RMX/86™ I/O SYSTEM REFERENCE MANUAL

Manual Order Number: 9803123-01

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

BXP	Intellec	Multibus
i	iSBC	Multimodule
ICE	iSBX	PROMPT
iCS	Library Manager	Promware
Insite	MCS	RMX
Intel	Megachassis	UPI
Intelevision	Micromap	μScope

and the combination of ICE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix.

A206/0580/1K FL

Preface

PURPOSE AND SCOPE OF MANUAL

The RMX/86 Operating System provides software support for Intel's iSBC 86/12 single-board computer. It consists of a nucleus, terminal handler, debugger and input/output system. Each user can configure the operating system to include only the features he needs.

The IOS reference manual is one of four manuals furnishing an overview of the RMX/86 Operating System and reference material for each of its components. The four manuals ideally should be read in the following sequence:

An Introduction to the RMX/86™ Operating System....9803124
RMX/86™ Nucleus, Terminal Handler, and Debugger Reference Manual.....9803122
RMX/86™ I/O System Reference Manual.....9803123
RMX/86™ System Programmer's Reference Manual.....142721

This manual assumes familiarity with concepts and terminology introduced in the Nucleus reference manual and with the PL/M programming language. It is intended primarily as a quick reference to the system calls available in the I/O system. Only PL/M calling sequences are shown here.

Detailed descriptions of I/O system calls are limited to those available to applications programmers. Some calls reserved for system programmers are discussed generally, but only to give an overview of I/O system operation. The latter are described in detail in the RMX/86 System Programmer's Reference Manual.

In the first seven chapters of this manual, system calls are named using a generic shorthand (such as CREATE$FILE) or a more specific form (such as A$CREATE$FILE for the asynchronous version of this call). The actual PL/M external-procedure names used to invoke these I/O operations are shown only in Chapter 8, where the detailed PL/M calling sequences are listed.

NOTE
Information in this manual relating to stream files, job creation and termination, loading jobs, and hybrid/synchronous processing describes software that is not included in the first release and as such is only preliminary and subject to change.

CONTENTS

PAGE

PREFACE..... i
Purpose And Scope of Manual..... i

CHAPTER 1
INTRODUCTION TO THE RMX/86 I/O SYSTEM
File Connections.....1-1
Devices.....1-1
Files.....1-3
 Physical Files.....1-3
 Stream Files.....1-3
 Named Files.....1-4
  Directory Tree.....1-4
  Accessing Named Files.....1-4
  File-Access Protection.....1-6
Three Levels of System Calls.....1-6
Manual Organization.....1-8

CHAPTER 2
ASYNCHRONOUS INPUT/OUTPUT
Calls That Create File Connections.....2-1
Calls That Modify File Connections.....2-1
Calls That Obtain File Information.....2-2
Calls That Perform File I/O.....2-3
A Call To Peform a Device-Level Function.....2-4
Calls That Delete Information, Files, and Connections.....2-4

CHAPTER 3
HYBRID INPUT/OUTPUT
Calls that Create File Connections.....3-1
Calls That Modify File Attributes.....3-2
Calls That Obtain File Information.....3-2
Calls That Delete Files and Connections.....3-2
Asynchronous Calls Used At the Hybrid Level.....3-3

CHAPTER 4
SYNCHRONOUS INPUT/OUTPUT
Calls That Create File Connections.....4-1
Calls That Modify File Attributes.....4-2
Calls That Obtain File Information.....4-2
Calls That Perform File I/O.....4-3
A Call To Perform Device-Level Function.....4-4
Calls That Delete Files and Connections.....4-4

CONTENTS (continued)

PAGE

CHAPTER 5
JOB ENVIRONMENT
Creating and Terminating Jobs.....5-1
Default User.....5-1
Default Prefix.....5-2

CHAPTER 6
TIME AND DATE FUNCTIONS.....6-1

CHAPTER 7
RMX/86 LOADER
Loader Operation.....7-1
Creating A Job.....7-1
Package Object.....7-1

CHAPTER 8
INVOKING I/O SYSTEM CALLS IN PL/M
Syntax Notation.....8-1
Input Parameter Specification.....8-2
Condition Codes.....8-6
System Calls.....8-7
A$Attach$File.....8-8
A$Change$Access.....8-10
A$Close.....8-12
A$Create$Directory.....8-13
A$Create$File.....8-15
A$Delete$Connection.....8-18
A$Delete$File.....8-19
AGetConnection$Status.....8-21
AGetDirectory$Entry.....8-23
AGetFile$Status.....8-25
AGetPath$Component.....8-30
A$Open.....8-31
A$Read.....8-33
A$Rename$File.....8-35
A$Seek.....8-37
A$Special.....8-38
A$Truncate.....8-40
A$Write.....8-41
CreateIOJob.....8-43
Delete$Package.....8-47
ExitIOJob.....8-48

CONTENTS (continued)

PAGE

Get$Default$Prefix....8-51
Get$Default$User....8-52
Get$Time....8-53
Get$Time$String....8-54
H$Attach$File....8-55
H$Change$Access....8-56
H$Create$Directory....8-58
H$Create$File....8-60
H$Delete$Connection....8-62
H$Delete$File....8-63
HGetFile$Status....8-64
H$Look$Up$Connection....8-69
H$Rename$File....8-70
Inspect$Package....8-71
S$Attach$File....8-72
S$Change$Access....8-74
S$Close....8-76
S$Create$Directory....8-77
S$Create$File....8-79
S$Delete$Connection....8-81
S$Delete$File....8-82
Set$Default$Prefix....8-83
Set$Default$User....8-84
SGetConnection$Status....8-85
SGetFile$Status....8-88
S$Load....8-93
S$Look$Up$Connection....8-98
S$Open....8-99
S$Read$Locate....8-101
S$Read$Move....8-103
S$Rename$File....8-104
S$Seek....8-105
S$Special....8-106
S$Truncate$File....8-108
S$Write$Move....8-109
S$Write$Update....8-110

APPENDIX A
SUMMARY OF I/O SYSTEM CALLS
Job-Level System Calls....A-1
Get Time/Date System Calls....A-1
Load File/Task System Call....A-2
Create-File-Connection System Calls....A-2
File Modification System Calls....A-2
File Input/Output System Calls....A-3
Device-Level Function System Calls....A-4
Get Status/Attribute System Calls....A-4
Delete Connection/File System Calls....A-5

CONTENTS (continued)

PAGE

APPENDIX B
PL/M EXTERNAL PROCEDURES.....B-1

APPENDIX C
I/O REQUEST/RESULT SEGMENT
Segment Structure.....C-1
Status Codes.....C-2

APPENDIX D
EXCEPTIONAL-CONDITION CODES
Programming Errors.....D-1
Environmental Conditions.....D-2
Loader Condition Codes.....D-3

GLOSSARY.....G1-3

TABLE OF FIGURES

FIGURE	TITLE	PAGE
1-1	Task-File Communication For Traffic-Monitoring Application	1-2
1-2	Directory Tree for Auto Dealership Application	1-5
8-1	Sample Directory Tree (Asynchronous Call)	8-3
8-2	Sample Directory Tree (Synchronous Call)	8-5

Chapter 1. INTRODUCTION TO THE RMX/86 I/O SYSTEM

The RMX/86 input/output system allows tasks to communicate with one another and with the outside world. The I/O system is built around the concept of file manipulation, where a "file" can be:

- a physical device (physical file),
- a data "stream" or pipeline between two or more tasks (stream file), or
- data or directory information residing on a random-access device (named file).

Figure 1-1 shows part of a traffic-control application using all three kinds of files. This broad interpretation of file types makes the RMX/86 I/O system truly device-independent.

Tasks access the I/O system through various system calls. These calls provide numerous file-processing functions, but their main roles are:

- to create, attach, and delete files;
- to read and write files once they are established.

Other system calls modify file attributes, obtain file status information, and perform special device-level functions. Still other calls provide auxiliary functions like creating and terminating jobs, setting or inspecting job-level default parameters, accessing time and date information, and loading object files.

FILE CONNECTIONS

When a file is created or attached, a connection to that file is also established. A file connection is an RMX/86 object that contains the means to access the file, a file pointer (in cases where the file has been opened for reading or writing), and, in some instances, input/output buffers.

When a task issues a CREATE$FILE, ATTACH$FILE, or CREATE$DIRECTORY system call, it receives the token for a file connection in return. The task can then supply this token to the I/O system to gain access to the file in subsequent operations. A file connection, once established, remains valid through any number of I/O operations until it is deleted. This saves the task considerable overhead; the file does not have to be located again each time it is opened or otherwise accessed.

DEVICES

A device might be any one of a broad spectrum of physical units, such as an I/O terminal, serial input or output unit (e.g., paper-tape unit or line printer), or random-access storage (e.g., floppy or hard-disk unit). A device might also be a thermistor in an oven or a traffic sensor embedded in a roadway.

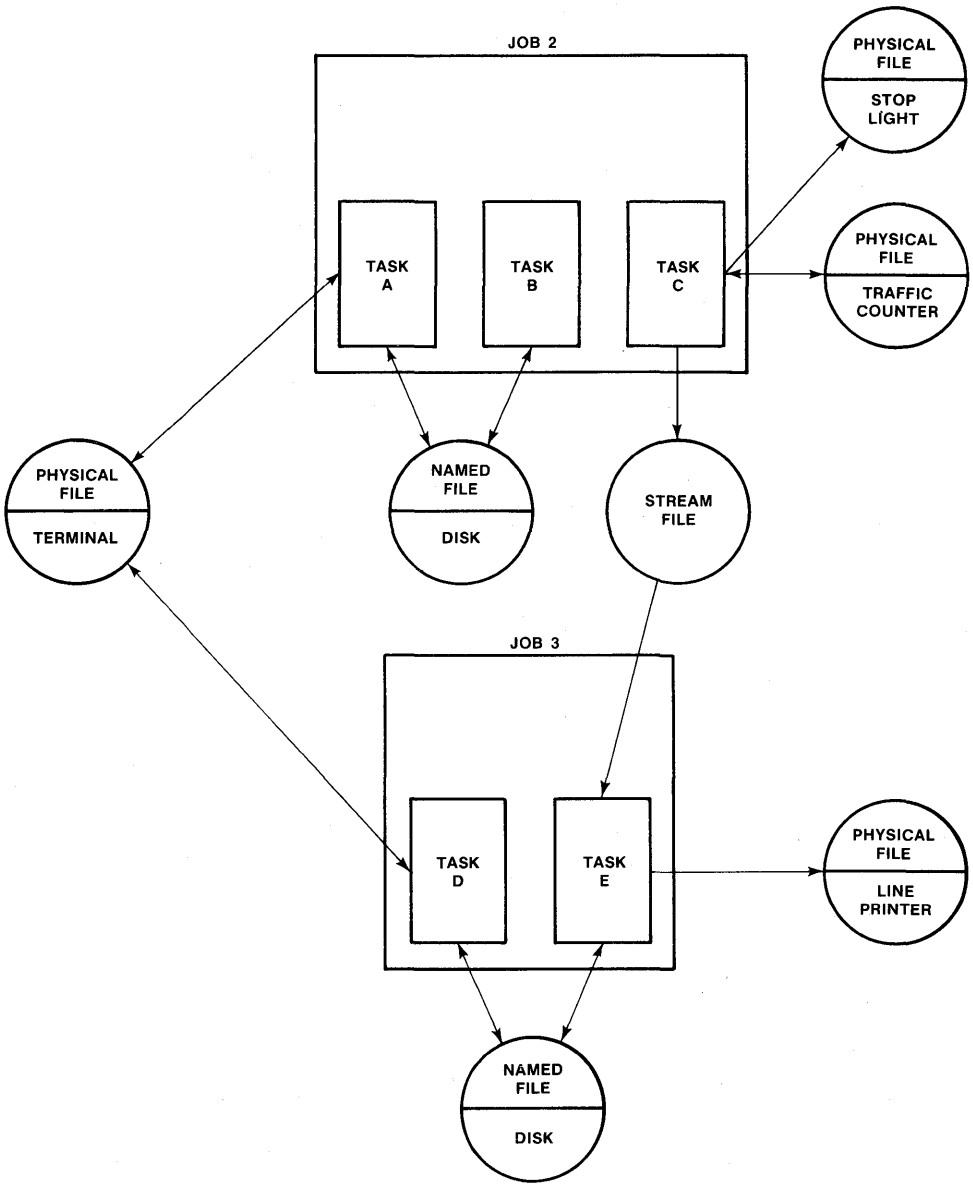


Figure 1-1. Task-File Intercommunication For Traffic-Monitoring Application

In the case of a random-access device, such as a disk drive, the device also includes a "volume." A volume is simply the physical storage medium used by the particular device (such as a diskette or hard disk platter).

Before a file can be created on a device, the device must be attached. When a device is attached, a token for a device connection is returned to the caller (normally a system program). This token is used later by applications programs creating files on the device.

FILES

PHYSICAL FILES

Some tasks deal directly with physical devices; in fact, some tasks deal only with physical devices. For example, a task might monitor a sensor continuously and print a message each time an event occurs.

To read the sensor data and write the output message, the task must establish physical-file connections to the sensor and printer devices. First, the devices must be attached, as mentioned above. The device connections returned are then used as parameters in CREATE$FILE system calls. Each call returns a physical-file connection to its respective device.

Using the tokens for these file connections as parameters, the task can now issue calls to OPEN the two physical-file connections, READ from the input connection, and WRITE to the output connection. When all I/O operations are completed, the file connections can be CLOSED and DELETED.

STREAM FILES

Stream files provide a mechanism for intertask communication without the use of external devices or media. A stream file is useful only when there are both a writer and a reader of the stream file, as when the output of one task is connected to the input of another.

To establish a stream-file communication link, the following steps are generally required (although the exact protocol can vary).

- Call CREATE$FILE to create the stream file and a connection to the file.
- Call ATTACH$FILE to create an additional connection to the file.
- Give one connection to the writing task and the other to the reading task; The tasks can then OPEN the connections for I/O.

Multiple readers and/or writers can use the same stream file by creating the appropriate set of connections. A read of a stream file is a destructive read, however, and such multiple operations must be carefully synchronized.

NAMED FILES

A named file is a sequence of bytes residing on a random-access device. Named files can be either data files or directory files. A directory file is a file whose entries are pointers to other (data or directory) files. A directory file can also be empty.

Directory Tree

Each random-access volume supports a tree of directories. Named files are components in these hierarchical trees (Figure 1-2). A named file is accessed by identifying the device where it resides and a path through the tree to which it belongs. As Figure 1-2 illustrates, the root of a directory tree is called the root directory. The tree's internal nodes are directories; its leaves are data files or empty directories.

The file names shown in this figure are completely arbitrary. When a named data or directory file is created, the name specified by the creator is automatically cataloged by the I/O system in the directory that points to the file. That directory is known as the file's parent directory. For example, the information needed to access file TUNE_SCHED is cataloged in its parent directory, TUNEUPS. The information needed to locate TUNEUPS is cataloged in directory SERVICE.

Accessing Named Files

A named file is accessed by specifying its directory-tree path in an I/O system call.

A directory-tree path has two parts. The first part, or prefix, designates the file in the directory tree where the path search is to begin. The second part, or subpath, describes the rest of the route through the tree to the desired file. The subpath is a list of directory names, ending with the name of the file being accessed. In Figure 1-2, for example, if the prefix designates directory SERVICE, the subpath to file 80_PART_LIST is

TUNEUPS/PARTS/80_PART_LIST

A subpath can also be null, in which case the prefix itself designates the target file.

The exact syntax for prefix, subpath, and file-name specification is described in Chapter 8, where calling sequences are covered in detail.

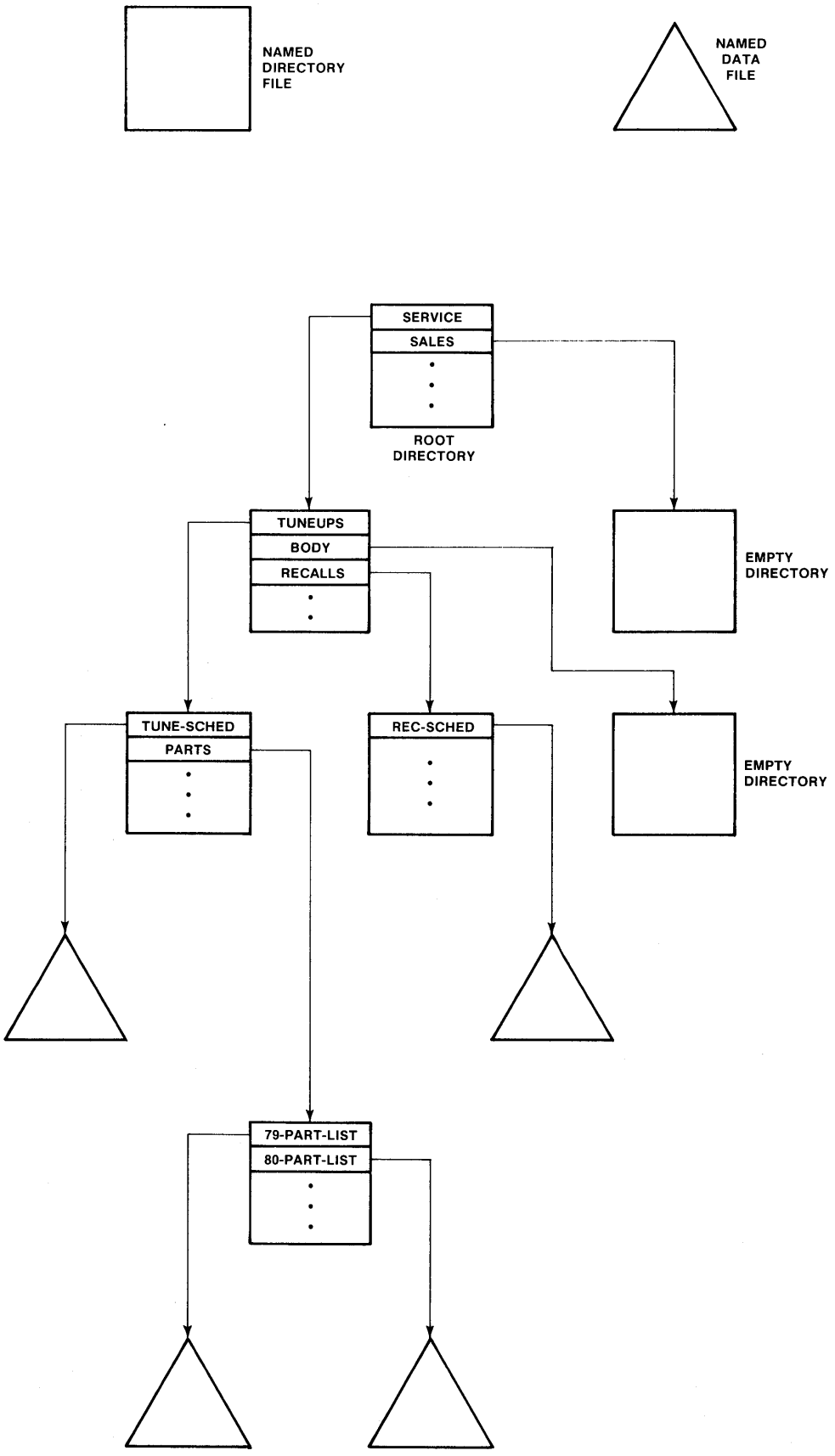


Figure 1-2. Directory Tree For Auto Dealership Application

File-Access Protection

The creator of a named file can limit access rights to the file for himself and other users. The I/O system provides file-access protection by combining the concepts of a user object and an access list.

Each user of the I/O system is associated with a user object that identifies not only that user, but also all groups to which he belongs (engineering team, quality assurance committee, etc.). These objects are designated in I/O system calls by a token or, in the H$CHANGE$ACCESS and S$CHANGE$ACCESS calls, by the logical name "WORLD," which signifies all users of the system.

The specific access rights allowable are:

Named Data Files	Named Directory Files
Delete	Delete
Read	Display
Append	Add Entry
Update	Change Entry

When a named file is created by an A$CREATE$FILE or A$CREATE$DIRECTORY system call, the creator specifies the token for his user object and a byte mask describing the (combination of) access right(s) to be granted. The access mask may later be changed or more users granted access by calling A$CHANGE$ACCESS (up to a total of three "user/access" pairs per file). When the file is subsequently accessed, the I/O system compares the specified user-id value with the user/access list for the file to see if that value is present and has the requested access privilege.

When a named file is created by a call to H$CREATE$FILE, H$CREATE$DIRECTORY, S$CREATE$FILE, or S$CREATE$DIRECTORY, the global user "WORLD" is assumed and full access rights are granted automatically. The rights can be limited by subsequent calls to H$CHANGE$ACCESS or S$CHANGE$ACCESS, but the user object is still "WORLD."

As this discussion of file protection indicates, the I/O system frequently offers several system calls to perform the same function. The following section describes these options in more detail.

THREE LEVELS OF SYSTEM CALLS

At the beginning of this chapter, the distinction was made between file-related system calls (that create, modify, delete, inspect, or perform I/O on files) and auxiliary system calls

(that create jobs, set job defaults, access time/date information, and load object files). In the case of file-related system calls, the I/O system provides three levels of operation.

- asynchronous
- hybrid (partially asynchronous, partially synchronous)
- synchronous

In a real-time environment, the events that trigger system response usually happen at unpredictable times and in an unpredictable sequence. That is, they happen asynchronously. The controller of the system is responsible for synchronizing these events.

At the asynchronous level of I/O operation, the controller of the system is the applications programmer. The programmer must be familiar with techniques for synchronizing his I/O operations, primarily using mailboxes (as described in the RMX/86 Nucleus, Terminal Handler and Debugger Reference Manual). The advantages of this most basic level of I/O operation are that it requires the least system memory of the three options, it gives the programmer the greatest flexibility in specifying system call parameters, and it allows the programmer to perform other operations in parallel with the asynchronous I/O operations.

At the hybrid level, many operations are synchronized automatically, simplifying the programmer's role. System calls are simplified also, as defaults are assumed for some parameters and response mailboxes are not needed. Where they are required, user objects and file-connection objects can be given logical names that can be specified instead of 16-bit tokens in system call parameters. The file connections created at this level are fully compatible with those created asynchronously, allowing the hybrid level to perform I/O on established data-file connections using asynchronous calls, and thus retain some of the flexibility of the asynchronous level.

At the synchronous level, all operations are synchronized automatically by the I/O system, relieving the programmer of this burden completely. System calls are simplified, as at the hybrid level, and the logical-naming capability exists at the synchronous level also. In addition, I/O buffers built into data-file connections allow automatic overlapping of I/O operations, making this level particularly efficient for processing sequential data. The automatic buffering also provides blocking and deblocking of I/O data, usually resulting in increased device throughput. The expense of this programming convenience is the greater system memory required for the synchronous level, and the inability to specify arbitrary buffering and synchronizing.

