort_transaction`. Transaction processing guarantees that any changes made to a transaction file are never left partially completed, regardless of the state of the hardware or software. Either all of the changes are completed, or none are made.

transaction identifier (TID)

In transaction processing, an identifying number assigned to a transaction by the system, when the transaction is started with the `s$start_transaction` subroutine. The term TID is also used to identity network transactions which may or may not be TPF transactions.

transaction log

In transaction processing, a log file containing all I/O operations performed on transaction files.

transaction priority

In transaction processing, a user-assigned value indicating the relative importance of a transaction. The priority value is one of the parameters used in the lock arbitration algorithm to help decide which transaction should win locks when lock contention occurs.

transaction protection

A feature that ensures the integrity of a file or set of files by requiring that all access to the file or files occurs within the bounds of a transaction.

two-way queue

In queue processing, a queue that requires a server to reply to a message received from a requester.

user heap

A portion of virtual address space in which the operating system can allocate storage for the user's programs. The user heap is a free storage region located after the executable image in the user's region of virtual storage. The user heap expands toward the user stack, which expands down from the highest virtual address.

user name

An identifier composed of a person name and a group name. For example, `Janet_Smith.Sales`.

user stack

A portion of a user's virtual address space that is used for the user's program procedure call stack.

virtual address space

A set of addresses to which a process can refer.

Virtual Operating System (VOS)

The virtual operating system on Stratus modules.

wait mode

A port setting that requires a task or process to wait until a requested I/O operation completes before the task or process can continue running.

wired memory

Pages of virtual memory that have physical pages associated with them. Unlike virtual memory pages, wired pages are not swapped out of memory.

Index