DOMAIN System Call Reference (Volume 2, IOS - VFMT) Update 1

Order No. 008857
Revision 00
Software Release 9.2

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: February, 1986

This document was produced using the SCRIBE document preparation system. (SCRIBE is a registered trademark of Unilogic, Ltd.)

Preface

This manual is part of a two-volume set that describes the DOMAIN® system calls. Each volume consists of a section that introduces the system calls followed by sections that describe a separate operating system manager (e.g., the process manager, stream manager, and variable formatting package). The sections that describe the managers are in **alphabetical order by manager name** and consist of a description of the data types used by the manager, the syntax of the manager's programming calls, and the error messages generated by the manager.

For easy organization, we have numbered the pages of this two volume reference set by system manager. For example, the third page in the ACLM section is page ACLM-3.

Volume 1 includes descriptions of the following managers:

ACLM
CAL
EC2
ERROR
GM
GMF
GPR

Volume 2 includes descriptions of the following managers:

IPC	PROC1
MBX	PROC2
MS	RWS
MTS	SIO
MUTEX	SMD
NAME	STREAM
PAD	TIME
PBUFS	TONE
PFM	TPAD
PGM	VEC
PM	VFMT

You should use this manual with the programming handbooks listed under Related Documents. These programming handbooks give detailed instructions about using these programming calls.

Audience

This manual is intended for programmers who are writing application programs using DOMAIN system calls. Readers of this manual should be familiar with FORTRAN, Pascal, or C and the operating system as described in the *DOMAIN System User's Guide*. This manual is not intended as a tutorial document, but as a reference for programmers who need to use operating system services.

Related Documents

The *Programming With General System Calls* handbook, order no. 005506, documents how to write programs that use standard DOMAIN system calls including the ACLM, CAL, EC2, ERROR, MTS, NAME, PAD, PBUFS, PFM, PGM, PM, PROC1, PROC2, RWS, SIO, STREAM, TIME, TONE, TPAD, and VFMT calls.

The *Programming With System Calls for Interprocess Communication* handbook, order no. 005696, documents how to write programs that use the DOMAIN interprocess facilities including the MBX, MS, IPC, MUTEX, and EC2 calls.

The *Programming With DOMAIN 2D Graphics Metafile Resource* handbook, order no. 005097, documents how to write programs that use the DOMAIN 2D Graphics Metafile Resource.

The *Programming With DOMAIN Graphic Primitives* handbook, order no. 005808, documents how to write graphics programs that use the DOMAIN Graphics Primitive Resource.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

UPPERCASE	Uppercase words or characters in formats and command descriptions represent keywords that you must use literally.
lowercase	Lowercase words or characters in formats and command descriptions represent values that you must supply.
[ ]	Square brackets enclose optional items.
{ }	Braces enclose a list from which you must choose an item.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you type the character.
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that we have omitted irrelevant parts of a figure or example.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference* manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same description on-line by typing:

$ HELP CRUCR <RETURN>

For documentation comments, a Reader's Response form is located at the back of each manual.

Introduction

This introductory section describes the DOMAIN system insert files and the format of the information found in the sections that follow. Each of these sections consist of a description of the data types used by a system manager, the syntax of the manager's programming calls, and the error messages generated by the system manager. We have arranged the sections of this manual **alphabetically**, by system manager name.

DOMAIN Insert Files

The DOMAIN system provides insert files that define data types, constants, values, and routine declarations. The insert files also define the exact form of each system call or routine. (Even the FORTRAN version does this using comments, although the FORTRAN compiler doesn't check the forms that you use.)

The DOMAIN system routines are divided, by function, into several subsystems. Each subsystem is controlled by a system manager. The routines of each subsystem are prefixed for easy indentification. A subsystem prefix consists of a number of identifying characters followed by the special underscore and dollar-sign characters, "_$." For example, the routines that perform stream functions are prefixed with STREAM_$. These subsystem prefixes are also used to distinguish DOMAIN data types and constants that are used by the subsystem routines.

Insert files are located in the directory /SYS/INS/. There is one insert file per subsystem for each programming language. Include the appropriate insert file for your programming language. For example, if you are using error routines in a Pascal program, you include the insert file, /SYS/INS/ERROR.INS.PAS. Using the same routines in a FORTRAN program, you include /SYS/INS/ERROR.INS.FTN. All insert files are specified using the syntax

/SYS/INS/subsystem-prefix.INS.language-abbreviation

where the language abbreviation is PAS (Pascal), FTN (FORTRAN), or C (C). The listing on the next page shows all the available insert files.