MANUAL ORGANIZATION

Chapters 2-4 describe these three levels of file operations in more detail and summarize the system calls available at each level. Chapters 5-7 summarize the system calls available for performing auxiliary functions. Chapter 8 provides the detailed PL/M calling sequences for every I/O system call available to the applications programmer.

Chapter 2. ASYNCHRONOUS INPUT/OUTPUT

Asynchronous operations represent the most basic level of RMX/86 I/O. While this level provides fewer features than the hybrid and synchronous levels, it does allow the programmer greater flexibility in specifying parameters and in performing multiple I/O operations simultaneously. The programmer must synchronize these operations himself, using result segments returned to mailboxes designated in asynchronous system calls. Asynchronous I/O processing also requires the least system memory of these three levels.

This chapter provides an overview of asynchronous system calls. The detailed PL/M calling sequences and descriptions of these calls can be found in Chapter 8. Not every system call is applicable for every kind of file, as indicated in the following summaries.

CALLS THAT CREATE FILE CONNECTIONS

Three system calls are available for creating connections at the asynchronous level:

- A$CREATE$FILE
- A$ATTACH$FILE
- A$CREATE$DIRECTORY

CREATING A DATA FILE CONNECTION

The A$CREATE$FILE system call returns a connection to a physical, stream, or named data file. If the file does not yet exist, this call also creates the file. If the file already exists, several options are available to the caller, as detailed in Chapter 8.

ATTACHING A FILE

The A$ATTACH$FILE system call creates a connection to an existing file. Once the connection is formed, it remains in existence until it is deleted, or until the creating task is deleted.

CREATING A DIRECTORY FILE

The A$CREATE$DIRECTORY system call applies to named directory files only. This call creates a new directory file and returns a connection to that file.

CALLS THAT MODIFY FILE CONNECTIONS

The asychronous level has two calls that modify file connections:

- A$CHANGE$ACCESS
- A$RENAME$FILE

CHANGING FILE ACCESS RIGHTS

The A$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file.

RENAMING A FILE

The A$RENAME$FILE system call applies to named files only. It is called to change the name of a file. A renamed data file can also be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent, but a directory file can only be renamed within its parent directory.

CALLS THAT OBTAIN FILE INFORMATION

The asynchronous level has four calls that obtain status and attribute information about files and their connections:

- AGETCONNECTION$STATUS
- AGETFILE$STATUS
- AGETDIRECTORY$ENTRY
- AGETPATH$COMPONENT

GETTING CONNECTION STATUS DATA

The AGETCONNECTION$STATUS system call returns information on the current status of one specific connection to a file.

GETTING FILE STATUS DATA

The AGETFILE$STATUS system call returns status and attribute information about a specific file and its connections. The form of the information differs for physical, stream, and named files.

GETTING DIRECTORY CONTENTS

The AGETDIRECTORY$ENTRY system call applies to named files only.

Entries in a directory are numbered sequentially starting from zero. By specifying an entry number in a call to AGETDIRECTORY$ENTRY, the caller can obtain the name of the file associated with that entry.

GETTING A PATH COMPONENT

The AGETPATH$COMPONENT system call is meaningful for named files only. The user who knows the token for a file connection can specify this token to AGETPATH$COMPONENT and receive the name of the file. This is the name by which it is cataloged in its parent directory.

If the specified token belongs to the root directory of a directory tree, a null string is returned because a root directory has no parent. A null string is also returned if the token for a physical or stream file is specified.

CALLS THAT PERFORM FILE I/O

Calls that perform file I/O are valid only for data files (not for directory file connections). Five such calls are available at the asynchronous level.

- A$OPEN
- A$SEEK
- A$READ
- A$WRITE
- A$CLOSE

Buffers used in read/write operations must be in a segment allocated by the free-space manager of the RMX/86 nucleus (that is, the segment must be allocated dynamically).

OPENING A DATA-FILE CONNECTION

The A$OPEN system call opens a file connection for input/output and specifies who, if anyone, may share the connection (readers, writers, or both). Opening a file connection also establishes a file pointer set to byte position zero. A$SEEK, A$READ, and A$WRITE all move this pointer.

MOVING THE FILE POINTER

The A$SEEK system call applies to physical and named data files only. It is called to move the file pointer for an open connection, thus allowing file data to be accessed randomly. The file pointer can be placed at any byte position in the file.

READING A FILE

The A$Read sytem call initiates reading via an open file connection. The connection is read as a string of bytes, and any number of bytes can be requested. The bytes are read starting at the current setting of the file pointer. Following the read operation, the file pointer is positioned just past the last byte read.

WRITING A FILE

The A$WRITE system call initiates a write operation from a user buffer into a connected file. The data is written beginning at the current setting of the file pointer. Following the write operation, the file pointer is positioned just after the last byte written.

CLOSING A DATA-FILE CONNECTION

The A$CLOSE system call is invoked to close an open connection when I/O operations are completed. A closed connection can be reopened without attaching the file again.

A CALL TO PERFORM A DEVICE-LEVEL FUNCTION

The asynchronous level includes an A$SPECIAL system call to perform special device-level functions. This call applies to physical files only.

The I/O system supports a number of functions at the device level (namely read, write, seek, attach device, detach device, open, and close). The A$SPECIAL system call allows the user to perform additional device-driver functions. For example, a rewind function would be desirable for a magnetic tape driver or a track formatting function for a random-access device.

CALLS THAT DELETE INFORMATION, FILES, AND CONNECTIONS

The asynchronous level has three calls that delete files (or part of a file) and connections.

- A$DELETE$FILE
- A$TRUNCATE
- A$DELETE$CONNECTION

DELETING A FILE

The A$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file is not actually deleted, however, until all connections to the file have been severed. Directory files cannot be deleted unless they are empty.

TRUNCATING A FILE

The A$TRUNCATE system call applies to named data files only. It truncates a file by freeing all allocated bytes beyond the current setting of the file pointer. A seek operation can be used to position the file pointer before the call to A$TRUNCATE. Truncation is performed immediately. If the file pointer is positioned at or beyond the end-of-file, no operation is performed.

DELETING A CONNECTION

The A$DELETE$CONNECTION system call severs a file connection established by A$CREATE$FILE, A$ATTACH$FILE , or A$CREATE$-DIRECTORY. If a connection is open when this call is made, the connection is closed before being severed.

A$DELETE$CONNECTION also deletes the file associated with the specified connection if the file is both marked for deletion (by a previous call to A$DELETE$FILE) and the specified connection is the last remaining connection to the file.

Chapter 3. HYBRID INPUT/OUTPUT

The hybrid level of I/O system calls is an extension of the asynchronous level described in Chapter 2. At this level, calls that create and delete files and connections, calls that modify file attributes, and certain status calls have synchronous interfaces. This means the programmer does not have to synchronize these operations himself. The calling sequences are simplified also. Logical names can be specified for path prefixes, default user objects are assumed, and no response mailbox need be specified since the programmer does no synchronizing. These synchronous system calls require more system memory than their asynchronous counterparts, however.

File connections created at the hybrid level are fully compatible with those created by asynchronous system calls. This allows the hybrid-level user to invoke asynchronous calls for file I/O and related operations, thereby retaining most of the flexibility of asynchronous input/output.

This chapter provides an overview of hybrid system calls. The detailed PL/M calling sequences and descriptions of these calls can be found in Chapter 8. Not every system call is applicable for every kind of file, as indicated in the following summaries.

CALLS THAT CREATE FILE CONNECTIONS

Three system calls are available for creating file connections at the hybrid level:

- H$CREATE$FILE
- H$ATTACH$FILE
- H$CREATE$DIRECTORY

CREATING A DATA FILE CONNECTION

The H$CREATE$FILE system call returns a connection to a physical, stream, or named data file. The connection can also be given a logical name, under which it is cataloged in the job's logical-name directory.

If the file does not yet exist, this call also creates a new file. If the file already exists, several options are available to the caller, as detailed in Chapter 8.

ATTACHING A FILE

The H$ATTACH$FILE system call creates a connection to an existing file. The connection can also be given a logical name, under which it is cataloged in the job's logical-name directory.

CREATING A DIRECTORY FILE

The H$CREATE$DIRECTORY system call applies to named directory files only. This call creates a new directory file and returns

a connection to that file. The connection can also be given a logical name, under which it is cataloged in the job's logical-name directory.

CALLS THAT MODIFY FILE ATTRIBUTES

The hybrid level has two calls that modify file attributes:

- H$CHANGE$ACCESS
- H$RENAME$FILE

CHANGING FILE ACCESS RIGHTS

The H$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file.

RENAMING A FILE

The H$RENAME$FILE system call applies to named files only. It is called to change the name of a file. The name changed is the subpath name cataloged in the file's parent directory, not the logical name cataloged in the job's logical-name directory.

A renamed data file can be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent. A directory file can only be renamed within its parent directory.

CALLS THAT OBTAIN FILE INFORMATION

The hybrid level has two calls that obtain status and attribute information about files and their connections:

- HGETFILE$STATUS
- H$LOOK$UP$CONNECTION

GETTING FILE STATUS DATA

The HGETFILE$STATUS system call returns status and attribute information about a specific file and its connections. The form of the information differs for physical, stream, and named files.

GETTING THE TOKEN FOR A CONNECTION

The H$LOOK$UP$CONNECTION system call returns the token for the file connection associated with a specified logical name.

CALLS THAT DELETE FILES AND CONNECTIONS

The hybrid level has two system calls that delete files and connections:

- H$DELETE$FILE
- H$DELETE$CONNECTION

DELETING A FILE

The H$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file is not actually deleted, however, until all connections to the file have been severed. Directory files cannot be deleted unless they are empty.

DELETING A CONNECTION

The H$DELETE$CONNECTION system call severs a file connection established by H$CREATE$FILE, H$ATTACH$FILE, or H$CREATE$-DIRECTORY. If the connection is open when this call is made, the connection is closed before being severed. The logical name for the connection, if one exists, is removed from the job's logical-name directory.

H$DELETE$CONNECTION also deletes the file associated with the specified connection if the file is both marked for deletion (by a previous call to H$DELETE$FILE) and the specified connection is the last remaining connection to the file.

ASYNCHRONOUS CALLS USED AT THE HYBRID LEVEL

The hybrid level uses many of the asynchronous system calls described in Chapter 2. The calls operate on established file connections to perform file I/O, obtain connection status or attribute information, or perform special device-level functions.

Ten asynchronous calls are supported at the hybrid level:

- A$OPEN
- A$CLOSE
- A$SEEK
- A$READ
- A$WRITE
- A$TRUNCATE
- A$SPECIAL
- AGETDIRECTORY$ENTRY
- AGETPATH$COMPONENT
- AGETCONNECTION$STATUS

These calls perform the same operations described in Chapter 2 (and in detail in Chapter 8).

Chapter 4. SYNCHRONOUS INPUT/OUTPUT

Synchronous operations represent the highest level of RMX/86 I/O. All calls at this level are synchronized automatically. PL/M calling sequences are simplified compared to asynchronous calls. Logical names can be specified for path prefixes, default user objects are assumed, and no response mailboxes need be specified since the programmer does no synchronizing.

Synchronous calls that create or open a data-file connection can also specify up to two buffers as part of that connection. These synchronous I/O system (SIOS) buffers allow overlapping of read and write operations. Data can be read into a buffer in anticipation of task requirements, or can be written from a buffer while the task is preparing additional output. When two SIOS buffers are used, the I/O system can begin the next read before processing on the current buffer is completed; similarly, it can begin writing out a full buffer while the other buffer is being filled. This makes the synchronous level especially efficient at processing sequential data.

As these features indicate, the synchronous level provides the most convenient tool for I/O processing of the three levels available, but it also requires the most system memory.

This chapter provides an overview of synchronous system calls. The detailed PL/M calling sequences and descriptions of these calls can be found in Chapter 8. Not every system call is applicable to every kind of file, as indicated in the following summaries.

CALLS THAT CREATE FILE CONNECTIONS

Three system calls are available for creating file connections at the synchronous level:

- S$CREATE$FILE
- S$ATTACH$FILE
- S$CREATE$DIRECTORY

CREATING A DATA FILE CONNECTION

The S$CREATE$FILE system call returns a connection to a physical, stream, or named data file. This composite connection includes any SIOS buffers requested in the call. The buffer size specified in this call is the default buffer size, which can be overridden by S$OPEN when the connection is opened. The connection returned by S$CREATE$FILE can also be given a logical name and cataloged in the job's logical-name directory under this name.

If the designated file does not yet exist, this call also creates the file. If the file already exists, several options are available to the caller, as detailed in Chapter 8.

ATTACHING A FILE

The S$ATTACH$FILE system call creates a connection to an existing file, including any desired SIOS buffers. As with S$CREATE$FILE, the buffer size specified is the default size, which can be overridden by S$OPEN.

The connection returned by this call can also be given a logical name, under which it is cataloged in the job's logical-name directory.

CREATING A DIRECTORY FILE

The S$CREATE$DIRECTORY system call applies to named directory files only. This call creates a new directory file and returns a connection to that file. The connection can also be given a logical name, under which it is cataloged in the job's logical-name directory.

CALLS THAT MODIFY FILE ATTRIBUTES

The synchronous level has two calls that modify file attributes:

- S$CHANGE$ACCESS
- S$RENAME$FILE

CHANGING FILE ACCESS RIGHTS

The S$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file.

RENAMING A FILE

The S$RENAME$FILE system call applies to named files only. It is called to change the name of a file. The name changed is the subpath name cataloged in the file's parent directory, not the logical name cataloged in the job's logical-name directory.

A renamed data file can also be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent, but a directory file can only be renamed within its parent directory.

CALLS THAT OBTAIN FILE INFORMATION

The synchronous level has three calls that obtain status and attribute information about files and their connections.

- SGETCONNECTION$STATUS
- SGETFILE$STATUS
- S$LOOK$UP$CONNECTION

GETTING CONNECTION STATUS DATA

The SGETCONNECTION$STATUS system call returns information on the current status of one specific connection to a file.

GETTING FILE STATUS DATA

The SGETFILE$STATUS system call returns status and attribute information about a specific file and its connections. The form of the information differs for physical, stream, and named files.

GETTING THE TOKEN FOR A CONNECTION

The S$LOOK$UP$CONNECTION system call returns the token for the file connection associated with a specified logical name.

CALLS THAT PERFORM FILE I/O

Calls that perform file I/O are valid only for data files (not for directory file connections). Seven such calls are available at the synchronous level:

- S$OPEN
- S$SEEK
- S$READ$MOVE
- S$READ$LOCATE
- S$WRITE$UPDATE
- S$CLOSE

OPENING A DATA-FILE CONNECTION

The S$OPEN system call opens a file connection for input/output and specifies who may share the connection (readers, writers, or both). Opening a file also establishes a file pointer set to byte position zero and, if the file is being opened for reading, initiates the first read operation.

S$OPEN can also request up to two SIOS buffers for the open connection and specify their size, if the size differs from that specified when the connection was created.

MOVING THE FILE POINTER

The S$SEEK system call applies to physical and named data files only. It is called to move the file pointer for an open connection, thus allowing file data to be accessed randomly. The file pointer can be placed at any byte position in the file.

READING TO A CALLER BUFFER

The S$READ$MOVE system call moves a collection of bytes from a file to a specified caller buffer.

Reading to an SIOS Buffer

The S$READ$LOCATE system call reads a collection of bytes from a designated file to an SIOS buffer. When the read operation has been completed, S$READ$LOCATE returns the location of the buffer.

WRITING FROM A CALLER BUFFER

The S$WRITE$MOVE system call writes a collection of bytes from a designated caller buffer to a file.

UPDATING A FILE

As mentioned above, S$READ$LOCATE returns the location of the data it reads (that is, a pointer to an SIOS buffer). The S$WRITE$UPDATE system call references this pointer to write the data back to its original location after it has been updated. A call to S$READ$LOCATE must be the most recent operation on the connection addressed by S$WRITE$UPDATE.

CLOSING A DATA-FILE CONNECTION

The S$CLOSE system call closes an open connection. Before closing the connection, this call waits for all I/O operations in progress on the file to be completed, makes sure that all data in the buffer of an output file is written, and releases the SIOS buffers created by S$OPEN.

A CALL TO PERFORM A DEVICE-LEVEL FUNCTION

The synchronous level includes an S$SPECIAL system call to perform special device-level functions. This call applies to physical files only.

The I/O system supports a number of functions at the device level (namely read, write, seek, attach device, detach device, open, and close). The S$SPECIAL system call allows the user to perform additional device-driver functions. For example, a rewind function would be desirable for a magnetic tape driver or a track formatting function for a random-access device.

CALLS THAT DELETE FILES AND CONNECTIONS

The synchronous level has three calls that delete files (or part of a file) and connections:

- S$DELETE$FILE
- S$TRUNCATE$FILE
- S$DELETE$CONNECTION

DELETING A FILE

The S$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file is not actually deleted, however, until all connections

to the file have been severed. Directory files cannot be deleted unless they are empty.

TRUNCATING A FILE

The S$TRUNCATE$FILE system call applies to named data files only. It truncates a file by freeing all allocated bytes beyond the current setting of the file pointer. A seek operation can be used to position the file pointer before the call to the S$TRUNCATE$FILE. Truncation is performed immediately. If the file pointer is positioned at or beyond the end-of-file, no operation is performed.

DELETING A DATA-FILE CONNECTION

The S$DELETE$CONNECTION system call severs a file connection established by S$CREATE$FILE or S$ATTACH$FILE. If the connection is open when S$DELETE$CONNECTION is called, it is closed before being severed. The logical name for the connection, if one exists, is removed from the job's logical name directory.

S$DELETE$CONNECTION also deletes the file associated with the specified connection if the file is both marked for deletion (by a previous call to S$DELETE$FILE) and the specified connection is the last remaining connection to the file.

Chapter 5. THE JOB ENVIRONMENT

A job is an environment for tasks. It includes not only the task(s) to be executed, but also the resources needed by the tasks(s). These include:

- memory
- exchanges (mailboxes and semaphores)
- directories
- other objects, such as connections

Each of these job components is described in the RMX/86 Nucleus, Terminal Handler, and Debugger Reference Manual.

The six job-related system calls described in this chapter fall into two categories:

- system calls that create or terminate jobs
- system calls that set or inspect the default user and prefix for a job

CREATING AND TERMINATING JOBS

Once a job exists, its tasks can create other jobs. These jobs become the children of the creating (parent) job.

In the I/O system, jobs are created by invoking the CREATEIOJOB system call. This call creates both a job and its first task. It also includes several parameters useful for I/O processing. These parameters can specify a default user, a default path prefix, and a pointer to a list of logical names to be used by the job. CREATEIOJOB also specifies a mailbox to be used for parent-child job communication.

The EXITIOJOB system call causes the calling task to terminate and notifies the parent job through the mailbox established by CREATEIOJOB.

EXITIOJOB performs the following operations:

- deletes all connections and detaches all logical devices attached by the job;
- creates a segment containing the exit message;
- sends the exit message to the mailbox provided by the parent job;
- deletes the calling task.

The parent job can then delete the current job, if it wishes to do so.

DEFAULT USER

When a named file is created, its creator is identified as the owner of the file.

The token for the owner's user object is a required parameter in many asynchronous system calls. The programmer can specify a default user for the entire job, however, simplifying parameter specification considerably. Once the default has been established asynchronous calls merely specify a zero wherever the user parameter is required. Hybrid and synchronous system calls simplify user specification even further by always assuming the default user.

We mentioned above that CREATEIOJOB can be used to specify a default user at the time a job is created. The SET$DEFAULT$USER system call can be invoked to establish a default user for the current job, or another job that already exists.

A given job's default user can be determined at any time by calling GET$DEFAULT$USER.

DEFAULT PREFIX

Many of the system calls described in Chapters 2-4 require a path prefix as a parameter (even if a subpath is not needed). Asynchronous calls require the token for a device connection or a file connection as a prefix. Hybrid and synchronous calls accept a logical name as the prefix designator.

Prefix specification can be simplified by establishing a default prefix for the job. Once the default is established, asynchronous calls need only specify a zero to designate the prefix parameter. Hybrid and synchronous calls can omit the path prefix entirely, in which case the default is assumed.

Setting a default prefix also locates a job within a directory tree. Subsequent references to files used by the job occur relative to the default directory associated with the job.

When a job is created, the default prefix can be set by a parameter of the CREATEIOJOB system call. The SET$DEFAULT$PREFIX system call can be invoked to establish the default prefix for the current job, or for another job that already exists.

A given job's default prefix can be determined at any time by calling GET$DEFAULT$PREFIX.

Chapter 6. TIME AND DATE FUNCTIONS

The time and date functions are part of the I/O system, but they are also separately configurable, for applications that may want to use them without the I/O system.

The time/date is maintained as a 32-bit counter containing the number of seconds since midnight, January 1, 1978. The I/O system supports two system calls for accessing this counter:

- GET$TIME
- GET$TIME$STRING

GET$TIME returns the time/date value as it is stored internally (that is, as the total seconds since January 1, 1978).

GET$TIME$STRING returns this information in more readable form. The time is returned in the format.

HH:MM:SS (hour:minute:second)

and the date in the format

MM/DD/YY (month/day/year)

Chapter 7. RMX/86 LOADER

The RMX/86 loader utility is called to load an object file into memory.

The loader can also be instructed to create a new job and in addition, can create a task to execute the code from the object file once loading is complete.

Object files must be located by LOC86 before they can be loaded. That is, load addresses must be specified as absolute memory addresses.

LOADER OPERATION

The system call invoked to load object files is S$LOAD (Synchronous load). This call can perform any of the following function combinations:

- Create a job, load a file, and create a task;
- Create a job and load a file;
- Load a file and create a task;
- Load a file.

CREATING A JOB

When a caller asks the loader to create a job, the loader invokes the CREATEIOJOB system call (described in Chapter 5). This gives the loaded task the ability to call EXITIOJOB to terminate itself when finished.

By creating a job for the loaded task, the caller can control the minimum and maximum memory used by tasks in the job. These two parameters are specified as part of the call to the loader.

PACKAGE OBJECT

The loader creates a special package object to simplify passing results returned by the load call. The package object allows several other objects' tokens to be "packaged" and returned as a single object, the token for the package.

To expand the contents of a package object, the user must specify its token to the INSPECT$PACKAGE system call. This call returns a set of tokens for the package object.

The DELETE$PACKAGE system call can be invoked to delete a package object.

Chapter 8. INVOKING I/O SYSTEM CALLS IN PL/M

This chapter describes the PL/M calling sequences to RMX/86 I/O system calls. The list is limited to calls that can be invoked from application programs. I/O-related calls reserved for system programmers (that is, calls that can affect an entire system) are described in the RMX/86 System Programmer's Reference Manual.

The system calls are listed here alphabetically by the same shorthand notation used throughout this manual. For example, S$DELETE$FILE refers to the synchronous-level delete-file system call and appears alphabetically before SET$DEFAULT$PREFIX. This notation is language independent and should not be confused with the actual form of the PL/M call. The precise format of each call is spelled out as part of its detailed description.

SYNTAX NOTATION

RMX/86 I/O operations are declared as typed or untyped external procedures by PL/M. These procedure declarations are shown in Appendix B. PL/M performs I/O operations by issuing external procedure calls.