In addition to including required subsystem insert files in a program, you must always include the BASE insert file for your programming language. You specify BASE insert files using the syntax

/SYS/INS/BASE.INS.language-abbreviation

These files contain some basic definitions that a number of subsystem routines use.

Summary of Insert Files

Insert File	Operating System Component
/SYS/INS/BASE.INS.lan	Base definitions -- must always be included
/SYS/INS/ACLM.INS.lan	Access control list manager
/SYS/INS/CAL.INS.lan	Calendar
/SYS/INS/ERROR.INS.lan	Error reporting
/SYS/INS/EC2.INS.lan	Eventcount
/SYS/INS/GM.INS.lan	Graphics Metafile Resource
/SYS/INS/GMF.INS.lan	Graphics Map Files
/SYS/INS/GPR.INS.lan	Graphics Primitives
/SYS/INS/IPC.INS.lan	Interprocess communications datagrams
/SYS/INS/KBD.INS.lan	[Useful constants for keyboard keys]
/SYS/INS/MBX.INS.lan	Mailbox manager
/SYS/INS/MS.INS.lan	Mapping server
/SYS/INS/MTS.INS.lan	Magtape/streams interface
/SYS/INS/MUTEX.INS.lan	Mutual exclusion lock manager
/SYS/INS/NAME.INS.lan	Naming server
/SYS/INS/PAD.INS.lan	Display Manager
/SYS/INS/PBUFS.INS.lan	Paste buffer manager
/SYS/INS/PFM.INS.lan	Process fault manager
/SYS/INS/PGM.INS.lan	Program manager
/SYS/INS/PM.INS.lan	User process routines
/SYS/INS/PROC1.INS.PAS	Process manager (Pascal only)
/SYS/INS/PROC2.INS.lan	User process manager
/SYS/INS/RWS.INS.lan	Read/write storage manager
/SYS/INS/SIO.INS.lan	Serial I/O
/SYS/INS/SMDU.INS.lan	Display driver
/SYS/INS/STREAMS.INS.lan	Stream manager
/SYS/INS/TIME.INS.lan	Time
/SYS/INS/TONE.lan	Speaker
/SYS/INS/TPAD.INS.lan	Touchpad manager
/SYS/INS/VEC.INS.lan	Vector arithmetic
/SYS/INS/VFMT.INS.lan	Variable formatter

The suffix ".lan" varies with the high-level language that you're using; it is either ".FTN", ".PAS", or ".C".

Organizational Information

This introductory section is followed by sections for each subsystem. The material for each subsystem is organized into the following three parts:

1. Detailed data type information (including illustrations of records for the use of FORTRAN programmers).

2. Full descriptions of each system call. Each call within a subsystem is ordered alphabetically.

3. List of possible error messages.

Data Type Sections

A subsystem's data type section precedes the subsystem's individual call descriptions. Each data type section describes the predefined constants and data types for a subsystem. These descriptions include an atomic data type translation (i.e., TIME_$REL_ABS_T = 4-byte integer) for use by FORTRAN programmers, as well as a brief description of the type's purpose. Where applicable, any predefined values associated with the type are listed and described.

Following is an example of a data type description for the TIME_$REL_ABS_T type.

TIME_$REL_ABS_T	A 2-byte integer. Indicator of type of time. One of the following pre-defined values: TIME_$RELATIVE Relative time. TIME_$ABSOLUTE Absolute time.

In addition, the record data types are illustrated in detail. Primarily, we have geared these illustrations to FORTRAN programmers who need to construct record-like structures, but we've designed the illustrations to convey as much information as possible for all programmers. Each record type illustration:

- Clearly shows FORTRAN programmers the structure of the record that they must construct using standard FORTRAN data type statements. The illustrations show the size and type of each field.

- Describes the fields that make up the record.

- Lists the byte offsets for each field. These offsets are used to access fields individually.

- Indicates whether any fields of the record are, in turn, predefined records.