Certain conventions are used in this chapter to describe the PL/M calling sequences. Calls can be entered using either upper-cased or lower-cased characters, but in this chapter PL/M reserved words (such as CALL) and procedure names (such as RQAREAD) are shown in capital letters. Lower-cased items represent input parameters to be replaced with actual values when the I/O procedure is called. Input parameters are separated by commas and the entire parameter list is enclosed in parentheses. A lower-cased item to the left of an equal (=) sign in a typed procedure call represents a value returned by the call. Procedure calls are terminated by a semicolon.

EXAMPLES

1. connection = RQHCREATE$FILE (log$name, path):

   This sequence calls the hybrid-level create-file typed procedure, specifying as input parameters a logical name for the connection being created and the path to the new file. The call returns a new file connection to the caller.

2. CALL RQSTRUNCATE$FILE (connection):

   This sequence calls the synchronous-level truncate-file untyped procedure, specifying a connection to the file being truncated as its input parameter.

INPUT PARAMETER SPECIFICATION

USER PARAMETER

This parameter is specified only in asynchronous sytem calls. It contains a 16-bit token designating the caller's user object. A zero specification designates the default user. Hybrid and synchronous system calls always assume the default user.

LOGICAL-NAME PARAMETER

This parameter is specified only in hybrid and synchronous system calls. It designates the logical name under which a new file connection is to be cataloged in the job's logical-name directory. A logical name can be 1-12 ASCII characters, including any printable ASCII character except the colon (:), slash (/), and up-arrow (↑) or cirumflex (^). This parameter can also be specified as a zero, in which case the new file connection is not cataloged in the logical-name directory.

FILE-PATH PARAMETER(S)

Files are designated in system calls by specifying their path, that is, their prefix and subpath. For named files, the prefix designates the starting point in a directory-tree scan and the subpath describes the rest of the route through the tree to the desired file. Since physical and stream files are not nodes in a directory tree, the subpath parameter has no meaning for them. The prefix designates the desired connection directly for these files.

Path Specification in Asynchronous Calls.

Asynchronous system calls specify paths using two parameters. The prefix parameter is a token designating an existing device connection or file connection. If this parameter is zero, the default prefix is assumed.

For named files, the subpath parameter is a pointer to an ASCII string. The form of this string is described below. The subpath can also be zero or can point to a null string, in which case the prefix points directly to the desired connection. For physical and stream files, the subpath parameter is always specified as zero.

Asynchronous system calls can specify paths in the following forms:

Prefix	Subpath	Meaning
Ø	Ø	Default prefix is desired connection.
Token	Ø	Token points to desired connection.
Token	Pointer to Null String	Token points to desired connection.
Token	Pointer to ASCII String	Token plus ASCII string lead to desired connection.

The subpath ASCII string is a list of file names separated by slashes, terminating with the desired file. A file name can be 1-14 ASCII characters, including any printable ASCII character except the slash (/) and up-arrow (↑) or circumflex (^). In Figure 8-1, for example, if the prefix is the token for directory OBSTETRICS and we wish to reference file OUT_PATIENT, the subpath parameter must point to the string

DELIVERY/POST_PARTUM/OUT_PATIENT

If the ASCII string begins with a slash, the prefix merely designates the tree and the subpath is assumed to start at the root directory of the tree associated with the prefix. For example, if the prefix designates directory GYNECOLOGY in Figure 8-1, the subpath to OUT$PATIENT is

/OBSTETRICS/DELIVERY/POST_PARTUM/OUT_PATIENT

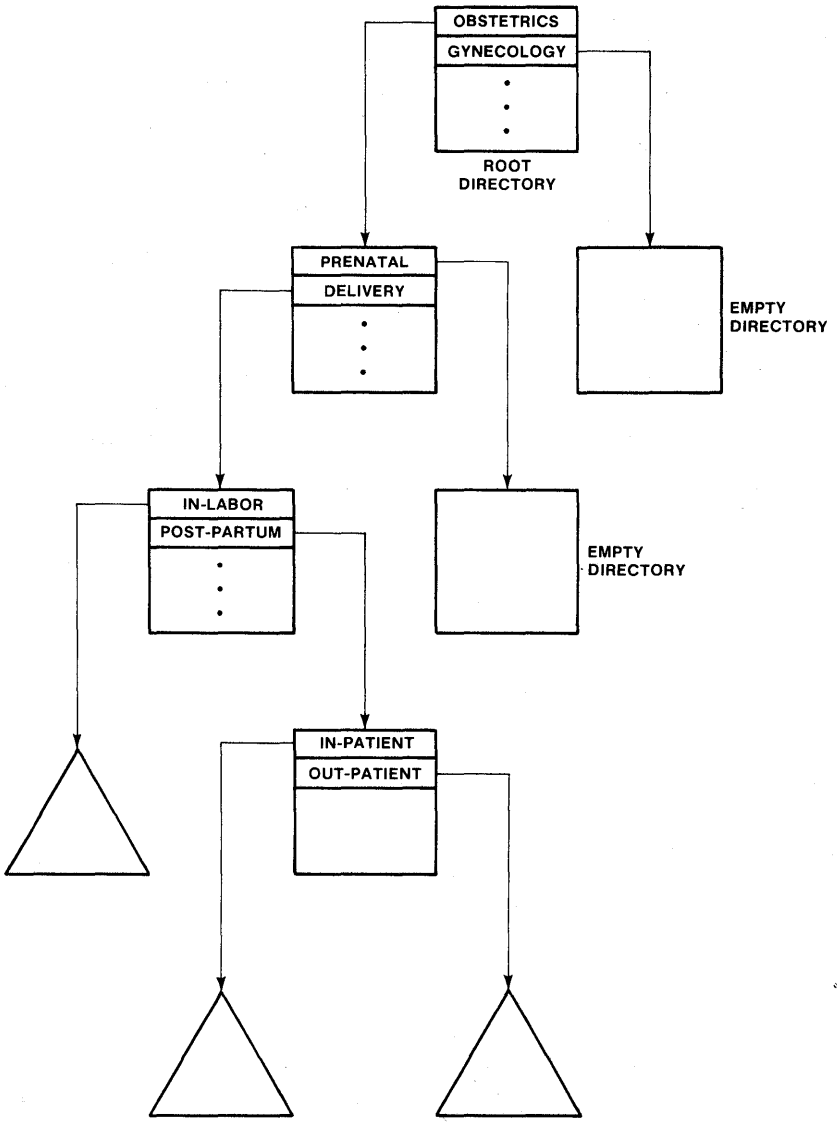


Figure 8-1. Sample Directory Tree (Asynchronous Call)

Files can also be addressed relative to other files in the tree, using "↑" as a path component. The "↑" refers to the parent directory of the current file in the path scan. For example, now that we have a connection to OUT_PATIENT in Figure 8-1, we can use that connection to specify a subpath to IN PATIENT. With the token for the OUT_PATIENT connection as our prefix, the subpath string would be

↑IN_PATIENT

Note that no slash follows the "↑" in this example; "↑" itself can be a separator.

Of course an even simpler approach would be to designate directory POST_PARTUM as the prefix, in which case the ASCII string becomes

IN_PATIENT

Path Specifications in Hybrid/Synchronous Calls

Hybrid and synchronous system calls specify the path as a single parameter, a pointer to an ASCII string. The ASCII string encompasses both the prefix and subpath.

The prefix can be specified using the logical name of the designated connection. The logical name must be enclosed in colons when it appears in the path string. If no name is specified, the default prefix is assumed.

The subpath is a list of file names separated by slashes, as defined for asynchronous calls above. A slash at the beginning of this list designates the root directory associated with the prefix. The up arrow (↑) character can be used to designate the parent directory of the current file in the scan.

The ASCII string addressed in hybrid and synchronous system calls can take any of the following forms:

ASCII String	Meaning
null	Default prefix is desired connection.
:log-name:	Name designates desired connection.
subpath	Tree scan starts at default prefix.
/subpath	Tree scan starts at root directory associated with default prefix.
:log$name:/subpath	Tree scan starts at root directory associated with file designated by logical name.
:log$name:subpath	Tree scan starts at file designated by logical name.

Figure 8-2 is the same as Figure 8-1, except that logical names (shown enclosed in colons) have been added for each directory and data file. Referring to this figure, file OUT_PATIENT might be addressed by any of the following strings (assuming OBSTETRICS as the default prefix):

```
:HOME:                  (assuming this connection already exists)
DELIVERY/POST_PARTUM/OUT_PATIENT
/OBSTETRICS/DELIVERY/POST_PARTUM/OUT_PATIENT
:GYN:/OBSTETRICS/DELIVERY/POST_PARTUM/OUT_PATIENT
:POST:OUT_PATIENT
:HOSP: ↑ OUT_PATIENT
```

RESPONSE MAILBOX PARAMETER

This parameter is specified only in asynchronous system calls. It contains a token designating the mailbox that is to receive the result of the call. This information is provided by tasks to synchronize parallel operations. The I/O result segment returned to this mailbox is described in Appendix C.

NOTE

Result information should be deleted once it is no longer needed. Otherwise, it will consume available memory.

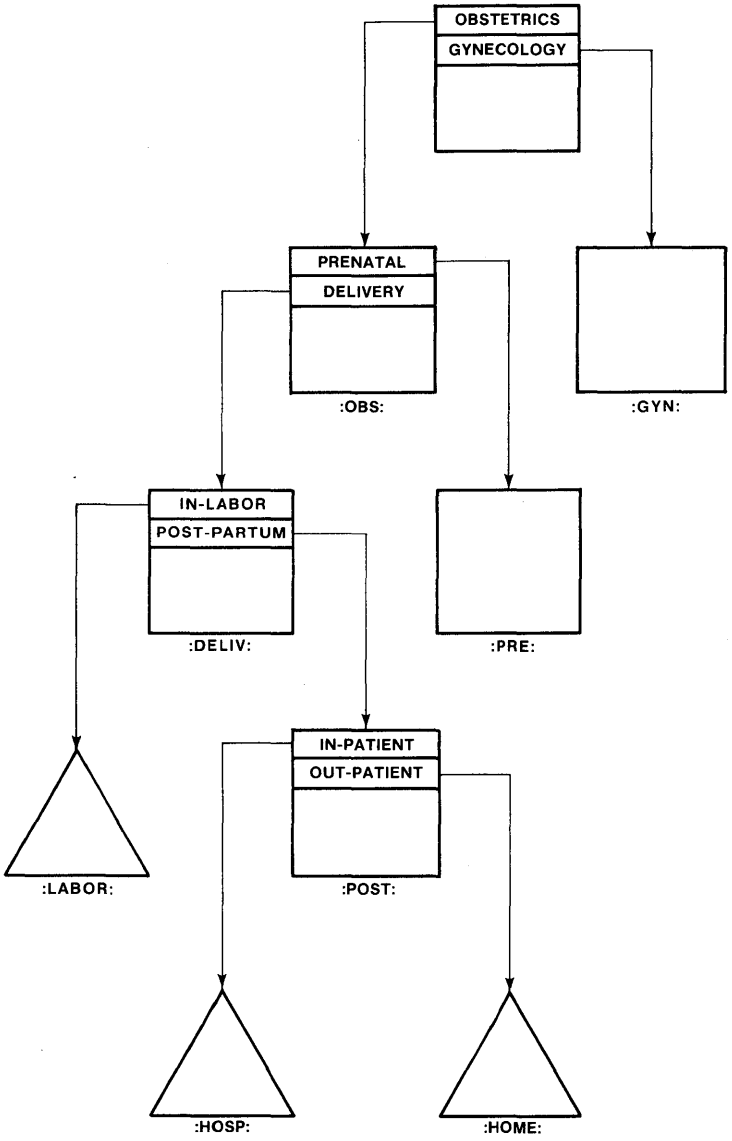


Figure 8-2. Sample Directory Tree (Synchronous Call)

SPECIAL OBJECTS AND DATA TYPES

Each system call explains in detail the expected contents of its input parameters. These explanations frequently refer to the following PL/M data types and RMX/86 special objects.

BYTE is an 8-bit unsigned binary number ranging from 0 to 255 and occupying one byte of memory.

BOOLEAN is a single-BYTE quantity taking the value TRUE (ØFFH) or FALSE (ØØH).

STRING is a sequence of BYTES, the first of which contains the number of BYTES in the sequence (not including the first, or length, BYTE). A length of zero specifies a null string.

WORD is a 16-bit unsigned binary number ranging from 0 to 65535 and occupying two contiguous BYTES of memory (with the loworder BYTE first).

TOKEN is two contiguous BYTES of memory whose contents are specified to obtain an RMX/86 object.

CONNECTION is a TOKEN for a connection object.

USER is a TOKEN for a user object.

POINTER is two WORDS of memory containing an OFFSET WORD and a segment TOKEN (base), respectively.

CONDITION CODES

The I/O system issues a condition code when a system call is invoked. If the call executes normally, the I/O system returns the code "E$OK." If an error is encountered, an exceptional condition occurs.

When an exceptional condition occurs, the system issues a condition code describing the error, then either returns to the calling task or passes control to an exception handler. See the RMX/86 Nucleus, Terminal Handler and Debugger Reference Manual for a detailed description of exception handling.

Exceptional conditions can be programming errors, meaning they can be corrected by recoding, or environmental conditions, meaning they can occur even after a program has been completely debugged. These exceptional conditions are detected synchronously with system call invocation (for example, detection of an invalid parameter). The condition code is returned to the location addressed by the "excep$ptr" field of the external procedure declaration associated with each system call (see Appendix B).

At the asynchronous level, exceptional conditions can also be detected asynchronously (for example, when the I/O system cannot know if a file exists before trying to attach it). Condition codes describing asynchronous exceptional conditions are returned in the "status" field of the I/O result segment (Appendix C). The I/O result segment is then sent to the mailbox designated by the "resp$mbox" parameter in asynchronous calls.

Each of the system call descriptions in this chapter lists the condition codes liable to result from invoking that call. An explanation of each exceptional-condition code and the probable reasons for its occurrence can be found in Appendix D. Asynchronous exceptional-condition codes are explained in Appendix C.

SYSTEM CALLS

The following pages provide a detailed description of each I/O system call, listed alphabetically. Appendix A provides a summary of these calls, grouped by function and correlated to the file types to which they apply. That summary also acts as a cross-reference to the following detailed descriptions.

A$ATTACH$FILE

ASYNCHRONOUS ATTACH$FILE SYSTEM CALL

The A$ATTACH$FILE system call creates a connection to an existing file. Once the connection is established, it remains in effect until it is detached (see A$DELETE$CONNECTION), or until the creating job is deleted (see EXITIOJOB). Once attached, the file may be opened, closed, read, written, etc., as many times as desired.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
CALL  RQ$A$ATTACH$FILE(user, prefix, subpath,
                       resp$mbox, excep$ptr);
```

INPUT PARAMETERS

user	is a token for the user object to be inspected in any access checking that takes place. A zero specifies the default user for the job. This parameter is ignored for physical and stream files.
prefix	is a token for the connection object to be used as the path prefix. Normally, this will be a connection to an existing file (followed by a null subpath). A zero specifies the default prefix for the job.
subpath	is a pointer to the string containing the subpath of the file to be attached. A null string indicates that the new connection is the file designated by the prefix: the new connection will not be open, regardless of the open state of the prefix.
resp$mbox	is a token for the mailbox that receives the result of the call.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is a token for the new connection if the call succeeded; an I/O result segment (Appendix C) is returned otherwise.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FNEXIST, E$FTYPE, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

A$CHANGE$ACCESS

ASYNCHRONOUS CHANGE$ACCESS SYSTEM CALL

A$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file. Depending on the contents of the "id" and "access" parameters specified in the system call, users may be added to or deleted from the files's ID-access list, or the access privileges granted to a particular user may be changed.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
CALL  RQ$A$CHANGE$ACCESS(user, prefix, subpath, id,
                         access, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

user — is a token for the user object to be inspected in access checking. A value of zero specifies the default user for the job.

prefix — is a token for the connection to be used as the path prefix. Typically, this would be a connection to the file whose access is being changed (followed by a null subpath). A zero specifies the default prefix for the job.

subpath — is a pointer to the string giving the subpath from the prefix to the file whose access is to be changed. A null string indicates the connection is to the file designated by the prefix. In this case, the user parameter is not used, since access rights are already frozen into the connection.

id — is a word giving the ID number of the user whose access is to be changed. If this ID does not already exist in the ID-access list, it is added. This list may contain a total of three ID-access pairs.

access — is a byte mask giving the new access rights for the ID. If a bit is set to one, the corresponding access is granted. For a named data file, the possible bit settings are:

INPUT PARAMETER
access (continued)

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

For a named directory file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

If zero is specified for the access parameter (that is, no access), the ID specified in the ID parameter is deleted from the file's ID-access list.

resp$mbox — is a token for the mailbox that receives an I/O result segment indicating completion of the access change. A zero value for this parameter indicates that no response is wanted.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result — is an I/O result segment indicating completion of the access change. See Appendix C.

FILE ACCESS REQUIREMENTS

The caller must be the owner of the file or must have change-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

A$CLOSE

ASYNCHRONOUS CLOSE SYSTEM CALL

The A$CLOSE system call closes an open file connection. It is called between uses of a file. A file connection must also be closed if the user wishes to change the open mode or shared status of the connection.

```
CALL  RQ$A$CLOSE(connection, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for a file connection to be closed.
resp$mbox	is a token for a mailbox that receives an I/O result segment indicating completion of the operation. A zero value for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating completion of the close operation.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

ASYNCHRONOUS CREATE$DIRECTORY SYSTEM CALL

The A$CREATE$DIRECTORY system call is applicable to named directory files only. When called, it creates a new directory file and returns a token for the new file connection. This system call cannot be used to create a connection to an existing directory.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
CALL  RQ$A$CREATE$DIRECTORY(user, prefix, subpath, access,
                            resp$mbox, excep$ptr);
```

INPUT PARAMETERS

user — is a token for the user object of the new directory's owner. It is also inspected to make sure the caller has proper access to the new directory's parent. If zero is specified, the default user for the job is assumed.

prefix — is a token for the connection to be used as the path prefix. A zero specifies the default prefix for the job.

subpath — is a pointer to the string giving the subpath of the directory to be created. The subpath string must not be null, and must point to a location in the directory tree where a file has not yet been defined.

access — is a byte mask giving the owner's initial access rights to the directory. For each bit in the mask, a one grants access and a zero denies it. The possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

resp$mbox — is a token for the mailbox that receives the result of this call.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

A$CREATE$DIRECTORY (continued)

RESULT SEGMENT

result	is a token for the directory file connection if the call succeeded; otherwise, an I/O result segment (Appendix C) is returned.

FILE ACCESS REQUIREMENT

The caller must have add-entry access to the parent of the new directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$SPACE, E$TYPE.

ASYNCHRONOUS CREATE$FILE SYSTEM CALL

The A$CREATE$FILE system call creates a physical, stream, or named data file and returns a token for the new file connection. If a named file designated by the prefix and subpath parameters already exists, one of the following situations occurs:

- If the "must$create" parameter is TRUE, an error condition code (E$FEXIST) is returned.
- If the "must$create" parameter is FALSE and the path designates a data file, a new connection to that file is returned (that is, A$CREATE$FILE acts like A$ATTACH$FILE). In this case, the file is truncated to zero length after being attached and any data in the file is lost. If the "size" parameter is nonzero, the requested space is subsequently allocated.
- If the "must$create" parameter is FALSE and the path designates a device or directory, an unamed file is created on the corresponding device. This file is deleted automatically when the last connection to it is deleted.

Many of the parameters specified in the A$CREATE$FILE call do not apply to physical and stream files. In these cases, the parameter is ignored.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
CALL  RQ$A$CREATE$FILE(user, prefix, subpath, access,
                       granularity, high$size low$size,
                       must$create, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

user — applies to named files only and is a token for the user object of the file's owner. It also furnishes the user ID for any access checking that might occur. A zero specifies the default user for the job. This parameter is ignored for physical and stream files.

prefix — is a token for a device or file connection. By implication, this parameter indicates the type of file (physical, stream, named) being created. For a stream file, the prefix is a device connection. In the case of a named file, the prefix acts as the starting point in a directory tree scan. A zero specifies the default prefix for the job.

A$CREATE$FILE (continued)

INPUT PARAMETERS (continued)

subpath — applies to named files only and is a pointer to the string giving the subpath for the file being created.

access — applies to named files only and is a byte mask giving the owner's initial access rights to the new file. For each bit, a one grants access and a zero denies it. The possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

granularity — applies to named files only and is a word giving the granularity of the file being created. This is the size (in bytes) of each logical block to be allocated to the file. The value specified in this parameter is rounded up to a multiple of the volume granularity. Note that a contiguous file can be extended into a noncontiguous file by writing past the contiguous extents.

The granularity parameter can have the following values:

Ø	Same as volume granularity
ØFFFFH	Contiguous
other	Number of bytes/allocation

When a contiguous file is extended, space is allocated in volume-granularity units. If "other" is specified, a multiple of 1024 bytes is recommended.

This parameter is isgnored for physical and stream files

high$size
low$size — is applicable to stream and named files only and is a word pair giving the number of bytes initially reserved for the file.

must$create — is applicable to named files only and is a boolean whose TRUE or FALSE setting determines the handling of input paths designating an existing file.

INPUT PARAMETERS (continued)

resp$mbox	is a token for the mailbox that receives the result of this call.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is the token for a new file connection if the call succeeded; otherwise, an I/O result segment (Appendix C) is returned.

FILE ACCESS REQUIREMENT

The caller must have add-entry access to the parent directory of the new named file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST,E$FTYPE, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$SPACE, E$TYPE.

A$DELETE$CONNECTION

ASYNCHRONOUS DELETE$CONNECTION SYSTEM CALL

The A$DELETE$CONNECTION system call severs the file connection established by A$CREATE$FILE, A$CREATE$DIRECTORY, or A$ATTACH$FILE. It frees the file connection object, and also deletes the associated file if the file is already marked for deletion (by a previous A$DELETE$FILE call), and if the specified connection is the last remaining connection to the file. If a connection is open when A$DELETE$CONNECTION is called, it is closed before being severed.

NOTE

Connections should be deleted when no longer needed to avoid exceeding a job's object limit.

```
CALL  RQ$A$DELETE$CONNECTION(connection, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the file connection to be severed.
resp$mbox	is a token for the mailbox that receives an I/O result segment indicating completion of the operation. A zero indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating the connection has been deleted.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

ASYNCHRONOUS DELETE$FILE SYSTEM CALL

The A$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file is not actually deleted, however, until all connections to the file have been severed (by A$DELETE$CONNECTION calls). Directory files cannot be deleted unless they are empty.

Some of the parameters specified above do not apply to stream files.

NOTES

The "prefix/subpath" parameters must specify an existing connection.

Any task invoking this call using a path that is not null must have a priority of 32 or greater. If the path is null, the task can have any priority.

```
CALL  RQ$A$DELETE$FILE(user, prefix, subpath, resp$mbox);
```

INPUT PARAMETERS

user	applies to named files only and is a token for the user object to be inspected in access checking. A zero specifies the default user for the job.
prefix	is a token for a connection. In the case of a named file, this prefix acts as the starting point in a directory tree scan. A zero specifies the default prefix for the job.
subpath	applies to named files only and is a pointer to a string giving the subpath for the file being deleted. A null string indicates that the prefix itself designates the desired file. In this instance, the user parameter is not used since the caller's access (or lack of access) is built into the connection addressed by the prefix parameter.
resp$mbox	is a token for a mailbox that receives an I/O result segment indicating the file is marked for deletion. A zero specification for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

A$DELETE$FILE (continued)

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating completion of this operation. The result is returned when the file is marked for deletion, rather than when the file is actually deleted.

FILE ACCESS REQUIREMENT

The caller must have delete access to the file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FNEXIST, E$FTYPE, E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

ASYNCHRONOUS GET$CONNECTION$STATUS SYSTEM CALL

The AGETCONNECTION$STATUS system call returns information about a designated file connection.

```
CALL  RQ$A$GET$CONNECTION$STATUS(connection, resp$mbox
                                 excep$ptr);
```

INPUT PARAMETERS

connection — is a token for the file connection whose status is desired.

resp$mbox — is a token for the mailbox that receives the connection-status information.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RETURN STRUCTURE

status — is the connection-status information returned to the specified mailbox. This information is structured as follows:

```
DECLARE
  conn$status     STRUCTURE(
    status             WORD,
    file$driver        BYTE,
    flags              BYTE,
    open$mode          BYTE,
    open$share         BYTE,
    file$pointer       DWORD,
    access             BYTE
);
```

These fields are interpreted as follows:

status — is a condition code indicating how the status-fetch operation completed. If this code is not E$OK, the remaining fields must be considered invalid.

file$driver — tells the type of file driver to which this connection is attached. Possible bit values are:

1 Physical files
2 Stream files
4 Named files

AGETCONNECTION$STATUS (continued)

RETURN STRUCTURE (continued)

flags — contains two flag bits. If bit 1 is set to one, this connection is active and can be opened. If bit 2 is set, this connection is a device connection.

open$mode — is the mode established when this connection was opened. Possible values are:

0	Connection is closed
1	Open for reading
2	Open for writing
3	Open for reading and writing

open$share — is the current shared status established when this connection was opened. Possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

The open mode and shared state are initially set by the A$OPEN call.

file$pointer — is the current setting of the file pointer for this connection, by byte location.

access — gives the access rights for this connection. For each bit set to one, the corresponding access is granted. Bit values are:

Bit	Data File	Directory
0	Delete	Delete
1	Read	Display
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

ASYNCHRONOUS GET$DIRECTORY$ENTRY SYSTEM CALL

The AGETDIRECTORY$ENTRY system call applies to named files only. When called, it returns the file name associated with a specified directory entry. This name is a single subpath component for a file whose parent is the designated directory.

```
CALL RQ$A$GET$DIRECTORY$ENTRY(connection, entry$num,
                              resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection — is a token for the directory file containing the desired entry.

entry$num — is a word giving the entry number of the desired file name. Entries within a directory are numbered sequentially starting from zero. An E$EMPTY$ENTRY condition code may be issued if a file has been deleted and the I/O system has not reissued the entry to another file.

resp$mbox — is a token for the mailbox that receives the file name.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RETURN STRUCTURE

dir$entry — is the directory-entry information returned to the mailbox, structured as follows:

```
DECLARE
  dir$entry$info     STRUCTURE(
     status            WORD,
     name (14)         BYTE
  );
```

status — indicates how the operation completed. EOK, EEMPTY$ENTRY, and E$DIR$END condition codes all indicate successful completion.

name — is the file name contained in the designated entry, null padded. This field is valid only if status = E$OK.

AGETDIRECTORY$ENTRY (continued)

FILE ACCESS REQUIREMENT

The caller must have display access to the designated directory.

CONDITION CODES

EOK, EBAD$CALL, E$DIREND, EEMPTY$ENTRY, E$EXIST, E$FACCESS, E$FLUSHING, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

ASYNCHRONOUS GET$FILE$STATUS SYSTEM CALL

The AGETFILE$STATUS system call returns status and attribute information about the designated file. Certain common information is returned regardless of the file driver type (physical, stream, or named). Additional information is returned for named files.

Note that this call returns some device-dependent information, and its use may cause an application to become device dependent.

```
CALL RQ$A$GET$FILE$STATUS(connection, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection — is a token for a connection to the file whose status is sought.

resp$mbox — is a token for the mailbox that receives the common (and named file) status information.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGEMENT

common$info — is the common file-status information returned to the designated mailbox. This information is structured as follows:

```
DECLARE
 common$info  STRUCTURE(
    status          WORD,
    num$conn        WORD,
    num$reader      WORD,
    num$writer      WORD,
    open$share      BTYE,
    named$file      BYTE,
    dev$name (14)   BYTE,
    file$drivers    WORD,
    functs          WORD,
    dev$gran        WORD,
    dev$size        DWORD,
    dev$conn        WORD
);
```

These fields are interpreted as follows:

status — is a condition code indicating how the status-fetch operation completed. If this code is not E$OK, the remaining fields must be considered invalid.

AGETFILE$STATUS (continued)

RESULT SEGMENTS (continued)

num$conn — is the number of connections to the file.

num$reader — is the number of connections currently open for reading.

num$writer — is the number of connections currently open for writing.

open$share — is the current shared status of the file. Possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

named$file — specifies whether the file is a named file and, therefore, whether the mailbox will contain named-file, as well as common information. A value of ØFFH indicates the additional information is present.

dev$name — is the name of the device where this file resides, null padded.

file$drivers — indicates which file drivers can be used with the file. If bit n is on, then file driver n+1 can be used. Bits are numbered right to left starting with zero.

Bit	Driver No.	Driver
0	1	Physical file
1	2	Stream file
2	3	reserved
3	4	Named file

functs — describes the functions supported by the device where this file resides. Each bit set to one indicates the corresponding function is supported.

RESULT SEGMENTS
functs (continued)

Bit	Function
Ø	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE
8-15	Reserved

dev$gran — is the device granularity, in bytes.

dev$size — is the size of the device, in bytes.

dev$conn — is the number of connections to the device.

named$file$info — is the additional information returned if the designated file is a named file. This information is structured as follows:

```
DECLARE
   named$file$info    STRUCTURE(
     fdesc$num             WORD,
     file$type             BYTE,
     file$gran             BYTE,
     owner                 WORD,
     create$time           DWORD,
     access$time           DWORD,
     mod$time              DWORD,
     file$size             DWORD,
     file$blocks           DWORD,
     vol$name (6)          BYTE,
     vol$gran              WORD,
     vol$size              DWORD,
     id$count              WORD,
     accessor (*)          STRUCTURE(
     access                BYTE,
     id                    WORD)
);
```

AGETFILE$STATUS (continued)

RESULT SEGMENTS (continued)

These fields are interpreted as follows:

fdesc$num	is the number of the file's file descriptor. The file descriptor is an I/O system data structure containing file attribute and status data.
file$type	indicates the type of the file. A file with type other than FT$DATA (data file, type=8) may only be accessed from within the I/O system.
file$gran	specifies the file granularity.
owner	is the user ID number of the file's owner.
create$time	is the time and date when the file was created.
access$time	is the time and date when the file was last accessed.
mod$time	is the time and date when the file was last modified.
file$size	is the total size, in bytes, of the data in the file.
file$blocks	is the number of volume blocks allocated to this file.
vol$name	is the ASCII name for the volume containing this file.
vol$gran	is the volume granularity, in bytes.
vol$size	is the size of the volume, in bytes.

RESULT SEGMENTS (continued)

id$count	is the number of access/ user-ID pairs declared for this file. These pairs are detailed in the structure that follows.
accessor	is a list of up to three access/user-ID pairs.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

AGETPATH$COMPONENT

ASYNCHRONOUS GET$PATH$COMPONENT SYSTEM CALL

AGETPATH$COMPONENT can be called no matter what type of file is supported, but if a connection to a physical or stream file is specified, the call simply returns a null string.

A caller who knows the token for a connection to a file can specify the token to this system call and receive the name of the file in return. This is the name by which the file is cataloged in its parent directory. If the connection is to the root directory of a volume (that is, if no parent directory exists), a null string is returned. A null string is also returned if the file is marked for deletion.

```
CALL  RQ$A$GET$PATH$COMPONENT(connection, resp$mbox,
                              excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the file connection whose name is sought.
resp$mbox	is a token for the mailbox that receives the result segment containing the file name associated with the designated connection.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

file$name is a result segment giving the name of the file. This segment should be structured as follows:

```
DECLARE FILE$NAME
  file$name    STRUCTURE(
    STATUS           WORD,
    name             BYTE
  );
```

These fields are interpreted as follows:

status	is a condition code indicating how the operation completed.
name	is an ASCII string giving the desired file name. This name is the same as the last item in the subpath string specified when the file was created or renamed.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

ASYNCHRONOUS OPEN SYSTEM CALL

The A$OPEN system call opens a connection for I/O operations. The connection must be opened before such operations as reading and writing can be performed.

A$OPEN checks the current shared status of the connected file, and returns an error message if the requested mode is inconsistent with the sharing permitted. Open requests are not queued.

A$OPEN also initializes the file pointer to byte position zero. Subsequent I/O system calls (A$SEEK, A$READ, and A$WRITE) will move this pointer.

If the file is attached by multiple connections, the file might be open for reading by some connections and open for writing by others at the same time. Any modification of the file by a writer will be seen by the reader, if a reader subsequently reads the modified part of the file.

Note also that access to a file may be denied due to mode/shared-state incompatibility. No deadlock occurs, however, since open calls are not queued. The system does not notify callers when the shared status of the connection changes. If such notification is important, users of the file should arrange a proper protocol.

```
CALL  RQ$A$OPEN(connection, mode, share, resp$mbox, excep$ptr);
```


INPUT PARAMETERS

connection — is a token for the connection to be opened.

mode — is a byte giving the mode desired for the open connection. Possible values are:

1	Open for reading
2	Open for writing
3	Open for both reading and writing

share — is a byte specifying the kind of sharing desired for this connection. Possible values are:

Ø	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

A$OPEN (continued)

INPUT PARAMETERS (continued)

resp$mbox	is a token for the mailbox that receives the I/O result segment indicating completion of this operation. A zero value for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating that the open operation has been completed.

FILE ACCESS REQUIREMENT

The mode parameter must be compatible with the current shared state of the connected file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$PARAM, E$SHARE, E$TYPE.

ASYNCHRONOUS READ SYSTEM CALL

The A$READ system call initiates a read operation from an open connection. The connection is read as a string of bytes, starting at the current file pointer, and any number of bytes can be requested. Some efficiency may be gained by starting reads on device block boundaries. After the read operation is finished, the file pointer points just past the last byte read.

The buffer specified by the "buff$ptr" parameter must be in a segment allocated by the free-space manager of the RMX/86 nucleus executive (that is, the segment must be allocated dynamically).

```
CALL  RQ$A$READ(connection, buff$ptr, count, resp$mbox,
                excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open file connection to be read.
buff$ptr	is a pointer to the buffer that receives the data read.
count	is a word giving the number of bytes to be read.
resp$mbox	is a token for the mailbox that receives the I/O result segment after the read is complete. A zero value for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating completion of the read operation. It also contains the actual number of bytes read. If a read operation is requested with the file pointer set at or beyond the end of the file, an actual-bytes-read count of zero is returned in the I/O result segment. If all the connections to a stream file are requesting read operations, an actual-bytes-read count of zero is returned in the I/O result segment.

A$READ (continued)

FILE ACCESS REQUIREMENT

The mode of the open connection must permit reading (see A$OPEN).

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

ASYNCHRONOUS RENAME$FILE SYSTEM CALL

The A$RENAME$FILE system call applies to named files only. It is called to change the name of a file.

A renamed data file can be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent. Renamed directory files must keep the same parent, however.

```
CALL  RQ$A$RENAME$FILE(connection, user, prefix, subpath,
                       resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for a connection to the file being renamed. This connection and all other connections to the file will remain in effect after the file is renamed.
user	is a token for the user object to be inspected in access checking. A zero specifies the default user for the job.
prefix	is a token for the connection to be used as the starting point in a path scan. A zero specifies the default prefix for the job.
subpath	is a pointer to the string giving the new subpath for the file. Prefix and subpath must lead to a nonexistent file. The string pointed to by the subpath parameter cannot be a null string. For a data file, prefix and subpath may specify a different directory from the original parent, but it must be on the same volume. A directory file can only be renamed within its parent directory, however.
resp$mbox	is a token for the mailbox that receives an I/O result segment indicating completion of the rename. A zero value for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating that the rename operation is finished.

A$RENAME$FILE (continued)

FILE ACCESS REQUIREMENTS

The caller must have delete access to the original file and must have add-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$SUPPORT, E$TYPE.

ASYNCHRONOUS SEEK SYSTEM CALL

The A$SEEK system call applies to physical and named files only. This call moves the file pointer for an open connection, thus allowing file contents to be accessed randomly. The file pointer can be moved to any byte position in the file.

```
CALL  RQ$A$SEEK(connection, mode, hi$ptr$move, low$ptr$move,
                resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection — is a token for the open file connection whose file pointer is to be moved.

mode — is a byte describing the movement of the file pointer. Possible values are:

- 1 — Move pointer back by "ptr$move" amount. If this action moves the pointer past the beginning of the file, the pointer is set to zero.
- 2 — Set the pointer to the location specified by "ptr$move."
- 3 — Move the file pointer forward by "ptr$move" amount.
- 4 — Move the pointer to the end of the file, minus the "ptr$move" specified.

hiptrmove, lowptrmove — is a word pair giving the number of bytes involved in the seek. The interpretation of "ptr$move" depends on the mode setting, as explained above.

resp$mbox — is a token for the mailbox that receives an I/O result segment when the seek is completed. A zero value for this parameter indicates that no response is wanted.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result — is an I/O result segment (Appendix C) indicating that the seek operation has been completed.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, E$IFDR, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED, E$PARAM, E$TYPE.

A$SPECIAL

ASYNCHRONOUS SPECIAL SYSTEM CALL

The A$SPECIAL system call applies to physical files only. It lets the caller perform special device-level functions.

The special function to be done can be specified using either of two methods. If the function requires little supporting information (such as a function to rewind a magnetic tape), its code can be specified directly as the "spec$func" parameter in the system call.

Where additional information is needed, the user must create an I/O parameter block as described below. The special function is specified indirectly, as a field in this parameter block. The block is then addressed by the "ioparm$ptr" in the system call.

If only the "spec$func" parameter is used to specify the special function, the "ioparm$ptr" parameter is specified as zero.

```
CALL  RQ$A$SPECIAL(connection, spec$func, ioparm$ptr,
                   resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for a connection to the file where the special function is to be performed.
spec$func	is a word (code) that allows the user to pass a special function to a file driver without being required to set up a parameter block.
ioparm$ptr	is a pointer to a parameter block. The contents of the parameter block depends upon the requirements of the file driver being used to implement the special function. If the file driver requires no parameters for the function being requested, then ioparm$ptr can be zero. If the function does require parameters, you must build the parameter block to satisfy the requirements of the specific file driver.
resp$mbox	is a token for the mailbox that receives an I/O result segment indicating that the special function has been completed. A zero value in this parameter indicates that no result is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating that the special function has been completed.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, E$IDDR, E$IFDR, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

A$TRUNCATE

ASYNCHRONOUS TRUNCATE SYSTEM CALL

The A$TRUNCATE system call applies to named files only. This call truncates a file at the current setting of the file pointer, freeing all allocated space beyond the pointer. A$SEEK can be called to position the pointer before A$TRUNCATE is called. If the file pointer is at or beyond the end-of-file, no operation is performed.

Truncation is performed immediately, rather than waiting until connections to the file are deleted.

```
CALL  RQ$A$TRUNCATE(connection, resp$mbox, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for an open connection to the file being truncated.
resp$mbox	is a token for the mailbox that receives an I/O result segment indicating the truncation has been completed. A zero value for this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating completion of this operation.

FILE ACCESS REQUIREMENT

The designated file connection must be open for writing.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$IFDR, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

ASYNCHRONOUS WRITE SYSTEM CALL

The A$WRITE call writes data from the caller's buffer to a connected file. The data is written starting at the current file pointer. After the write, the file pointer is positioned just after the last byte written. Some efficiency may be gained by starting writes on device block boundaries.

NOTE

The buffer supplying the data to be written should not be modified until the write request has been acknowledged at the response mailbox.

This buffer must reside in a segment allocated by the RMX/86 nucleus' free-space manager (that is, the segment must be allocated dynamically).

```
CALL  RQ$A$WRITE(connection, buff$ptr, count, resp$mbox,
                 excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open connection to be written.
buff$ptr	is a pointer to the buffer that supplies the data to be written.
count	is a word giving the number of bytes to be written.
resp$mbox	is a token for the mailbox that receives the I/O result segment indicating the write is complete. A zero value in this parameter indicates that no response is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

result	is an I/O result segment (Appendix C) indicating that the write operation has been completed. It also contains the actual number of bytes written. If all the connections to a stream file are requesting write operations, an actual-bytes written count of zero is returned.

A$WRITE (continued)

FILE ACCESS REQUIREMENTS

The designated file connection must be open for writing, and the caller must have append or update access to the connection.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$SPACE, E$SUPPORT, E$TYPE.

CREATE I/O JOB SYSTEM CALL

The CREATEIOJOB system call creates a job and one task within the job. In doing so, it invokes the nucleus' CREATE$JOB call, requiring the caller to define the parameters needed by this call. The CREATEIOJOB call also includes several parameters useful for I/O processing and creates a mailbox for communication between the new job and its parent.

```
job = RQ$CREATE$IO$JOB(job$data$ptr, prefix$ptr,
                       log$nam$ptr, task$flags,
                       task$prior, task$start$addr,
                       data$token, stack$ptr,
                       stack$size, excep$ptr);
```

job$data$ptr — is a pointer to a job parameter structure as defined below.

prefix$ptr — is a pointer to a string that gives the default prefix for the job (specified as a logical name). A null string or zero pointer specifies that the calling job's default prefix is to be used.

lognamptr — is a pointer to a contiguous set of string/ token pairs, each giving a logical name to be inherited by the new job from its creator. A zero token associates its corresponding string with the parent job's token for the same string. The list is ended by a null string. A zero indicates that no logical names are to be inherited by the new job.

task$flags — is a word containing information about the initial task in the new job.

task$prior — is a byte specifying the priority of the new task. This priority must be lower (higher numerically) or equal to the parent job's maximum priority. A zero specifies that the task has the highest priority allowed by its job.

task$start$addr — is a pointer to the entry point of the task to be created within the new job.

data$token — is a token to which the initial task's DS and ES registers will be initialized. If zero, DS and ES are initialized to the value of the new job's CS register.

CREATEIOJOB (continued)

INPUT PARAMETERS (continued)

stack$ptr	is a pointer to the stack of the task being created within the new job. If the base of this address is zero, the system allocates a stack segment equal in size to the "stack$size" parameter described below.
stack$size	is a word containing the size of the stack for the new job's initial task.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

The job$param$ptr parameter points to a predefined job parameter structure declared as follows:

```
DECLARE
  job$data              STRUCTURE(
    dir$size                  WORD,
    param$obj                 TOKEN,
    pool$min                  WORD,
    pool$max                  WORD,
    max$objects               WORD,
    max$tasks                 WORD,
    max$prior                 BYTE,
    excep$hdler$offset        WORD,
    excep$hdler$base          WORD,
    excep$mode                BYTE,
    job$flags                 WORD,
    global$job                TOKEN,
    user                      TOKEN,
    msg$mbox                  TOKEN
  );
```

where:

dir$size	contains the maximum number of entries allowed in the new job's object directory.
param$obj	designates the new job's parameter object. A zero specifies that the job does not have a parameter object.
pool$min	contains the minimum size, in 16-byte pages, of the created job's memory pool. This figure is also the pool's initial size.

INPUT PARAMETERS
where: (continued)

pool$max	contains the maximum size, in 16-byte pages, of the created job's memory pool. This figure must be greater than or equal pool$min.
max$objects	contains the maximum number of objects that the new job can contain simultaneously. ØFFFFH specifies an unlimited number.
max$tasks	contains the maximum number of tasks that can exist simultaneously within the new job. ØFFFFH specifies an unlimited number.
max$prior	is the maximum priority of any task in the new job from 1 (high) to 255 (low). A zero specifies that the highest priority is the same as that of the parent job's tasks.
excep$hdler$offset excep$hdler$base	form a pointer to the exception handler for the new job. A zero specifies the system default exception handler to be used.
excep$mode	specifies if and when control should be passed to the job's exception handler. Possible values are: Ø — No exceptions recognized. 1 — Recognize programming errors only. 2 — Recognize environmental conditions only. 3 — Recognize all exceptions.
job$flags	contains job information needed by the RMX/86 nucleus. If any tasks in the job use the 8087 component, the low-order bit should be set to one.
global$job	specifies the job whose object directory is to be used as the global logical-name directory for the job being created. A zero specifies that the caller's global job is to be used.

CREATEIOJOB (continued)

where (continued)

user	specifies the default user object for the new job. A zero specifies that the calling job's default is to be used.
msg$mbox	is a token for the mailbox through which any exit message will be sent to the parent job. A zero specifies that no message is wanted. (See EXITIOJOB).

RETURN VALUE

job	is a token for the new job created by this call.

CONDITION CODES

EOK, ELIMIT, ELOGNAME$NEXIST, E$MEM, E$NHUSER, E$NOPREFIX, E$NOUSER, E$PREFIX$STRING$NEXIST.

DELETE$PACKAGE SYSTEM CALL

DELETE$PACKAGE is called to delete a package object previously created by the loader, or other source.

```
CALL  RQ$DELETE$PACKAGE(package, excep$ptr);
```

INPUT PARAMETERS

package	is a token for the package object to be deleted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

EXITIOJOB

EXIT I/O JOB SYSTEM CALL

The EXITIOJOB system call causes the calling task to be deleted. EXITIOJOB performs the following operations:

- deletes all connections and detaches all logical devices attached by the job;
- creates a segment for the exit message;
- sends the exit message to the mailbox provided by the parent job when the exiting job was created.;
- deletes the calling task.

The parent of the exiting job can then delete it, if it wishes to do so.

NOTES

A job whose last task has been deleted will not exit automatically.

No other tasks in a job can be in the ready state when a task in the job calls EXITIOJOB.

Tasks that call EXITIOJOB must use hybrid or synchronous file system calls to create connections. Otherwise, a system failure may occur when a child job is deleted after being exited.