The following is the description and illustration of the CAL_$TIMEDATE_REC_T predefined record:

CAL_$TIMEDATE_REC_T	Readable time format. The diagram below illustrates the CAL_$TIMEDATE_REC_T data type:

predefined type	byte: offset		field name
	0:	integer	year
	2:	integer	month
	4:	integer	day
	6:	integer	hour
	8:	integer	minute
	10:	integer	second

Field Description:

year
Integer representing the year.

month
Integer representing the month.

day
Integer representing the day.

hour
Integer representing the hour (24 hr. format).

minute
Integer representing the minute.

second
Integer representing the second.

FORTRAN programmers, note that a Pascal variant record is a record structure that may be interpreted differently depending on usage. In the case of variant records, as many illustrations will appear as are necessary to show the number of interpretations.

System Call Descriptions

We have listed the system call descriptions alphabetically for quick reference. Each system call description contains:

- An abstract of the call's function.
- The order of call parameters.
- A brief description of each parameter.
- A description of the call's function and use.

These descriptions are standardized to make referencing the material as quick as possible.

Each parameter description begins with a phrase describing the parameter. If the parameter can be declared using a predefined data type, the descriptive phrase is followed by the phrase ",in XXX format" where XXX is the predefined data type. Pascal or C programmers, look for this phrase to determine how to declare a parameter.

FORTRAN programmers, use the second sentence of each parameter description for the same purpose. The second sentence describes the data type in atomic terms that you can use, such as "This is a 2-byte integer." In complex cases, FORTRAN programmers are referenced to the respective subsystem's data type section.

The rest of a parameter description describes the use of the parameter and the values it may hold.

The following is an example of a parameter description:

```
access
   New  access  mode,  in  MS_$ACC_MODE_T  format.  This  is a 2-byte integer.
   Specify only one of the following predefined values:

      MS_$R           Read access.

      MS_$WR          Read and write access.

      MS_$RIW         Read with intent to write.

  An object which is locked MS_$RIW may not be changed to MS_$R.
```

Error Sections

Each error section lists the status codes that may be returned by subsystem calls. The following information appears for each error:

- Predefined constant for the status code.
- Text associated with the error.

IOS

This section describes the data types, the call syntax, and the error codes for the IOS programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

IOS_$MAX	127	Highest possibe number in stream ID.
IOS_$NO_STREAM	16#7FFF	Placeholder for stream ID.

VARIABLES

XOID_$NIL	A variable whose value is the NIL XOID and doesn't change. Used for comparisons and assignments of XOID_$T variables.

DATA TYPES

IOS_$ABS_REL_T

A 2-byte integer. Specifies whether seek is relative or absolute. One of the following predefined values:

IOS_$RELATIVE
Seek from the current position.

IOS_$ABSOLUTE
Seek from the beginning of the object (BOF).

IOS_$CONN_FLAG_T

A 2-byte integer. Attributes associated with a stream connection. One of the following predefined values:

IOS_$CF_TTY
Connection behaves like a terminal.

IOS_$CF_IPC
Connection behaves like an interprocess communication (IPC) channel.

IOS_$CF_VT
Connection behaves like a DOMAIN Display Manager pad.

IOS_$CF_WRITE
Connection can be written to.

IOS_$CF_APPEND
Connection's stream marker can be positioned to the end of the object before each put call.

IOS_$CF_UNREGULATED
Other processes can read and write to the connection.

IOS_$CF_READ_INTEND_WRITE
Connection open for read access, and can later

	be open for write access. Other processes can have read access.
IOS_$CONN_FLAG_SET	A 4-byte integer. A set of connection attributes, in IOS_$CONN_FLAG_T format, indicating which attributes of the specified connection are set. For a list of options, see IOS_$CONN_FLAG_T above.
IOS_$CREATE_MODE_T	A 2-byte integer. Specifies the action to be taken if the name already exists or specifies creation of umnamed objects. One of the following predefined values: IOS_$LOC_NAME_ONLY_MODE Create a temporary unnamed object, uses pathname to specify location of object, and locates it on the same volume. IOS_$MAKE_BACKUP_MODE Create a backup (.bak) object when closed. IOS_$NO_PRE_EXIST_MODE Return an error if object already exists. IOS_$PRESERVE_MODE Save contents of object, if it exists, opens object, and positions stream marker at the beginning of the object (BOF). IOS_$RECREATE_MODE Delete existing object and creates new one of same name. IOS_$TRUNCATE_MODE Open object, then truncates the contents.
IOS_$DIR_TYPE_T	A 2-byte integer. Specifies type of directory. One of the following predefined values: IOS_$WDIR Current working directory. IOS_$NDIR Current naming directory
IOS_$EC_KEY_T	A 2-byte integer. Specifies eventcount key type. One of the following predefined values: IOS_$GET_EC_KEY Key that is advanced with each get call. IOS_$PUT_EC_KEY Key that is advanced with each put call.