```
CALL RQ$EXIT$IO$JOB(user$excep$code, return$data,
                    return$data$len, excep$ptr);
```

INPUT PARAMETERS

user$excep$code — is a word giving a user exception code. If this code is zero, the exit is considered normal and the exit message contains zeros in its return$code and exception$code fields. If user$excep$code is nonzero, an abnormal exit is assumed and the exit message contains a two in its return$code field. The exception$code field has the same value as user$excep$code.

return$data — is a pointer to the data to be returned to the parent job. If the pointer is zero, no data is returned. Otherwise, the number of bytes specified by the following parameter is copied into the return$data field of the exit message segment.

INPUT PARAMETERS (continued)

return$data$len — is a word giving the number of bytes to be copied into the exit message segment.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

exit$msg — is the exit message returned to the mailbox established by the parent job when the exiting job was created. It has the following structure:

```
DECLARE
 exit$message    STRUCTURE(
   return$code       WORD,
   exception$code    WORD,
   job$token         WORD,
   return$data$len   WORD,
   return$data(*)    BYTE
);
```

where:

return$code — explains the returning condition. Possible values for this field are:

- 0 — Normal exit; job terminated successfully. EXITIOJOB was called with user$excep$code parameter set to zero.
- 1 — Job exited due to a system-related error. The I/O system caused the job to exit.
- 2 — Job exited due to a user-related error. EXITIOJOB was called with a nonzero user$excep$code paramater.

exception$code — gives the exception code, which is influenced by the return$code field's contents. If return$code is zero, this field is zero. If return$code is one (system-detected error), this field contains an I/O system condition code. If return$code is two (user-related error), this field has the same code as the user$excep$code input parameter.

EXITIOJOB (continued)

RESULT SEGMENT
where: (continued)

job$token	is a word giving the exited job's token relative to the parent job.
return$data$len	gives the length (in bytes) of the return$data provided with the EXITIOJOB call.
return$data	is a sequence of bytes containing the return$data provided with the EXITIOJOB call.

CONDITION CODES

EOK, ENOT$CONFIGURED

GET$DEFAULT$PREFIX SYSTEM CALL

The GET$DEFAULT$PREFIX system call allows the caller to determine the default prefix for the specified job.

```
connection = RQ$GET$DEFAULT$PREFIX(job, excep$ptr);
```

INPUT PARAMETERS

job	is a token for the job whose default prefix is sought. A zero specifies the calling job.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection object which is the default prefix for the designated job.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$NOPREFIX, E$NOT$CONFIGURED, E$TYPE.

GET$DEFAULT$USER

GET$DEFAULT$USER SYSTEM CALL

The GET$DEFAULT$USER system call allows the calling task to determine the default user object associated with the designated job.

```
user$id = RQ$GET$DEFAULT$USER(job, excep$ptr);
```

INPUT PARAMETERS

job	is a token for the job whose default user object is sought. A zero specifies the calling job.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

user$id	is a token for the user object which is the default user for the designated job.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, ENOTCONFIGURED, E$NOUSER, E$TYPE.

GET$TIME SYSTEM CALL

The GET$TIME system call returns the date/time value from its doubleword counter.

```
date$time = RQ$GET$TIME(excep$ptr);
```

INPUT PARAMETER

excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGEMENT

date$time	is a doubleword date/time value expressed as the number of seconds since midnight, January 1, 1978.

CONDITION CODES

EOK, EBAD$CALL, E$NOT$CONFIGURED.

GET$TIME$STRING

GET$TIME$STRING SYSTEM CALL

The GET$TIME$STRING system call returns the current date and time.

```
CALL  RQ$GET$TIME$STRING(dt$ptr, excep$ptr);
```

INPUT PARAMETERS

dt$ptr	is a pointer to the location where the date and time values are returned.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

date	is a series of ASCII characters giving today's date in the form: MM/DD/YY where: "MM" is the month (Ø1 - 12) "DD" is the day (Ø1 - 31) "YY" is the year (ØØ - 99)
time	is a series of ASCII characters giving the current time in the form: HH:MM:SS where: "HH" is the hour (ØØ - 24) "MM" is the minute (ØØ - 59) "SS" is the second (ØØ - 59)

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, ENOTCONFIGURED, E$PARAM, E$TYPE.

HYBRID ATTACH$FILE SYSTEM CALL

The H$ATTACH$FILE system call creates a connection to an existing file. The new connection can also be given a logical name and cataloged in the job's logical-name directory. The caller is assumed to be the default user for the job.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
connection = RQ$H$ATTACH$FILE(log$name, path, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical name under which the new connection is to be cataloged in the job's logical-name directory. A zero in this parameter or a null string indicates the connection will not have a logical name.
path	is a pointer to the string containing the prefix and subpath to the file being attached.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the new connection to the file designated by the path parameter.

CONDITON CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FNEXIST, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

H$CHANGE$ACCESS

HYBRID CHANGE$ACCESS SYSTEM CALL

The H$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file. The caller is assumed to be the default user.

NOTE

Currently, all users have access to all files at the hybrid level. The new access rights specified in this call apply to all users, including the caller.

```
CALL  RQ$H$CHANGE$ACCESS(path, mode, name, access, excep$ptr);
```

INPUT PARAMETERS

path — is a pointer to the string giving the prefix and subpath to the file whose access rights are to be changed.

mode — is a byte value, currently ignored.

name — is a pointer to the string giving the name of the accessor whose rights are being added, deleted, or changed. Currently, the string must be "WORLD," which includes all users.

access — is a byte mask giving the new access rights for the specified accessor. If a bit is set to one, the corresponding access is granted. For a named data file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

For a named directory file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

INPUT PARAMETERS (continued)

excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENTS

The caller must be the owner of the file or must have change-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

H$CREATE$DIRECTORY

HYBRID CREATE$DIRECTORY SYSTEM CALL

The H$CREATE$DIRECTORY system call is applicable to named directory files only. When called, it creates a new directory file and returns a token for the connection to the new file. The new connection can also be cataloged under a specified logical name in the calling job's logical-name directory.

The caller is assumed to be the default user for the job. Full owner access to the directory is assumed; that is, bits 0-3 in the access byte mask are all set to one.

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

NOTE

Any task invoking this call
must have a priority of 32 or greater.

```
connection = RQ$H$CREATE$DIRECTORY(log$name, path, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical name under which the new connection is to be cataloged in the calling job's logical-name directory. A zero in this parameter or a null string specifies that the connection is not to be given a logical name.
path	is a pointer to the string containing the prefix and subpath to the directory being created. This string cannot be null, and the directory name cannot exist already.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection to the new directory file.

FILE ACCESS REQUIREMENT

The caller must have add-entry access to the parent of the new directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$SPACE, E$TYPE.

H$CREATE$FILE

HYBRID CREATE$FILE SYSTEM CALL

The H$CREATE$FILE system call creates a new physical, stream, or named data file and returns a token for the new file connection. The connection can also be given a logical name and cataloged in the job's logical-name directory. The owner of the file is assumed to be the job's default user. Full owner access is assumed for the file, meaning bits 0-3 of the access byte mask are set to one.

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

The H$CREATE$FILE call also assumes the file granularity is the same as the volume granularity and the file size (preallocation) is zero bytes.

If a named file designated by the path parameter already exists, one of the following situations occurs:

- If the "must$create" parameter is TRUE, an error condition code (E$FEXIST) is returned.
- If the "must$create" parameter is FALSE and the path designates a data file, a new connection to that file is returned (that is, H$CREATE$FILE acts like H$ATTACH$FILE).
- If the "must$create" parameter is FALSE and the path designates a device or directory, an unnamed file is created on the corresponding device. This file is deleted automatically when the last connection to it is deleted.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
connection = RQ$H$CREATE$FILE(log$name, path, must$create,
                              excep$ptr);
```

INPUT PARAMETERS

log$name — is a pointer to the string giving the logical name under which the new connection will be cataloged in the job's logical-name directory. A zero in this parameter or a null string specifies the connection will not be given a logical name.

INPUT PARAMETERS (continued)

path	is a pointer to the string giving the prefix and subpath for the file being created.
must$create	is a boolean whose TRUE or FALSE setting determines the handling of input paths designating an existing named file.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection to the newly created file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST, E$FTYPE, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$SPACE, E$TYPE.

H$DELETE$CONNECTION

HYBRID DELETE$CONNECTION SYSTEM CALL

The H$DELETE$CONNECTION system call severs the file connection established by H$CREATE$FILE, H$CREATE$DIRECTORY, or H$ATTACH$FILE. It frees the connection object and also deletes the associated file if the file is already marked for deletion and if the deleted connection was the last remaining connection to the file. If a connection is open when H$DELETE$CONNECTION is called, it is closed before being severed.

```
CALL  RQ$H$DELETE$CONNECTION(connection, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the connection to be deleted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED.

HYBRID DELETE$FILE SYSTEM CALL

The H$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file will not actually be deleted, however, until all connections to the file have been severed (by calls to H$DELETE$CONNECTION). Directory files cannot be deleted unless they are empty.

The caller is assumed to be the default user for the job.

```
CALL  RQ$H$DELETE$FILE(path, excep$ptr);
```

INPUT PARAMETERS

^path	is a pointer to the string giving the prefix and subpath to the file being deleted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENT

The caller must have delete access to the file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FNEXIST, E$FTYPE,E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

HGETFILE$STATUS

HYBRID GET$FILE$STATUS SYSTEM CALL

The HGETFILE$STATUS system call returns status and attribute information about the designated file. Certain common information is returned regardless of the file driver type (physical, stream, or named). Additional information is returned for named files.

Note that this call returns some device-dependent information, and its use may cause an application to become device dependent.

```
CALL  RQ$H$GET$FILE$STATUS(path, file$info$ptr, excep$ptr);
```

INPUT PARAMETERS

path	is a pointer to the string giving the prefix and subpath to the file whose status is sought.
file$info$ptr	is a pointer to the location that receives the common (and named file) status information.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN STRUCTURES

common$info — is the common file-status information returned. This information is structured as follows:

```
DECLARE
  common$info     STRUCTURE(
    dev$share           BYTE,
    num$conn            WORD,
    num$reader          WORD,
    num$writer          WORD,
    open$share          BYTE,
    named$file          BYTE,
    dev$name (14)       BYTE,
    file$drivers        WORD,
    functs              WORD,
    dev$gran            WORD,
    dev$size            DWORD,
    dev$conn            WORD
  );
```

RETURN STRUCTURES
common$info (continued)

These fields are interpreted as follows:

dev$share — indicates whether the device where this file resides is sharable or nonsharable. Possible values are:

Ø	Sharable device
1	Nonsharable device

num$conn — is the number of connections to the file.

num$reader — is the number of connections currently open for reading.

num$writer — is the number of connections currently open for writing.

open$share — is the current shared status of the file. Possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

named$file — specifies whether the file is a named file and, therefore, whether file$info$ptr wil contain named-file, as well as common information. A value of ØFFH indicates the additional information is present.

dev$name — is the name of the device where this file resides, null padded.

file$drivers — indicates which file drivers can be used with the file. If bit n is on, then file driver n+1 can be used. Bits are numbered right to left starting with zero.

HGETFILE$STATUS (continued)

RETURN STRUCTURES
common$info (continued)

Bit	Driver No.	Driver
0	1	Physical file
1	2	Stream file
2	3	reserved
3	4	Named file

functs — describes the functions supported by the device where this file resides. Each bit set to one indicates the corresponding function is supported.

Bit	Function
Ø	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE
8-15	Reserved

dev$gran — is the device granularity, in bytes.

dev$size — is the size of the device, in bytes.

dev$conn — is the number of connections to the device.

named$file$info — is the additional information returned if the designated file is a named file. This information is structured as follows:

```
DECLARE
 named$file$info     STRUCTURE(
   fdesc$num             WORD,
   file$type             BYTE,
   file$gran             BYTE,
   owner (14)            BYTE,
   create$time (16)      BYTE,
   access$time (16)      BYTE,
   mod$time (16)         BYTE,
   file$size             DWORD,
   file$blocks           DWORD,
   vol$name (16)         BYTE,
   vol$gran              WORD,
   vol$size              DWORD,
   id$count              WORD,
```

RETURN STRUCTURES
named$file$info (continued)

```
        accessor (*)          STRUCTURE(
          access                BYTE,
          id                    WORD)
    );
```

These fields are interpreted as follows:

fdesc$num	is the number of the file's file descriptor. The file descriptor is an I/O system data structure containing file attribute and status data.
file$type	indicates the type of the file. A file with type other than FT$DATA (data file, type=8) may only be accessed from within the I/O system.
file$gran	specifies the file granularity.
owner	is the ASCII name of the file's owner.
create$time	is the time and date when the file was created.
access$time	is the time and date when the file was last accessed.
mod$time	is the time and date when the file was last modified.
file$size	is the total size, in bytes, of the data in the file.
file$blocks	is the number of volume blocks allocated to this file.
vol$name	is the ASCII name for the volume containing this file.
vol$gran	is the volume granularity, in bytes.
vol$size	is the size of the volume, in bytes.

HGETFILE$STATUS (continued)

RETURN STRUCTURES
named$file$info (continued)

id$count	is the number of access/user-ID pairs declared for this file. These pairs are detailed in the structure that follows.
accessor	is a list of up to three access/user-ID pairs.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

HYBRID LOOKUPCONNECTION SYSTEM CALL

The H$LOOK$UP$CONNECTION system call returns information about a file connection to the caller. The caller can specify a conection's logical name to this system call and receive the token associated with that connection in return. The I/O system looks for the name first in the job's logical-name directory. If the name is not found, it then looks in the global directory for the job, and finally in the system's logical-name directory. If the name does not reside in either directory, an exception code is returned.

```
connection = RQ$H$LOOK$UP$CONNECTION(log$name, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical file name to be looked up.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection associated with the specified logical name.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$NOT$CONFIGURED, E$PARAM, E$TYPE.

H$RENAME$FILE

HYBRID RENAME$FILE SYSTEM CALL

The H$RENAME$FILE system call applies to named files only. It is called to change the name of a file (that is, the name by which it is cataloged in its parent directory). A renamed data file can also be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent. A directory file can only be renamed within its parent directory, however.

The caller is assumed to be the default user for the job.

```
CALL  RQ$H$RENAME$FILE(old$path, new$path, excep$ptr);
```

INPUT PARAMETERS

old$path	is a pointer to the string giving the prefix and subpath to the file being renamed.
new$path	is a pointer to the string giving a new prefix and subpath for the file. This parameter must specify a nonexistent file. For a data file, it may specify a different directory than the original parent, but the new parent must be on the same volume. For a directory file, however, the original parent must be retained.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENTS

The caller must have delete access to the original file and must have add-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$SUPPORT, E$TYPE.

INSPECT$PACKAGE SYSTEM CALL

INSPECT$PACKAGE is called to expand the contents of a package object previously created by the loader or other source.

```
CALL  RQ$INSPECT$PACKAGE(package, tok$list$ptr, excep$ptr);
```

INPUT PARAMETERS

package — is a token for the package to be inspected.

tok$list$ptr — is the structure that lists the tokens contained in the package object. This information is returned in the following form:

```
DECLARE
 token$list  STRUCTURE(
  num$slots       WORD,
  num$tokens      WORD,
  tokens (*)      WORD
);
```

These fields are interpreted as follows:

num$slots — contains the number of tokens slots in the list.

num$tokens — is the actual number of tokens contained in the package object. INSPECT$PACKAGE will never store more tokens than the space alotted in num$slots.

tokens — is a contiguous series of words, each containing one of the tokens making up the package object.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

S$ATTACH$FILE

SYNCHRONOUS ATTACH$FILE SYSTEM CALL

The S$ATTACH$FILE system call creates a composite connection to an existing file, including buffers needed for synchronous I/O operations on the connection. The composite connection formed can be given a logical name and cataloged in the calling job's logical-name directory.

The "buff$size" parameter specifies the default buffer size. A different, actual buffer size can be specified by S$OPEN when the connection is opened.

The caller is assumed to be the default user for the job.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
connection = RQ$S$ATTACH$FILE(log$name, path, buff$size,
                              num$buff, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical name under which the composite object (high-level connection) is to be cataloged in the job's logical-name directory. A zero in this parameter or a null string indicates that the object should not be cataloged.
path	is a pointer to the string giving the prefix and subpath of the file to be attached.
buff$size	is a word giving the default SIOS (or user) buffer size (in bytes). A zero indicates that the buffer size should be the same as the file granularity.
num$buff	is a byte giving the default number of SIOS buffers to be used. The maximum allowed is two.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the composite object (high-level connection) attached to the file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FNEXIST, E$FTYPE, E$IO, E$LIMIT, E$MEM, E$NOBUFF, E$NOPREFIX, ENOTCONFIGURED, E$NOUSER, E$NUMBUFF, E$PARAM, E$TYPE,.

S$CHANGE$ACCESS

SYNCHRONOUS CHANGE$ACCESS SYSTEM CALL

The S$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file. The caller is assumed to be the default user.

NOTE

Currently, all users have access to all files at the synchronous level. The new access rights specified in this call will apply to all users, including the caller.

```
CALL  RQ$S$CHANGE$ACCESS(path, mode, name, access, excep$ptr);
```

INPUT PARAMETERS

path — is a pointer to the string giving the prefix and subpath to the file whose access rights are to be changed.

mode — is a byte value, currently ignored.

name — is a pointer to the string giving the name of the accessor whose rights are being added, deleted, or changed. Currently, this string must be "WORLD," which includes all users.

access — is a BYTE mask giving the new access rights for the specified accessor. If a bit is set to one, the corresponding access is granted. For a named data file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

For a named directory file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

INPUT PARAMETERS (continued)

excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENTS

The caller must be the owner of the file or must have change-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

S$CLOSE

SYNCHRONOUS CLOSE SYSTEM CALL

The S$CLOSE system call closes an open composite connection, including the buffers associated with that connection. It performs the following steps:

- waits for I/O operations in progress to be completed.
- makes sure that all data in the buffer of an output file is completely written if the buffer is partially filled.
- releases the buffer(s) created by S$OPEN.
- closes the file connection.

```
CALL  RQ$S$CLOSE(connection, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open composite connection to be closed.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$CANNOT$CLOSE, E$CONTEXT, E$EXIST, E$FLUSHING, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

SYNCHRONOUS CREATE$DIRECTORY SYSTEM CALL

The S$CREATE$DIRECTORY system call applies to named directory files only. When called, it creates a new directory file. The new connection can also be cataloged under a specified logical name in the job's logical-name directory.

The caller is assumed to be the default user for the job. Full owner access to the directory is assumed; that is, bits 0-3 in the access byte mask are all set to one.

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
connection = RQ$S$CREATE$DIRECTORY(log$name, path, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical name under which the new connection is to be cataloged in the job's logical-name directory. A zero in this parameter or a null string specifies that the connection is not to be given a logical name.
path	is a pointer to the string containing the prefix and subpath to the directory being created. This string cannot be null, and the directory name cannot exist already.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection to the new directory file.

FILE ACCESS REQUIREMENT

The caller must have add-entry access to the parent of the new directory.

S$CREATE$DIRECTORY (continued)

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST, E$FTYPE, E$IFDR, EIO, ELIMIT, EMEM, ENOPREFIX, ENOTCONFIGURED, E$NOUSER, E$PARAM, E$SPACE, E$TYPE.

SYNCHRONOUS CREATE$FILE SYSTEM CALL

The S$CREATE$FILE system call creates a new physical, stream or named data file. It also creates a composite connection to the file including a token from a segment giving the number and size of SIOS buffers to be used in I/O operations on the file. The token for this connection is returned by the S$CREATE$FILE call. The connection can also be given a logical name and cataloged in the job's logical-name directory.

The owner of the file is assumed to be the job's default user. Full owner access is assumed, meaning bits 0-3 of the access byte mask are set to one.

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved

The S$CREATE$FILE calls also assumes the file granularity is the same as the volume granularity and the file size (pre-allocation) is zero bytes.

If a named file designated by the path parameter already exists, one of the following situations occurs:

- If the "must$create" parameter is TRUE, an error condition code (E$FEXIST) is returned.
- If the "must$create" parameter is FALSE and the path designates a data file, a new connection to that file is returned (that is, S$CREATE$FILE acts like S$ATTACH$FILE) and the file is truncated to zero length.
- If the "must$create" parameter is FALSE and the path designates a device or directory, an unnamed file is created on the corresponding device. This file is deleted automatically when the last connection to it is deleted.

NOTE

Any task invoking this call must have a priority of 32 or greater.

```
connection = RQ$S$CREATE$FILE(log$name, path, buff$size,
                              num$buff, must$create,
                              excep$ptr);
```

S$CREATE$FILE (continued)

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical name under which the new connection is to be cataloged in the job's logical-name directory. A zero in this parameter or a null string specifies that the connection is not to be given a logical name.
path	is a pointer to the string giving the prefix and subpath for the file being created.
buff$size	is a word giving the default size (in bytes) of any SIOS buffers created. A zero indicates that the buffer size is the same as the file granularity for a named file, the device granularity for a physical file, and the folume granularity for a stream file.
num$buff	is a byte giving the default number of SIOS buffers to be used. The maximum allowed is two.
must$create	is a boolean whose TRUE or FALSE setting determines the handling of input paths designating an existing named file.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the composite object (high-level connection) for the newly created file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FNEXIST, E$FTYPE, E$IO, E$LIMIT, E$MEM, E$NOBUFF, E$NOPREFIX, ENOTCONFIGURED, E$NOUSER, E$NUMBUFF, E$PARAM, E$SPACE, E$TYPE.

SYNCHRONOUS DELETE$CONNECTION SYSTEM CALL

The S$DELETE$CONNECTION system call severs the composite file connection created by S$CREATE$FILE or S$ATTACH$FILE. It frees the composite connection object. It also deletes the file if the file is already marked for deletion, and if the specified connection is the last remaining connection to the file. If the connection is open when S$DELETE$CONNECTION is called, it is closed before being severed.

```
CALL  RQ$S$DELETE$CONNECTION(connection, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the composite connection object to be severed.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

S$DELETE$FILE

SYNCHRONOUS DELETE$FILE SYSTEM CALL

The S$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion. The file will not actually be deleted, however, until all connections to the file have been severed (by calls to S$DELETE$CONNECTION). Directory files cannot be deleted unless they are empty.

The caller is assumed to be the default user for the job.

```
CALL  RQ$S$DELETE$FILE(path, excep$ptr);
```

INPUT PARAMETERS

path	is a pointer to the string giving the prefix and subpath to the file being deleted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENT

The caller must have delete access to the file.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FNEXIST, E$FTYPE, E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$TYPE.

SET$DEFAULT$PREFIX SYSTEM CALL

The SET$DEFAULT$PREFIX system call sets the default prefix for an existing job. A job created by calling CREATEIOJOB can have its default prefix set at the time it is created.

```
CALL  RQ$SET$DEFAULT$PREFIX(job, prefix, excep$ptr);
```

INPUT PARAMETERS

job	is a token for the job whose default prefix is to be set. A zero specifies the current job.
prefix	is a token for the connection that is to become the default prefix.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

SET$DEFAULT$USER

SET$DEFAULT$USER SYSTEM CALL

The SET$DEFAULT$USER system call sets the default user for an existing job. A job created by calling CREATEIOJOB can have its default user set at the time it is created.

```
CALL  RQ$SET$DEFAULT$USER(job, user, excep$ptr);
```

INPUT PARAMETERS

job	is a token for the job whose default user object is to be set. A zero designates the current job.
user	is a token for the user object that is to become the default user.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$LIMIT, E$MEM, ENOTCONFIGURED, E$TYPE.

SYNCHRONOUS GET$CONNECTION$STATUS SYSTEM CALL

SGETCONNECTION$STATUS is called to obtain status information associated with the designated connection.

```
CALL  RQ$S$GET$CONNECTION$STATUS(connection, conn$info$ptr,
                                 excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the connection whose status is sought.
conn$info$ptr	is a pointer to the location that receives the status information.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RESULT SEGMENT

conn$info is the desired status information, structured as follows:

```
DECLARE
 conn$info     STRUCTURE
   flags           BYTE,
   open$mode       BYTE,
   open$share      BYTE,
   file$pointer    DWORD,
   access          BYTE,
   num$buff        BYTE,
   buff$size       WORD,
   seek            BOOLEAN
);
```

These fields are interpreted as follows:

flags	contains two flag bits. If bit 1 is set to one, this is an active connection and can be opened. If bit 2 is set, this is a device or directory file connection.
open mode	is the mode established when this connection was opened. Possible values are:

SGETCONNECTION$STATUS (continued)

RESULT SEGEMENT
conn$info (continued)

	0 Connection is closed 1 Open for reading only 2 Open for writing only 3 Open for reading and writing
open$share	is the shared status established when this connection was opened. Possible values are: 0 Private use only 1 Share with readers only 2 Share with writers only 3 Share with all users
file$pointer	gives the pointer's current byte location in the file.
access	gives the access rights for this connection. Possible settings of the byte mask are: Bit — Access 0 — Delete 1 — Read 2 — Append 3 — Update 4-7 — Reserved
num$buff	is the number of buffers specified for read and write operations on this connection.
buff$size	is the size (in bytes) of the buffers. The default buffer size is the size specified in the S$CREATE$FILE or S$ATTACH$FILE call that created this connection.
seek	if TRUE, indicates the SEEK function is supported; If FALSE, indicates the SEEK function is not supported.

SGETCONNECTION$STATUS (continued)

CONDITON CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

SGETFILE$STATUS

SYNCHRONOUS GET$FILE$STATUS SYSTEM CALL

The SGETFILE$STATUS system call returns status and attribute information about the designated file. Certain common information is returned regardless of the file driver type (physical, stream or named). Additional information is returned for named files.

Note that this call returns some device-dependent information, and its use may cause an application to become device dependent.

```
CALL  RQ$S$GET$FILE$STATUS(path, file$info$ptr, excep$ptr);
```

INPUT PARAMETERS

path	is a pointer to the string giving the prefix and subpath to the file whose status is sought.
file$info$ptr	is a pointer to the location that receives the common (and named file) status information.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN STRUCTURES

common$info — is the common file-status information returned. This information is structured as follows:

```
DECLARE
 common$info     STRUCTURE(
  dev$share         BYTE,
  num$conn          WORD,
  num$reader        WORD,
  num$writer        WORD,
  open$share        BYTE,
  named$file        BYTE,
  dev$name (14)     BYTE,
  file$drivers      WORD,
  functs            WORD,
  dev$gran          WORD,
  dev$size          DWORD,
  dev$conn          WORD
 );
```

These fields are interpreted as follows:

dev$share — indicates whether the device where this file resides is sharable or nonsharable. Possible values are:

RETURN STRUCTURES
common$info (continued)

0 Sharable device
1 Nonsharable device

num$conn — is the number of connections to the file.

num$reader — is the number of connections currently open for reading.

num$writer — is the number of connections currently open for writing.

open$share — is the current shared status of the file. Possible values are:

0 Private use only
1 Share with readers only
2 Share with writers only
3 Share with all users

named$file — specifies whether the file is a named file and, therefore, whether file$info$ptr will contain named-file, as well as common information. A value of ØFFH indicates the additional information is present.

dev$name — is the name of the device where this file resides, null padded.

file$drivers — indicates which file drivers can be used with the file. If bit is on, then file driver n+1 can be used. Bits are numbered right to left starting with zero.

Bit	Driver No.	Drivers
0	1	Physical file
1	2	Stream file
2	3	reserved
3	4	Named file

functs — describes the functions supported by the device where this file resides. Each bit set to one indicates the corresponding function is supported.

GETFILE$STATUS (continued)

RETURN STRUCTURES
common$info
functs (continued)

Bit	Function
Ø	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE
8-15	Reserved

dev$gran — is the device granularity, in bytes.

dev$size — is the size of the device, in bytes.

dev$conn — is the number of connections to the device.

named$file$info — is the additional information returned if the designated file is a named file. This information is structured as follows:

```
DECLARE
 named$file$info     STRUCTURE(
   fdesc$num            WORD,
   file$type            BYTE,
   file$gran            BYTE,
   owner (14)           BYTE,
   create$time (16)     BYTE,
   access$time (16)     BYTE,
   mod$time (16)        BYTE,
   file$size            DWORD,
   file$blocks          DWORD,
   vol$name (6)         BYTE,
   vol$gran             WORD,
   vol$size             DWORD,
   id$count             WORD,
   accessor (*)         STRUCTURE(
     access                BYTE,
     id                    WORD)
);
```

These fields are interpreted as follows:

RETURN STRUCTURES
named$file$info (continued)

fdesc$num	is the number of the file's file descriptor. The file descriptor is an I/O system data structure containing file attribute and status data.
file$type	indicates the type of the file. A file with type other than FT$DATA (data file, type=8) may only be accessed from within the I/O system.
file$gran	specifies the file granularity.
owner	is the ASCII nme of the file's owner.
create$time	is the time and date when the file was created.
access$time	is the time and date when the file was last accessed.
mod$time	is the time and date when the file was last modified.
file$size	is the total size, in bytes, of the data in the file.
file$blocks	is the number of volume blocks allocated to this file.
vol$name	is the ASCII name for the volume containing this file.
vol$size	is the size of the volume, in bytes.
id$count	is the number of access/user-ID pairs declared for this file. These pairs are detailed in the structure that follows.
accessor	is currently limited to "WORLD".

SGETFILE$STATUS (continued)

CONDITION CODES

EOK, EBAD$CALL, E$EXIST, E$FLUSHING, E$LIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

SYNCHRONOUS LOAD SYSTEM CALL

S$LOAD is called to load an 8086 object file into memory during job execution. Object files must be located by LOC86 before they can be loaded. The user must be certain that all absolute address assignments are in a space he has reserved for this purpose during system configuration.

S$LOAD can also create a new job for the object file being loaded, as well as a new task for executing the file. If no new task is created, S$LOAD returns register initialization values for the loaded task.

```
job = RQ$S$LOAD(job$data$ptr, prefix$ptr, log$names$ptr,
                task$flags, task$prior, load$function,
                path, return$data$ptr, excep$ptr);
```

INPUT PARAMETERS

job$data$ptr	is a pointer to the job data structure defined below. This structure is used only if a new job is to be created and is ignored otherwise.
prefix$ptr	is a pointer to a string giving the default prefix for the job (specified as a logical name). A null string or zero pointer specifies that the calling job's default prefix is to be used.
log$names$ptr	is a pointer to a contiguous set of string/token pairs, each giving a logical name to be inherited by the new job from its creator. A zero token assoicates its corresponding string with the parent job's token for the same string.
task$flags	is a word containing information about the initial task in the new job.
task$prior	is a byte specifying the priority of the new task. This priority must be lower (higher numerically) or equal to the parent job's maximum priority. A zero specifies that the task has the highest priority allowed by its job.
load$function	is a byte specifying the function to be done: 0 Create new job and initial task 1 Load into calling job and create task 2 Load into calling job and return register initialization values

S$LOAD (continued)

INPUT PARAMETERS (continued)

path	is a pointer to the string giving the prefix and subpath of the object file to be loaded.
return$data$ptr	is a pointer to the buffer that receives a result segment when S$LOAD is finished.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

The job$data$ptr parameter must point to a previously-defined structure of the following form:

```
DECLARE
 job$data          STRUCTURE(
   dir$size             WORD,
   param$obj            TOKEN,
   min$job$size         WORD,
   max$job$size         WORD,
   max$objects          WORD,
   max$tasks            WORD,
   max$prior            BYTE,
   excep$hdler$offset   WORD,
   excep$hdler$base     WORD,
   excep$mode           BYTE,
   job$flags            WORD,
   global$job           TOKEN,
   user                 TOKEN,
   msg$mbox             TOKEN
 );
```

where:

dir$size	is a word containing the maximum number of entries in the created job's object directory.
param$obj	is the token for the created job's parameter object. A zero specifies that the job does not have a parameter object.
minjobsize	is the minimum size, in 16-byte pages, of the newly-created job's memory pool. It is also the initial size of the pool.
maxjobsize	is the maximum pool size, in 16-byte pages, allowed for the new job.

INPUT PARAMETERS
where: (continued)

max$objects	is a word containing the maximum number of objects that this job can contain simultaneously. ØFFFFH specifies an unlimited number.
max$tasks	is a word containing the maximum number of tasks that can exist simultaneously within this job. ØFFFFH specifies an unlimited number.
max$prior	is a byte specifying this job's maximum task priority from Ø (high) to 255 (low). A zero specifies that the maximum priority equals the maximum task priority of the calling job.
excep$hdler$offset excep$hdler$base	point to the first instruction of the job's default exception handler. Zeros specify that the system default exception handler is to be used.
job$flags	contains job information needed by the RMX/86 nucleus. If any tasks in the job use the 8087 component, the low-order bit should be set to one.
global$job	is a token for a job whose object directory is to be used as the global logical-name directory for the job being created. A zero specifies that the caller's global job is to be used.
user	is a token for the job's default user object. A zero specifies that the calling job's default user is the new job's default also.
msg$mbox	is a token for the mailbox that will receive the exit message when this job is exited. A zero specifies that no message is desired. (See EXITIOJOB for exit message formats.

RETURN VALUE

job	is a token for the job created by the loader, or zero if no job was created.

S$LOAD (continued)

RESULT SEGMENT

result$seg — is the result segment returned to the location designated by return$data$ptr. The format of the result segment varies depending on the load function selected.

load$function = 0

```
DECLARE
 result$seg      STRUCTURE(
   status            WORD,
   error$displace    DWORD,
   error$rec$type    BYTE,
   num$undef$refs    WORD,
   mem$req           WORD,
   mem$received      WORD
);
```

load$function = 1

```
DECLARE
  result$seg     STRUCTURE(
    status            WORD,
    error$displace    DWORD,
    error$rec$type    BYTE,
    num$undef$refs    WORD,
    task$token        TOKEN,
);
```

load$function = 2

```
DECLARE
  result$seg     STRUCTURE(
  status            WORD,
  error$displace    DWORD,
  error$rec$type    BYTE,
  num$undef$refs    WORD,
  init$CS           TOKEN,
  init$IP           WORD,
  init$SS           TOKEN,
  stack$offset      WORD,
  stack$size        WORD,
  init$DS           TOKEN
);
```

where:

status — indicates whether the load operation terminated normally (E$OK) or due to an error (E$IO).

RESULT SEGMENT
where: (continued)

error$displace	if nonzero, defines the displacement within the file of an erroneous object record.
errorrectype	if nonzero, gives the record type of the erroneous object record.
num$undef$refs	is the number of undefined external references encountered during the load.
mem$req	is the desired maximum memory size.
mem$received	is the number of 16-byte pages actually allocated to the new job.
task$token	is the token for the newly-created task.
init$CS	is the initial value of the loaded task's CS register. It is zero if the register is not initialized in the object file.
init$IP	gives the initial value of the loaded task's IP register. It is zero if not initialized in the object file.
init$SS	is a token used to initialize the loaded task's SS register. It is zero if the register is not initialized in the object file.
stack$offset	gives the location of the bottom of the stack relative to SS. It is zero if not initialized.
stack$size	gives the size of the stack. The loaded task's stack pointer should be initialized to the sum of stack$offset and stack$size. This field is zero if not initialized.
init$DS	is a token used to initialize the task's DS register. It is zero if not initialized in the object file.

CONDITION CODES

EOK, EABS$ADDRESS, E$BADGRP, EBADHDR, EBADSEG, ECHECKSUM, EEOF, EFIXUP, E$LFUNC, E$NO$LMEM, E$NO$MEM, E$RECFMT, EREC$LENGTH, E$REC$TYPE, E$REG$INIT, E$SEG$ALLOC.

S$LOOK$UP$CONNECTION

SYNCHRONOUS LOOKUPCONNECTION SYSTEM CALL

The S$LOOK$UP$CONNECTION system call returns information about a file conection to the caller. The caller can specify a connection's logical name to this system call and receive the token associated with that connection in return. The I/O system looks for the name first in the job's logical-name directory. If the name is not found, it then looks in the global directory for the job, and finally in the system's logical-name directory. If the name does not reside in either directory, an exception code is returned.

```
connection = RQ$S$LOOK$UP$CONNECTION(log$name, excep$ptr);
```

INPUT PARAMETERS

log$name	is a pointer to the string giving the logical file name to be looked up.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

connection	is a token for the connection associated with the specified logical name.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$NOT$CONFIGURED, E$PARAM, E$TYPE.

SYNCHRONOUS OPEN SYSTEM CALL

The S$OPEN system call opens a composite file connection for I/O. The connection must be opened before I/O operations such as reading and writing can be performed. S$OPEN also initializes the connection for a particular mode of shared access. The following steps are performed:

- create and initialize I/O buffer(s);
- initialize the file pointer to zero;
- open the connection;
- check the current shared state of the connected file and return a condition code of this state conflicts with the mode of the open request;
- initiate the first read if the open is requested for reading and if at least one buffer has been created;
- check for stream file driver; if present, assume "num$buff" is zero.

```
CALL  RQ$S$OPEN(connection, mode, share, buff$size,
                num$buff, excep$ptr);
```

INPUT PARAMETERS

connection — is a token for the composite connection to be opened.

mode — is a byte giving the mode of the open request. Possible values are:

1	Read only
2	Write only
3	Read and write (update)

share — is a byte specifying the kind of sharing desired. Possible values are:

Ø	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

buff$size — is a word giving the size (in bytes) of the buffer(s) to be used by the I/O system. A zero indicates the default buffer size is to be used. The default size is the size specified in the S$CREATE$FILE or S$ATTACH$FILE call that created this connection.

S$OPEN (continued)

INPUT PARAMETERS (continued)

num$buff — is a byte specifying the number of buffers to be used for multiple buffering. For locate mode, at least one buffer is needed. For move mode, zero or more can be specified. The maximum allowed is two buffers. If two buffers are specified, multiple buffering is implied and read-ahead and write-ahead will be performed.

The default value for this parameter is the number of buffers specified in the S$CREATE$FILE or S$ATTACH$FILE call that created this connection. The default is indicated by specifying ØFFH for this parameter.

excep$ptr — is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENT

The current shared status of the file must be compatible with the requested mode of open.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$LIMIT, EMEM, ENOBUFF, ENOTCONFIGURED, E$NUMBUFF, E$PARAM, E$SHARE, E$TYPE.

SYNCHRONOUS READ$LOCATE SYSTEM CALL

The S$READ$LOCATE system call reads a collection of bytes from a file designated by the caller. After this call has been completed, the location addressed by the "data$ptr" parameter contains a pointer to the SIOS buffer containing the bytes that were read.

The actual number of bytes read is always less than or equal to the number of bytes requested in the system call unless an exceptional condition occurred. A zero is returned when an end-of-file was encountered and nothing was read. This is a normal condition and E$OK is the condition code returned.

NOTE

If the number of bytes being read spans two SIOS buffers, only the bytes in the currently active buffer are read. The other buffer can be read by issuing a second S$READ$LOCATE call.

S$READ$LOCATE is often used in conjunction with the S$WRITE$UPDATE system call to update files. See the description of S$WRITE$UPDATE for further details.

```
bytes$read = RQ$S$READ$LOCATE(connection, data$ptr, count,
                                        excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open connection to be read.
data$ptr	is a pointer to the SIOS buffer to receive the data being read.
count	is a word giving the number of bytes to be read.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

bytes$read	is a word giving the actual number of bytes read. A zero indicates the end-of-file was reached with no bytes being read.

FILE ACCESS REQUIREMENT

The spcified connection must be open for reading.

S$READ$LOCATE (continued)

CONDITION CODES

EOK, EBADBLK, EBADCALL, E$CONTEXT, E$DEVNR, E$EXIST, E$FATALHW, E$FLUSHING, E$IO, E$LIMIT, E$MEM, E$MIXREADMODE, E$NOBUFF, ENOTCONFIGURED, E$NUMBUFF, E$PARITY, E$TYPE.

SYNCHRONOUS READ$MOVE SYSTEM CALL

The S$READ$MOVE call reads a collection of bytes from a file to the specified caller buffer.

The actual number of bytes read is always less than or equal to the number of bytes requested in the system call, unless an exceptional condition occurred. A zero is returned if an end-of-file is encountered and no bytes were read. This is a normal condition and returns an E$OK condition code.

```
byte$read = RQ$S$READ$MOVE(connection, buff$ptr, count,
                           excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open connection to be read.
buff$ptr	is a pointer to the user buffer that is to receive the data read.
count	is a word giving the number of bytes to be read.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

bytes$read	is a word giving the actual number of bytes read.

FILE ACCESS REQUIREMENT

The specified connection must be open for reading.

CONDITION CODES

EOK, EBADBLK, EBADCALL, E$CONTEXT, E$DEVNR, E$EXIST, E$FATALHW, E$FLUSHING, E$IO, E$LIMIT, E$MEM, E$MIXREADMODE, E$NOBUFF, ENOTCONFIGURED, ENUMBUFF, E$PARITY, E$TYPE.

S$RENAME$FILE

SYNCHRONOUS RENAME$FILE SYSTEM CALL

The S$RENAME$FILE system call applies to named files only. It is called to change the name of a file (that is, the name by which it is cataloged in its parent directory). A renamed data file can be recataloged in a different parent directory, so long as that directory is on the same volume as the file's original parent. Renamed directory files must keep the same parent, however.

The caller is assumed to be the default user for the job.

```
CALL  RQ$S$RENAME$FILE(old$path, new$path, excep$ptr);
```

INPUT PARAMETERS

old$path	is a pointer to the string giving the prefix and subpath to the file being renamed.
new$path	is a pointer to the string giving a new prefix and subpath for the file. This parameter must specify a nonexistent file. For a data file, it may specify a different directory than the original parent, but the new parent must be on the same volume. A directory file can only be renamed within its parent directory, however.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENTS

The caller must have delete access to the original file and must have add-entry access to the file's parent directory.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FEXIST, E$FLUSHING, E$FNEXIST, E$FTYPE, E$IFDR, E$IO, E$LIMIT, E$MEM, E$NOPREFIX, E$NOT$CONFIGURED, E$NOUSER, E$PARAM, E$SUPPORT, E$TYPE.

SYNCHRONOUS SEEK SYSTEM CALL

The S$SEEK system call applies to physical and named files only. It moves the file pointer to the specified byte position in the file. The designated connection must be open at the time S$SEEK is called.

```
CALL  RQ$S$SEEK(connection, mode, hi$ptr$move, low$ptr$move,
                excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open connection file whose file pointer is to be moved.
mode	is a byte describing the movement of the file pointer. Possible values are:
	1 Move pointer back by "ptr$move" amount. If this action moves the pointer past the beginning of the file, the pointer is set to byte position zero.
	2 Set the pointer to the location specified by "ptr$move."
	3 Move the file pointer forward by "ptr$move" amount.
	4 Move the pointer to the end of the file, minus the "ptr$move" specified.
hiptrmove lowptrmove	is a word pair giving the position in the file to which the file pointer is to be moved. The interpretation of "ptr$move depends on the mode setting, as explained above.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, E$IFDR, E$IO, E$LIMIT, E$MEM, ENOTCONFIGURED, E$PARAM, E$TYPE.

S$SPECIAL

SYNCHRONOUS SPECIAL SYSTEM CALL

The S$SPECIAL system call applies to physical files only. It lets the caller perform special device-level functions.

The special function to be done can be specified using either of two methods. If the function requires little supporting information (such as a function to rewind a magnetic tape), its code can be specified directly as the "spec$func" parameter in the system call.

Where additional information is needed, the user must create an I/O parameter block as described below. The special function is specified indirectly, as a field in this parameter block. The block is then addressed by the "ioparm$ptr" in the system call.

If only the "spec$func" parameter is used to specify the special function, the "ioparm$ptr" parameter is specified as zero.

```
CALL  RQ$S$SPECIAL(connection, spec$func, ioparm$ptr,
                   ioresp$ptr, excep$ptr);
```


INPUT PARAMETERS

connection	is a token for a connection to the file where the special function is to be performed.
spec$func	is a word (code) that allows the user to pass a special function to a file driver without being required to set up a parameter block.
ioparm$ptr	is a pointer to a parameter block. The contents of the parameter block depends on the requirements of the file driver being used to implement the special function. If the file driver requires no parameter for the function being requested, then the ioparm$ptr can be zero. If the function does require parameters, you must build the parameter block to satisfy the requirements of the specific file driver.
ioresp$ptr	is a pointer to the location that receives an I/O result segment indication that the special function has been completed (see Appendix C). A zero indicates that no result is wanted.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FLUSHING, E$IDDR, E$IFDR, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

S$TRUNCATE$FILE

SYNCHRONOUS TRUNCATE$FILE SYSTEM CALL

The S$TRUNCATE$FILE system call applies to named files only. When called, it truncates a file at the current setting of the file pointer and frees all allocated space beyond the pointer. S$SEEK can be called to position the file pointer before calling S$TRUNCATE$FILE. If the pointer is at or beyond the end-of-file, no operation is performed.

Truncation takes effect immediately and differs from S$DELETE$FILE in this respect. A file cannot be deleted until all connections to the file have been severed.

```
CALL  RQ$S$TRUNCATE$FILE(connection, excep$ptr);
```

INPUT PARAMETER

connection	is a token for an open connection to the file being truncated.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

FILE ACCESS REQUIREMENT

The file to be truncated must be open for writing.

CONDITION CODES

EOK, EBAD$CALL, E$CONTEXT, E$EXIST, E$FACCESS, E$FLUSHING, E$IFDR, EIO, ELIMIT, EMEM, ENOT$CONFIGURED, E$TYPE.

SYNCHRONOUS WRITE$MOVE SYSTEM CALL

The S$WRITE$MOVE system call writes a collection of bytes from a user buffer to a designated file. The byte count returned equals the number of bytes requested unless an exceptional condition occurs.

```
bytes$written = RQ$S$WRITE$MOVE(connection, buff$ptr,
                                count, excep$ptr);
```

INPUT PARAMETERS

connection	is a token for the open connection to be written.
buff$ptr	is a pointer to the user buffer containing the data to be written.
count	is a word giving the number of bytes to be written.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

bytes$written	is a word giving the actual number of bytes written.

FILE ACCESS REQUIREMENTS

The caller must have update access to the file. The specified file must be open for writing.

CONDITION CODES

EOK, EBADBLK, EBADCALL, E$CONTEXT, E$DEVNR, E$EXIST, E$FATALHW, E$FLUSHING, E$IO, E$LIMIT, E$MEM, E$MIXWRITEMODE, E$NOBUFF, ENOTCONFIGURED, E$NUMBUFF, E$PARITY, E$SPACE, E$SUPPORT, E$TYPE.

S$WRITE$UPDATE

SYNCHRONOUS WRITE$UPDATE SYSTEM CALL

The S$WRITE$UPDATE system call writes data from a specified SIOS buffer to an open composite connection. This call can be combined with a call to S$READ$LOCATE to update file information.

A file update is initiated when S$READ$LOCATE reads data from a designated connection to the SIOS buffer specified by its "data$ptr" parameter S$READ$LOCATE. The data in the buffer can then be modified as necessary. Finally, S$WRITE$UPDATE is invoked to write data from the same SIOS buffer back to the same open connection.

When the write operation has been completed, this call returns the actual number of bytes written. Normally, this count equals the number of bytes requested. The number of bytes requested, in turn, should be less than or equal to the number of bytes read by the preceding S$READ$LOCATE call.

```
bytes$written = RQ$S$WRITE$UPDATE(connection, count,
                                  data$ptr, excep$ptr);
```


INPUT PARAMETERS

connection	is a token for the open connection to be written. An S$READ$LOCATE call must have been the most recent operation on this connection.
count	is a word giving the number of bytes to be written. This value must be less than or equal to the value returned by the preceding S$READ$LOCATE call.
excep$ptr	is a pointer to the location that receives the condition code resulting from this call.

RETURN VALUE

bytes$written	is a word giving the actual number of bytes written.

FILE ACCESS REQUIREMENTS

The caller must have update access to the specified file. The file must be open for reading and writing.

CONDITION CODES

EOK, EBADBLK, EBADCALL, E$CONTEXT, E$DEVNR, E$EXIST, E$FATAHW, E$FLUSHING, E$IO, E$LIMIT, E$MEM, E$NOBUFF, E$NOT$CONFIGURED, E$NREADLOCATE, E$NUMBUFF, E$PARITY, E$SPACE, E$SUPPORT, E$TYPE.

APPENDIX A

SUMMARY OF I/O SYSTEM CALLS

This appendix summarizes RMX/86 I/O system calls by function and, where applicable, indicates the file types to which they apply:

PF	Physical file
SF	Stream file
NF	Named data file
ND	Named directory file

The page reference listed with each call points to the PL/M calling sequence and detailed description for the call.

JOB-LEVEL SYSTEM CALLS

System Call	Function	Page
CREATEIOJOB	Create job & initial task.	8-43
EXITIOJOB	Terminate job.	8-48
SET$DEFAULT$PREFIX	Set default prefix for job.	8-83
GET$DEFAULT$PREFIX	Inspect default prefix.	8-51
SET$DEFAULT$USER	Set default user for job.	8-84
GET$DEFAULT$USER	Inspect default user.	8-52

GET TIME/DATE SYSTEM CALLS

System Call	Function	Page
GET$TIME	Get date/time value in internally-stored format.	8-53
GET$TIME$STRING	Get date/time value in user-oriented format.	8-54

LOAD FILE/TASK SYSTEM CALL

System Call	Function	Page
S$LOAD	Synchronous load.	8-93

CREATE-FILE-CONNECTION SYSTEM CALLS

System Call	Function	PF	SF	NF	ND	Page
A$CREATE$FILE	Asynchronous data file creation.	*	*	*		8-15
H$CREATE$FILE	Hybrid data-file creation.	*	*	*		8-60
S$CREATE$FILE	Synchronous data-file creation.	*	*	*		8-79
A$ATTACH$FILE	Asynchronous attach file.	*	*	*	*	8-8
H$ATTACH$FILE	Hybrid attach file.	*	*	*	*	8-55
S$ATTACH$FILE	Synchronous attach file.	*	*	*	*	8-72
A$CREATE$DIRECTORY	Asynchronous create directory.				*	8-13
H$CREATE$DIRECTORY	Hybrid create directory.				*	8-58
S$CREATE$DIRECTORY	Synchronous create directory.				*	8-77

FILE MODIFICATION SYSTEM CALLS

System Call	Function	PF	SF	NF	ND	Page
A$CHANGE$ACCESS	Asynchronous change access rights to file.			*	*	8-10

FILE MODIFICATION SYSTEM CALLS (continued)

		PF	SF	NF	ND	Page
H$CHANGE$ACCESS	Hybrid change access rights to file.			*	*	8-56
S$CHANGE$ACCESS	Synchronous change access rights to file.			*	*	8-74
A$RENAME$FILE	Asynchronous rename file.			*	*	8-35
H$RENAME$FILE	Hybrid rename file.			*	*	8-70
S$RENAME$FILE	Synchronous rename			*	*	8-104

FILE INPUT/OUTPUT SYSTEM CALLS

System Call	Function	PF	SF	NF	ND	Page
A$OPEN	Asynchronous open file.	*	*	*		8-31
S$OPEN	Synchronous open file.	*	*	*		8-99
A$SEEK	Asynchronous move file pointer.	*		*		8-37
S$SEEK	Synchronous move file pointer.	*		*		8-105
A$READ	Asynchronous read file.	*	*	*		8-33
S$READ$LOCATE	Synchronous read to SIOS buffer.	*	*	*		8-101
S$READ$MOVE	Synchronous read to caller buffer.	*	*	*		8-103
A$WRITE	Asynchronous write file.	*	*	*		8-41
S$WRITE$MOVE	Synchronous write from caller buffer.	*	*	*		8-109

FILE INPUT/OUTPUT SYSTEM CALLS (continued)

		PF	SF	NF	ND	Page
S$WRITE$UPDATE	Synchronous write from SIOS buffer used by S$READ$-LOCATE.	*	*	*		8-110
A$CLOSE	Asynchronous close file.	*	*	*		8-11
S$CLOSE	Synchronous close file.	*	*	*		8-76

DEVICE-LEVEL FUNCTION SYSTEM CALLS

System Call	Function	PF	SF	NF	ND	Page
A$SPECIAL	Asynchronous perform device-level function.	*				8-38
S$SPECIAL	Synchronous perform device-level function.	*				8-106

GET STATUS/ATTRIBUTE SYSTEM CALLS

System Call	Function	PF	SF	NF	ND	Page
AGETCONNECTION$STATUS	Asynchronous get connection status.	*	*	*	*	8-21
SGETCONNECTION$STATUS	Synchronous get connection status.	*	*	*		8-85
AGETFILE$STATUS	Asynchronous get file status.	*	*	*	*	8-25
HGETFILE$STATUS	Hybrid get file status.	*	*	*	*	8-64
SGETFILE$STATUS	Synchronous get file status.	*	*	*	*	8-88

GET STATUS/ATTRIBUTE SYSTEM CALLS (continued)

		P F	S F	N F	N D	Page
AGETDIRECTORY$ENTRY	Asynchronous inspect directory entry.				*	8-23
AGETPATH$COMPONENT	Asynchronous obtain path name from connection token.			*	*	8-30
H$LOOK$UP$CONNECTION	Hybrid obtain connection token from logical name.	*	*	*	*	8-69
S$LOOK$UP$CONNECTION	Synchronous obtain connection token from logical name.	*	*	*	*	8-98

DELETE CONNECTION/FILE SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$DELETE$CONNECTION	Asynchronous delete file connection.	*	*	*	*	8-18
H$DELETE$CONNECTION	Hybrid delete file connection.	*	*	*	*	8-62
S$DELETE$CONNECTION	Synchronous delete high-level file connection.	*	*	*		8-81
A$TRUNCATE	Asynchronous truncate file.			*		8-40
S$TRUNCATE$FILE	Synchronous truncate file.			*		8-108
A$DELETE$FILE	Asynchronous delete file.		*	*	*	8-19
H$DELETE$FILE	Hybrid delete file.		*	*	*	8-63
S$DELETE$FILE	Synchronous delete file.		*	*	*	8-82

APPENDIX B

PL/M EXTERNAL PROCEDURES

This appendix lists the PL/M external procedures declared for the I/O system calls described in this manual. The procedures are listed in the same sequence as the system call descriptions in Chapter 8.

```
rq$a$attach$file: PROCEDURE(user, prefix, path$p, resp$mbox,
                            excep$p) EXTERNAL;
   DECLARE
     user                   WORD,
     prefix                 WORD,
     path$p                 POINTER,
     resp$mbox              WORD,
     excep$p                POINTER:
END  rq$a$attach$file;


rq$a$change$access: PROCUEDURE(user, prefix, path$p, id,
                               access, resp$mbox, excep$p)
                               EXTERNAL;
    DECLARE
     user                   WORD,
     prefix                 WORD,
     path$p                 POINTER,
     id                     WORD,
     access                 BYTE,
     resp$mbox              WORD,
     excep$p                POINTER:
END  rq$a$change$access;


rq$a$close: PROCEDURE(conn, resp$mbox, excep$p) EXTERNAL;
    DECLARE
     conn                   WORD,
     resp$mbox              WORD,
     excep$p                POINTER;
END  rq$a$close;


rq$a$create$directory: PROCEDURE(user, prefix, path$p,
                                 owner$access, resp$mbox,
                                 excep$p) EXTERNAL;
    DECLARE
     user                   WORD,
     prefix                 WORD,
     path$p                 POINTER,
     owner$access           BYTE,
     resp$mbox              WORD,
     excep$p                POINTER;
END  rq$a$create$directory;
```

```
rq$a$create$file: PROCEDURE(user, prefix, path$p, owner$access,
                            gran, high$size, low$size,
                            must$create, resp$mbox, excep$p)
                            EXTERNAL;
    DECLARE
      user                   WORD,
      prefix                 WORD,
      path$p                 POINTER,
      owner$access           BYTE,
      gran                   WORD,
      high$size              WORD,
      low$size               WORD,
      must$create            BYTE,
      resp$mbox              WORD,
      excep$p                POINTER;
END  rq$a$create$file;


rq$a$delete$connection: PROCEDURE(conn, resp$mbox, excep$p)
                                  EXTERNAL;
    DECLARE
      conn                   WORD,
      resp$mbox              WORD,
      excep$p                POINTER;
END  rq$a$delete$connection;


rq$a$delete$file: PROCEDURE(user, prefix, path$p, resp$mbox,
                            excep$p) EXTERNAL;
    DECLARE
      user                   WORD,
      prefix                 WORD,
      path$p                 POINTER,
      resp$mbox              WORD,
      excep$p                POINTER;
END  rq$a$delete$file;


rq$a$connection$status: PROCEDURE(conn, resp$mbox, excep$p)
                                  EXTERNAL;
    DECLARE
      conn                   WORD,
      resp$mbox              WORD,
      excep$p                POINTER;
END  rq$a$connection$status;


rq$a$get$directory$entry: PROCEDURE(conn, entry$num, resp$mbox,
                                    excep$p) EXTERNAL;
    DECLARE
      conn                   WORD,
      entry$num              WORD,
      resp$mbox              WORD,
      excep$p                POINTER;
END  rq$a$get$directory$entry;
```

```
rq$a$get$file$status: PROCEDURE(conn, resp$mbox, excep$p)
                                  EXTERNAL;
    DECLARE
      conn                        WORD,
      resp$mbox                   WORD,
      excep$p                     POINTER;
END  rq$a$get$file$status;


rq$a$get$path$component: PROCEDURE(conn, resp$mbox, excep$p)
                                     EXTERNAL;
    DECLARE
      conn                        WORD,
      resp$mbox                   WORD,
      excep$p                     POINTER;
END  rq$a$get$path$component;


rq$a$open: PROCEDURE(conn, mode, share, resp$mbox, excep$p)
                     EXTERNAL;
    DECLARE
      conn                        WORD,
      mode                        BYTE,
      share                       BYTE,
      resp$mbox                   WORD,
      excep$p                     POINTER;
END  rq$a$open;


rq$a$special$: PROCEDURE(conn, spec$func,parms$p,resp$mbox,
                         excep$p)  EXTERNAL;
    DECLARE
      conn                        WORD,
      spec$func                   WORD,
      parms$p                     POINTER,
      resp$mbox                   WORD,
      excep$p                     POINTER;
END  rq$a$special$;


rq$a$read: PROCEDURE(conn, buff$p, count, resp$mbox, excep$p)
                     EXTERNAL;
    DECLARE
      count                       WORD,
      conn                        WORD,
      buff$p                      POINTER,
      resp$mbox                   WORD,
      excep$p                     POINTER;
END  rq$a$read;
```

```
rq$a$rename$file: PROCEDURE(conn, user, prefix, path$p,
                            resp$mbox, excep$p) EXTERNAL;
    DECLARE
      conn                  WORD,
      user                  WORD,
      prefix                WORD,
      path$p                POINTER,
      resp$mbox             WORD,
      excep$p               POINTER;
END  rq$a$rename$file;

rq$a$seek: PROCEDURE(conn, mode, hi$ptr$move, low$ptr$move,
                     resp$mbox, excep$p) EXTERNAL;
    DECLARE
      conn                  WORD,
      mode                  BYTE,
      hi$ptr$move           WORD,
      low$ptr$move          WORD,
      resp$mbox             WORD,
      excep$p               POINTER;
END  rq$a$seek;

rq$a$truncate$: PROCEDURE(conn, resp$mbox, excep$p) EXTERNAL;
    DECLARE
      conn                  WORD,
      resp$mbox             WORD,
      excep$p               POINTER;
END  rq$a$truncate$file;

rq$a$write: PROCEDURE(conn, buff$p, count, resp$mbox, excep$p)
                      EXTERNAL;
    DECLARE
      conn                  WORD,
      buff$p                POINTER,
      count                 WORD,
      resp$mbox             WORD,
      excep$p               POINTER;
END  rq$a$write;

rq$create$IO$job: PROCEDURE(job$p, task$addr, stack$p,
                            prefix$p, log$name$p, msg$mbox,
                            excep$p) WORD EXTERNAL;
    DECLARE
      job$p                 POINTER,
      task$addr             POINTER,
      stack$p               POINTER,
      prefix$p              POINTER,
      log$name$p            POINTER,
      msg$mbox              WORD,
      excep$p               POINTER;
END  rq$create$IO$job;
```

```
rq$delete$package: PROCEDURE (package, excep$p) EXTERNAL;
    DECLARE
      package                   WORD,
      excep$p                   POINTER;
END  rq$delete$package;


rq$exit$IO$job: PROCEDURE(user$exception$code, return$data,
                          return$data$len, excep$p) EXTERNAL;
    DECLARE
      user$exception$code       WORD,
      return$data               POINTER,
      return$data$len           WORD,
      excep$p                   POINTER;
END  rq$exit$IO$job;


rq$get$default$prefix: PROCEDURE(job$t, excep$p) WORD EXTERNAL;
    DECLARE
      job$t                     WORD,
      excep$p                   POINTER;
END  rq$get$default$prefix;


rq$get$default$user: PROCEDURE(job$t, excep$p) WORD EXTERNAL;
    DECLARE
      job$t                     WORD,
      excep$p                   POINTER;
END  rq$get$default$user;


rq$get$time: PROCEDURE (excep$p) DWORD EXTERNAL;
    DECLARE
      excep$p                   POINTER;
END  rq$get$time;


rq$get$time$string: PROCEDURE(dt$p, excep$p) EXTERNAL;
    DECLARE
      dt$p                      POINTER,
      excep$p                   POINTER;
END  rq$get$time$string;


rq$h$attach$file: PROCEDURE(l$name, path, excep$p) WORD
                            EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      excep$p                   POINTER;
END  rq$h$attach$file;
```

```
rq$h$change$access: PROCEDURE(path, mode, name, access, excep$p)
                                 EXTERNAL;
    DECLARE
      path                      POINTER,
      mode                      BYTE,
      name                      POINTER,
      access                    BYTE,
      excep$p                   POINTER;
END  rq$h$change$access;


rq$h$create$directory: PROCEDURE(l$name, path, excep$p) WORD
                                    EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      excep$p                   POINTER;
END  rq$h$create$directory;

rq$h$create$file: PROCEDURE(l$name, path, excep$p) WORD
                               EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      excep$p                   POINTER;
END  rq$h$create$file;

rq$h$delete$connection: PROCEDURE(conn, excep$p) EXTERNAL;
    DECLARE
      conn                      WORD,
      excep$p                   POINTER;
END  rq$h$delete$connection;

rq$h$delete$file: PROCEDURE(path, excep$p) EXTERNAL;
    DECLARE
      path                      POINTER,
      excep$p                   POINTER;
END  rq$h$delete$file;

rq$h$get$file$status: PROCEDURE(path, file$info$p, excep$p)
                                   EXTERNAL;
    DECLARE
      path                      POINTER,
      file$info$p               POINTER,
      excep$p                   POINTER;
END  rq$h$get$file$status;
```

```
rq$h$look$up$connection: PROCEDURE(l$name, excep$p) WORD
                                    EXTERNAL;
    DECLARE
      l$name                    POINTER,
      excep$p                   POINTER;
END  rq$h$look$up$connection;


rq$h$rename$file: PROCEDURE(old$path, new$path, excep$p)
                            EXTERNAL;
    DECLARE
      old$path                  POINTER,
      new$path                  POINTER,
      excep$p                   POINTER;
END  rq$h$rename$file;


rq$inspect$package: PROCEDURE(package, tok$list$p, excep$p)
                             EXTERNAL;
    DECLARE
      package                   WORD,
      tok$list$p                POINTER,
      excep$p                   POINTER;
END  rq$inspect$package;


rq$s$attach$file: PROCEDURE(l$name, path, buff$size, num$buff,
                            excep$p) WORD EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      buff$size                 WORD,
      num$buff                  BYTE,
      excep$p                   POINTER;
END  rq$s$attach$file;


rq$s$change$access: PROCEDURE(path, mode, name, access, excep$p)
                             EXTERNAL;
    DECLARE
      path                      POINTER,
      mode                      BYTE,
      name                      POINTER,
      access                    BYTE,
      excep$p                   POINTER;
END  rq$s$change$access;


rq$s$close: PROCEDURE(conn, excep$p) EXTERNAL;
    DECLARE
      conn                      WORD,
      excep$p                   POINTER;
END  rq$s$close;
```

```
rq$s$create$directory: PROCEDURE(l$name, path, excep$p)
                                  WORD EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      excep$p                   POINTER;
END  rq$s$create$directory


rq$s$create$file: PROCEDURE(l$name, path, buff$size, num$buff,
                            excep$p) WORD EXTERNAL;
    DECLARE
      l$name                    POINTER,
      path                      POINTER,
      buff$size                 WORD,
      num$buff                  BYTE,
      excep$p                   POINTER;
END  rq$s$create$file;


rq$s$delete$connection: PROCEDURE(conn, excep$p) EXTERNAL;
    DECLARE
      conn                      WORD,
      excep$p                   POINTER;
END  rq$s$delete$connection;


rq$s$delete$file: PROCEDURE(path, excep$p) EXTERNAL;
    DECLARE
      path                      POINTER,
      excxep$p                  POINTER;
END  rq$s$delete$file;


rq$set$default$prefix: PROCEDURE(job$t, prefix, excep$p)
                                     EXTERNAL;
    DECLARE
      job$t                     WORD,
      prefix                    WORD,
      excep$p                   POINTER;
END  rq$set$default$prefix;


rq$set$default$user: PROCEDURE(job$t, user, excep$p) EXTERNAL;
    DECLARE
      job$t                     WORD,
      user                      WORD,
      excep$p                   POINTER;
END  rq$set$default$user;
```

```
rq$s$get$connection$status: PROCEDURE(conn, conn$info$p,
                                      excep$p) EXTERNAL;
    DECLARE
      conn                    WORD,
      conn$info$p             POINTER,
      excep$p                 POINTER;
END  rq$s$get$connection$status;


rq$s$get$file$status: PROCEDURE(path, file$info$p, escep$p)
                                 EXTERNAL;
    DECLARE
      path                    POINTER,
      file$info$p             POINTER,
      excep$p                 POINTER;
END  rq$s$get$file$status;


rq$s$load: PROCEDURE(path, load$func, job$data$p, reg$val$p,
                     excep$p) WORD EXTERNAL;
    DECLARE
      path                    POINTER,
      load$func               BYTE,
      job$data$p              POINTER,
      reg$val$p               POINTER,
      excep$p                 POINTER;
END  rq$s$load;


rq$s$look$up$connection: PROCEDURE (l$name, excep$p) WORD
                                    EXTERNAL;
    DECLARE
      l$name                  POINTER,
      excep$p                 POINTER;
END  rq$s$look$up$connection;


rq$s$open: PROCEDURE(conn, mode, share, buff$size, num$buff,
                     excep$p) EXTERNAL;
    DECLARE
      conn                    WORD,
      mode                    BYTE,
      share                   BYTE,
      buff$size               WORD,
      num$buff                BYTE,
      excep$p                 POINTER;
END  rq$s$open;
```

```
rq$s$read$locate: PROCEDURE(conn, data$p, count, excep$p) WORD
                            EXTERNAL;
    DECLARE
      conn                   WORD,
      data$p                 POINTER,
      count                  WORD,
      excep$p                POINTER;
END  rq$s$read$locate;

rq$s$read$move: PROCEDURE(conn, buff$p, count, excep$p) WORD
                          EXTERNAL;
    DECLARE
      conn,                  WORD,
      buff$p                 POINTER,
      count                  WORD,
      excep$p                POINTER;
END  rq$s$read$move;

rq$s$rename$file: PROCEDURE(old$path, new$path, excep$p)
                            EXTERNAL;
    DECLARE
      old$path               POINTER,
      new$path               POINTER,
      excep$p                POINTER;
END  rq$s$rename$file;

rq$s$seek: PROCEDURE(conn, mode, hi$ptr$move, low$ptr$move,
                     excep$p) EXTERNAL;
    DECLARE
      conn                   WORD,
      mode                   BYTE,
      hi$ptr$move            WORD,
      low$ptr$move           WORD,
      excep$p                POINTER;
END  rq$s$seek;

rq$s$special:  PROCEDURE(conn, spec$func, parms$p, ioresp$p,
                         excep$p) EXTERNAL;
    DECLARE
      conn                   WORD,
      spec$func              WORD,
      parms$p                POINTER,
      ioresp$p               POINTER,
      excep$p                POINTER;
END  rq$s$special;

rq$s$truncate$file: PROCEDURE(conn, excep$p) EXTERNAL;
    DECLARE
      conn                   WORD,
      excep$p                POINTER;
END                                        rq$s$truncate$file;
```

```
rq$s$write$move: PROCEDURE(conn, buff$p, count, excep$p) WORD
                           EXTERNAL;
    DECLARE
      conn                    WORD,
      buff$p                  POINTER,
      count                   WORD,
      excep$p                 POINTER;
END  rq$s$write$move;


rq$s$write$update: PROCEDURE(conn, count, excep$p) WORD
                           EXTERNAL;
    DECLARE
      conn                    WORD,
      count                   WORD,
      excep$p                 POINTER;
END  rq$s$write$update;
```

APPENDIX C

I/O REQUEST/RESULT SEGMENT

An I/O request/result segment is a data structure used to request I/O from a device driver and to indicate completion of an asynchronous I/O system call. Only the designer of a device driver needs to be concerned with the interpretation of this structure beyond its first three fields. Consequently, only the first three fields are defined below.

When a task makes an asynchronous system call, it expects a connection or an I/O result segment to be returned to the mailbox specified by the "resp$mbox" parameter. The I/O result segment includes a status field containing "E$OK" if the call was completed successfully, or an asynchronous exceptional-condition code if an error occurred. The result segment also contains the actual number of bytes read or written, if appropriate.

SEGMENT STRUCTURE

The I/O request/result segment is structured as follows:

```
DECLARE
   iors    STRUCTURE(
      status              WORD,
      unit$status         WORD,
      actual              WORD,
      actual$fill         WORD,
      device              WORD,
      unit                BYTE,
      func                BYTE,
      spec$func           WORD,
      dev$loc             DWORD,
      buff$ptr            POINTER,
      count               DWORD,
      aux$ptr             POINTER,
      link$for            POINTER,
      link$back           POINTER,
      resp$mbox           WORD,
      done                BYTE
);
```

SEGMENT STRUCTURE (continued)
where:

status	indicates how the operation completed. "E$OK" indicates successful completion; "E$IO" indicates an error condition.

SEGMENT STRUCTURE (continued)
where:

unit$status	contains device-dependent error code information and is valid only if status = E$IO. The codes that can be returned to this field are listed in the next section of this appendix.
actual	is the actual number of bytes transferred.

STATUS CODES

The following table lists the asynchronous exceptionalcondition codes returned in the status field of the I/O result segment. The table lists the condition codes, their hexadecimal equivalents, and their interpretations.

Condition Code	Hex	Interpretation
E$OK	0000H	System call was completed successfully.
E$CONTEXT	0005H	System call invoked in illegal context.
E$DEVFD	0022H	Device and file driver incompatibility.
E$DIR$$END	0025H	End of directory.
E$EMPTY$ENTRY	0024H	Empty directory entry.
E$FACCESS	0026H	Access to file not granted.
E$FEXIST	0020H	File already exists.
E$FLUSHING	002CH	Connection is flushing requests.
E$FNEXIST	0021H	File does not exist.
E$FTYPE	0027H	Incompatible file type.
E$IDDR	002AH	Illegal Device Driver Request.
E$IO	002BH	I/O error.
E$LIMIT	0004H	Object limit reached.
E$MEM	0002H	Insufficient memory.
E$SHARE	0028H	Improper file sharing requested.
E$SPACE	0029H	No space left.
E$SUPPORT	0023H	Unsupported request.

APPENDIX D

EXCEPTIONAL-CONDITION CODES

The I/O system checks for exceptional conditions when a system call is invoked. When an exceptional condition occurs, the system issues a condition code describing the error, then either returns to the caller or passes control to an exception handler.

Exceptional-condition codes returned asynchronously (that is, returned in an I/O result segment) are listed in Appendix C. This appendix lists the codes for the exceptional conditions detected synchronously with system call invocation. These codes are returned to the location addressed by the "excep$ptr" field of the external-procedure declaration associated with each call (see Appendix B). The codes are listed below with the hexadecimal equivalents and their interpretations.

PROGRAMMING ERRORS

Condition Code	Hex	Interpretation
EBADCALL	8005H	Call invoked illegally.
E$IFDR	8020H	Illegal file driver request.
E$JNEXIT	0040H	Job containing other jobs tried to exit.
E$MIXREADMODE	80454	READ$LOCATE and READ$MOVE on same connection.
EMIXWRITE$MODE	8046H	WRITE$MOVE and WRITE$UPDATE on same connection.
E$NHUSER		Not high-level user object.
E$NOBUFF	0045H	No SIOS buffers specified.
E$NOPREFIX	8022H	No default prefix.
E$NOUSER	8021H	No default user.
E$NREADLOCATE	0049H	WRITE$UPDATE not preceded by READ$ LOCATE.
E$NUMBUFF	0046H	Too many SIOS buffers specified.

PROGRAMMING ERRORS (continued)

Condition Code	Hex	Interpretation
E$PARAM	8004H	Illegal parameter.
E$PREFIX$SYNTAX	804BH	Illegal prefix syntax.
D$NAME$USED	804CH	Name already in use.
ENOTDEV$NAME	8040H	Not a valid device name.
ENOTCONN$NAME	804EH	Not a valid connection name.
E$TYPE	8002H	Specified object is wrong type.

ENVIRONMENTAL CONDITIONS

E$CANNOT$CLOSE	0044H	Asynchronous I/O error while closing file.
E$CONTEXT	0005H	Call invoked in illegal context.
E$EXIST	0006H	Object referenced by token does not exist.
E$FACCESS	0026H	File access not granted.
E$FEXIST	0020H	File already exists.
E$FNEXIST	0021H	File does not exist.
E$FTYPE	0027H	Incompatible file type.
E$LIMIT	0004H	Calling job or system has reached object limit.
E$MEM	0002H	Insufficient memory.
ENOTCONFIGURED	0008H	
E$SPACE	0029H	No space left.
E$SUPPORT	0023H	Combination of parameters not supported.

LOADER CONDITION CODES

The following condition codes are unique to the loader and are returned by calls S$LOAD.

Condition Code	Hex	Interpretation
EABSADDRESS	0060H	Invalid absolute load address.
EBADGRP	0061H	Invalid group definition record.
EBADHDR	0062H	Invalid header record in object file.
EBADSEG	0063H	Invalid segment definition record.
E$CHECKSUM	0064H	Checksum error.
E$EOF	0065H	Unexpected end of file.
E$FIXUP	0066H	Invalid fixup.
E$LFUNC	0067H	Invalid load function requested.
ENOLMEM	0068H	Insufficient memory to run loader.
ENOMEM	0069H	Insufficient memory to load and run task.
ERECFMT	006AH	Unspecified error in object record.
ERECLENGTH	006BH	Ivalid record length.
ERECTYPE	006CH	Invalid object-record type.
EREGINIT	006DH	Uninitialized register detected; task is not created.
ESEGALLOC	006EH	Segment allocation error.

GLOSSARY

ACCESS MASK. A byte specifying the access rights to a file. Each bit set to one permits the corresponding access. Possible values are:

Bit	Named Data Files	Named Directory Files
0	Delete	Delete
1	Read	Display
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

CONDITION CODE. A code returned when a system call is issued. "E$OK" indicates successful completion of the call. All other codes indicate exceptional or error conditions.

CONNECTION. An RMX/86 object established when a device is attached (device connection) or a file is created or attached (file connection). A connection contains information needed to access the device or file, or to perform I/O on the file.

DEFAULT PREFIX. A user-specified parameter (token for a connection) which can serve as the "prefix" parameter in system calls issued within the job where the default prefix is specified.

DEFAULT USER. A user-specified parameter (token for a user object) which can serve as the "user" parameter in system calls issued within the job where the default user is specified.

DEVICE. Any of a broad spectrum of traditional and non-traditional physical units, such as a line printer or thermistor.

DIRECTORY FILE. A named file whose entries include the names of other directories or named data files and information for accessing these files.

EXCEPTIONAL CONDITION. A condition indicating a result other than successful completion of a system call. The I/O system flags this situation by issuing an exceptional-condition code.

FILE-ACCESS PROTECTION. The I/O system mechanism that limits access to named files to certain specified users. The specific access rights granted to these users can be limited also. (see ACCESS MASK).

FILE DESCRIPTOR. An I/O system internal structure containing information about named files.

FILE GRANULARITY. The size (in bytes) of each logical block to be allocated to a file (for any allocations after its initial preallocation).

FILE NAME. A 1-14 ASCII-characater name used to designate a named file. This is the last name in the subpath string specified when the file is created. The file name is cataloged in the file's parent directory.

JOB. An operating environment and resource bank for tasks.

LOGICAL NAME. A 1-12 ASCII-character symbolic name assigned to a connection object or user object at the hybrid or synchronous levels of I/O operation. This name can be used instead of a token to refer to these objects in subsequent system calls.

LOGICAL-NAME DIRECTORY. A job-level directory used to catalog logical names.

NAMED FILE. A collection of bytes residing on a random-access storage device. A named file can be either a data file or a directory file.

PARAMETER. An item in a PL/M system call syntax description, to be replaced with an acutal value when the call is issued.

PARENT DIRECTORY. That directory containing the file name and access information for a given file.

PATH. The parameter(s) specified to locate a named file within a directory tree. It consists of PREFIX and a SUBPATH .

PHYSICAL FILE. A file associated with a physical device other than a random-access storage device.

PREFIX. A parameter containing a token for a connection or the logical name for such a token. In the case of a physical or stream file, or where the SUBPATH parameter is null in a reference to a named file, the prefix specifies a file being accessed. If the SUBPATH is not null, the prefix indicates the starting point in a directory tree search for the named file being accessed. (See SUBPATH).

RESPONSE MAILBOX. A mailbox used at the asynchronous level to synchronize I/O operations. The result of an asynchronous system call is returned to a response mailbox designated by the caller.

RESULT SEGMENT. A status segment returned to a designated RESPONSE MAILBOX when an asynchronous system call has completed operation.

ROOT DIRECTORY. The base directory in a hierarchical directory tree.

STREAM FILE. A file mechanism allowing tasks to communicate with one another without the intervention of external devices or media.

SUBPATH. A parameter that points to a string describing the route from the starting point in a directory tree (designated by a PREFIX parameter) to the named file being sought. If subpath points to a null string, the PREFIX itself designates the desired file. (See PATH and PREFIX).

SYSTEM CALL. The means by which the applications programmer accesses the facilities of the RMX/86 I/O system.

TASK. The active principle within a job that performs the operations required of the job.

TOKEN. A 16-bit item used to designate an RMX/86 object. Users exchange tokens in system calls to gain access to these objects. (See CONNECTION and USER OBJECT).

USER OBJECT. An RMX/86 object that includes not only the identification of a given user, but also identifies all groups to which he belongs.

VOLUME. The physical storage medium used by a random-access storage device, such as a diskette or hard-disk platter.

VOLUME LABEL. A record on a volume containing information needed to support hierarchical directory trees and named files. This label is used internally by the I/O system.

INDEX

A$ATTACH$FILE, 2-1, 2-4, 8-8
A$CHANGE$ACCESS, 1-6, 2-1, 2-2, 8-10
A$CLOSE, 2-3, 3-3, 8-12
A$CREATE$DIRECTORY, 1-6, 2-1, 2-4, 8-13
A$CREATE$FILE, 1-6, 2-1, 2-4, 8-15
A$DELETE$CONNECTION, 2-4, 8-18
A$DELETE$FILE, 2-4, 8-19
AGETCONNECTION$STATUS, 2-2, 3-3, 8-21
AGETDIRECTORY$ENTRY, 2-2, 3-3, 8-23
AGETFILE$STATUS, 2-2, 8-25
AGETPATH$COMPONENT, 2-2, 3-3, 8-30
A$OPEN, 2-3, 3-3, 8-31
A$READ, 2-3, 3-3, 8-32
A$RENAME$FILE, 2-1, 2-2, 8-35
A$SEEK, 2-3, 3-3, 8-37
A$SPECIAL, 2-4, 3-3, 8-38
A$TRUNCATE, 2-4, 3-3, 8-40
A$WRITE, 2-3, 3-3, 8-41
BOOLEAN, 8-6
BYTE, 8-6
CONNECTION, 8-6
CREATEIOJOB, 5-1, 5-2, 7-1, 8-43
DELETE$PACKAGE, 7-1, 8-47
EABSADDRESS, D-3
EBADCALL, D-1
EBADGRP, D-3
EBADHDR, D-3
EBADSEG, D-3
E$CANNOT$CLOSE, D-2
E$CHEKSUM, D-3
E$CONTEXT, C-2, D-2
E$DEVFD, C-2
EDIREND, C-2
E$EMPTY$ENTRY, C-2
E$EOF, D-3
E$EXIST, D-2
E$FACESS, C-2, D-2
E$FEXIST, C-2, D-2
E$FIXUP, D-3
E$FLUSHING, C-2
E$FNEXIST, C-2, D-2
E$FTYPE, C-2, D-2
E$IDDR, C-2
E$IFDR, D-1
E$IO, C-2
E$JNEXIT, D-1
E$LFUNC, D-3

E$LIMIT, C-2, D-2
E$MEM, C-2, D-2
EMIXREAD$MODE, D-1
EMIXWRITE$MODE, D-1
E$NAME$USED, D-2
E$NHUSER, D-1
ENOLMEM, D-3
ENOMEM, D-3
E$NOBUFF, D-1
E$NOPREFIX, D-1
ENOTCONFIGURED, D-2
ENOTCONN$NAME, D-2
ENOTDEV$NAME, D-2
E$NOUSER, D-1
E$NREADLOCATE, D-1
E$NUMBUFF, D-1
E$OK, 8-6, C-2
E$PARAM, D-2
E$PREFIX$SYNTAX, D-2
ERECFMT, D-3
ERECLENGTH, D-3
ERECTYPE, D-3
EREGINIT, D-3
ESEGALLOC, D-3
E$SHARE, C-2
E$SPACE. C-2, D-2
E$SUPPORT, C-2, D-2
E$TYPE, D-2
EXITIOJOB, 5-1, 7-1, 8-48
GET$DEFAULT$PREFIX, 5-2, 8-51
GET$DEFAULT$USER, 5-2, 8-52
GET$TIME, 6-1, 8-53
GET$TIME$STRING, 6-1, 8-54
H$ATTACH$FILE, 3-1, 3-3, 8-55
H$CHANGE$ACCESS, 1-6, 3-2, 8-56
H$CREATE$DIRECTORY, 1-6, 3-1, 3-3, 8-6
H$CREATE$FILE, 1-6, 3-1, 3-3, 8-60
H$DELETE$CONNECTION, 3-2, 3-3, 8-62
H$DELETE$FILE, 3-2, 3-3, 8-63
HGETFILE$STATUS, 3-2, 8-64
H$LOOK$UP$CONNECTION, 3-2, 8-69
H$RENAME$FILE, 3-2, 8-70
I/O request segment, C-1
I/O result segment, 8-7, C-1
INSPECT$PACKAGE, 7-1, 8-71
LOC86, 7-1
PL/M data types, 8-6
PL/M external procedures, B-1 to B-11
PL/M interface, 8-1 to 8-110
POINTER, 8-6

RMX/86 objects, 8-6
S$ATTACH$FILE, 4-1, 4-2, 4-5, 8-72
S$CHANGE$ACCESS, 1-6, 4-2, 8-74
S$CLOSE, 4-3, 4-4, 8-76
S$CREATE$DIRECTORY, 1-6, 4-1, 4-2, 8-77
S$CREATE$FILE, 1-6, 4-1, 4-2, 4-5, 8-79
S$DELETE$CONNECTION, 4-4, 4-5, 8-81
S$DELETE$FILE, 4-4, 4-5, 8-82
SGETCONNECTION$STATUS, 4-2, 4-3, 8-5
SGETFILE$STATUS, 4-2, 4-3, 8-88
S$LOAD, 7-1, 8-93
S$LOOK$UP$CONNECTION, 4-2, 4-3, 8-98
S$OPEN, 4-1, 4-2, 4-4, 8-99
S$READ$LOCATE, 4-3, 4-4, 8-101
S$READ$MOVE, 4-3, 8-103
S$RENAME$FILE, 4-2, 8-104
S$SEEK, 4-3, 8-105
S$SPECIAL, 4-4, 8-106
S$TRUNCATE$FILE, 4-4, 4-5, 8-108
S$WRITE$MOVE, 8-109
S$WRITE$UPDATE, 4-3, 4-4, 8-110
SET$DEFAULT$PREFIX, 5-2, 8-83
SET$DEFAULT$USER, 5-2, 8-84
STRING, 8-6
TOKEN, 8-6
USER, 8-6
WORD, 8-6
WORLD, 1-6
access list, 1-6
access protection, 1-6
access rights, 1-6, 2-2, 3-2, 4-2
applications programmer, 1-7
asynchronous system calls, 1-7, 2-1 to 2-4, 3-3, 5-2, 8-2, 8-7
attach$file, 1-1
attaching a device, 1-3
blocking, 1-7
buffers, 1-7, 2-3, 4-1, 4-4
children jobs, 5-1
closing a connection, 1-3, 2-3, 2-4, 4-3, 4-4, 4-5
condition code, 8-6
configuration, 6-1
connection, 1-1, 2-1, 2-2, 3-2, 4-3
create$directory, 1-1
create$file, 1-1
creating connections
  to data files, 2-1, 3-1, 4-1, 4-2, A-2
  to directory files, 2-1, 3-1, 4-2, A-2
creating jobs, 5-1
date, 6-1, A-1
deblocking, 1-7

default buffer size, 4-1, 4-2
default path prefix, 5-1, 5-2, 8-2, 8-4
default user objects, 3-1, 4-1, 5-1, 5-2, 8-2
deleting, 2-4, 3-2, 3-3, 4-4, 4-5, 5-1
deleting a connection, 1-3, 2-4, 3-2, 3-3, 4-5, 5-1
deleting a file, 2-4, 3-2, 3-3, 4-4, 4-5
deletion system calls, A-5
device, 1-1, 1-4
device connection, 8-2
device-level system calls, A-4
directory entries, 2-2
directory tree, 1-4, 2-3, 8-2
directory-tree path, 1-4
end of file, 2-4, 4-5
file I/O system calls, A-3
file access protection, 1-6
file connection, 1-1, 3-1, 8-2
file connection status, 2-2, 3-2, 4-2, 4-3
file names, 1-4, 2-2, 8-2
file pointer, 2-3, 2-4, 4-3, 4-5
file status, 2-2, 3-2, 4-2, 4-3
files, 1-1, 2-2, 3-2
files marked for deletion, 2-4, 3-3
groups of users, 1-6
hybrid system calls, 1-7, 3-1 to 3-3, 5-2, 8-2, 8-4
job oriented system calls, 5-1, 5-2, A-1
jobs, 5-1, 5-2
levels of system calls, 1-6
loader, 7-1, A-2
logical name directory, 3-1 to 3-3, 4-1, 4-2, 4-5, 8-2
logical names, 1-7, 3-1 to 3-3, 4-1 to 4-3, 4-5, 5-1, 8-2, 8-4
mailboxes, 1-7, 2-1, 3-1, 4-1, 5-1, 8-4, 8-7
named data files, 1-4, 2-1 to 2-4, 3-1 to 3-3, 4-1 to 4-3, 4-5
named directory files, 1-4, 2-1, 2-2, 2-4, 3-1 to 3-3, 4-2, 4-5
named files, 1-1, 1-4, 2-1 to 2-4, 3-1 to 3-3, 4-1 to 4-4, 4-5
null subpath, 1-4
object file, 7-1
opening a connection, 1-3, 2-3, 4-3
overlapped operations, 4-1
owner of a file, 5-1
package object, 7-1
parent directory, 1-4, 2-2, 2-3, 3-2, 4-2, 8-4
parent jobs, 5-1
path, 1-4, 3-1, 4-1, 5-1, 5-2, 8-2
path syntax, 8-2 to 8-4
physical devices, 1-3
physical files, 1-1, 1-3, 2-1 to 2-4, 3-1, 3-2,
4-1, 4-3, 4-4, 8-2
prefix, 1-4, 3-1, 4-1, 5-1, 5-2, 8-2
random access, 2-3, 4-3

random access device, 1-3, 1-4,
random access volume, 1-4
reading, 2-3, 4-3, 4-4
renaming files, 2-2, 3-2, 4-2
result segments, 2-1, C-1
root directory, 1-4, 2-3, 8-3, 8-4
rq$a$attach$file, B-1
rq$a$change$access, B-1
rq$a$close, B-1
rq$a$connection$status, B-2
rq$a$create$directory, B-1
rq$a$create$file, B-2
rq$a$delete$connection, B-2
rq$a$delete$file, B-2
rq$a$get$directory$entry, B-2
rq$a$get$file$status, B-3
rq$a$get$path$component, B-3
rq$a$open, B-3
rq$a$read, B-3
rq$a$rename$file, B-4
rq$a$seek, B-4
rq$a$special, B-3
rq$a$truncate, B-4
rq$a$write, B-4
rq$create$IO$job, B-4
rq$delete$package, B-5
rq$exit$IO$job, B-5
rqgetdefault$prefix, B-5
rqgetdefault$user, B-5
rqgettime, B-5
rqgettime$string, B-5
rq$h$attach$file, B-5
rq$h$change$access, B-6
rq$h$create$directory, B-6
rq$h$create$file, B-6
rq$h$delete$connection, B-6
rq$h$delete$file, B-6
rq$h$get$file$status, B-6
rq$h$lookupconnection, B-7
rq$h$rename$file, B-7
rq$inspect$package, B-7
rq$s$attach$file, B-7
rq$s$change$access, B-7
rq$s$close, B-7
rq$s$create$directory, B-8
rq$s$create$file, B-8
rq$s$delete$connection, B-8
rq$s$delete$file, B-8
rq$s$get$connection$status, B-9
rq$s$get$file$status, B-9

rq$s$load, B-9
rq$s$lookupconnection, B-9
rq$s$open, B-9
rq$s$read$locate, B-10
rq$s$read$move, B-10
rq$s$rename$file, B-10
rq$s$seek, B-10
rq$s$special, B-10
rq$s$truncate, B-10
rq$s$write$move, B-11
rq$s$write$update, B-11
rqsetdefault$prefix, B-8
rqsetdefault$user, B-8
seeking, 2-3, 4-3
stream file, 1-1, 1-3, 1-4, 2-1 to 2-4, 3-1 to 3-3,
4-1, 4-3, 8-2
subpath, 1-4, 3-2, 4-2, 5-2, 8-2
synchronous system calls, 1-7, 4-1 to 4-5, 5-2, 8-2, 8-4
system calls, 1-1
time, 6-1, A-1
tokens, 1-3, 3-2, 4-3
tree of directories, 1-4
truncating, 2-4, 4-5
updating, 4-4
user, 1-6, 5-1
user object, 1-6, 1-7, 3-1, 4-1, 5-2, 8-2
user-id, 1-6
user/access list, 1-6
volume, 1-3, 3-2, 4-2
writing, 2-3, 4-4

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

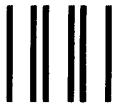
ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

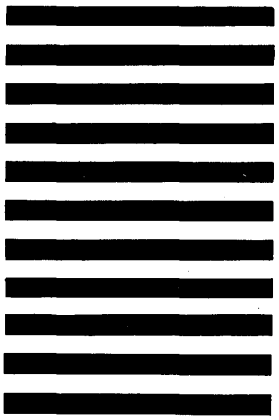
Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications
3065 Bowers Avenue
Santa Clara, CA 95051