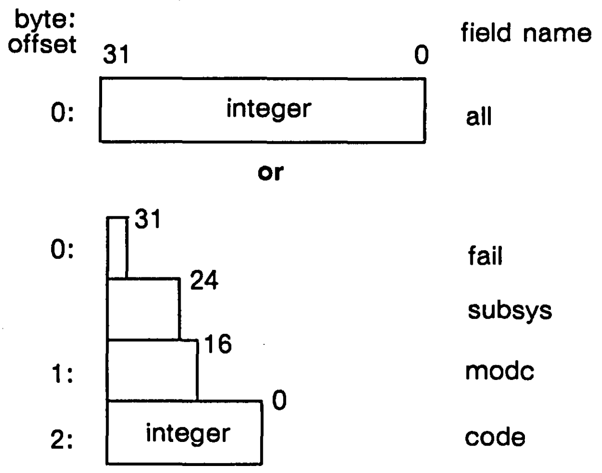
IOS_$ID_T	A 2-byte integer, ranging in value from 0 to IOS_$MAX. The stream ID.
IOS_$MGR_FLAG_T	A 2-byte integer. Object attributes associated with an object's manager. One of the following predefined values: IOS_$MF_CREATE Manager permits type to create objects. IOS_$MF_CREATE_BAK Manager permits type to create backup (.bak) objects. IOS_$MF_IMEX Manager permits type to export streams to new processes. IOS_$MF_FORK Manager permits type to pass streams to forked processes. IOS_$MF_FORCE_WRITE Manager permits type to force-write object contents to stable storage (for most object types, this is the disk). IOS_$MF_WRITE Manager permits objects to be written to. IOS_$MF_SEEK_ABS Manager permits objects to perform absolute seeks. IOS_$MF_SEEK_SHORT Manager permits objects to seek using short (4-byte) seek keys. IOS_$MF_SEEK_FULL Manager permits objects to seek using full (8-byte) seek keys. IOS_$MF_SEEK_BYTE Manager permits objects to seek to byte positions. IOS_$MF_SEEK_REC Manager permits objects to seek to record positions. IOS_$MF_SEEK_BOF Manager permits objects to seek to the beginning of the object. IOS_$MF_REC_TYPE

	Manager supports different record type formats. IOS_$MF_TRUNCATE Manager permits objects to be truncated. IOS_$MF_UNREGULATED Manager permits objects to have shared (unregulated) concurrency mode. IOS_$MF_SPARSE Manager permits objects to be as sparse. IOS_$MF_READ_INTEND_WRITE Manager permits objects to have read-intend-write access.
IOS_$MGR_FLAG_SET	A 4-byte integer. A set of object manager attributes, in IOS_$MGR_FLAG_T format, indicating which attributes of the specified object's manager are set. For a list of options, see IOS_$MGR_FLAG_T above.
IOS_$NAME_TYPE_T	A 2-byte integer. Specifies format of pathname. One of the following predefined values: IOS_$ROOT_NAME Absolute pathname relative to the network root directory (//); for example, //node/sid/file. IOS_$WDIR_NAME Leaf name if object's name is a name in current working directory; otherwise, specifies absolute pathname. IOS_$NDIR_NAME Leaf name if object's name is a name in current naming directory; otherwise, specifies absolute pathname. IOS_$NODE_NAME Name relative to the node's entry directory (/) if object is a name in boot volume; otherwise, specifies absolute pathname; for example, /sid/file. IOS_$NODE_DATA_FLAG Leaf name if object's name is a name in current 'node_data directory; otherwise, specifies absolute pathname. IOS_$LEAF_NAME Leaf name regardless of object's name.

	IOS_$RESID_NAME Residual name if object is defined using extended naming.
IOS_$OBJ_FLAG_T	A 2-byte integer. Attributes associated with an object. One of the following predefined values: IOS_$OF_DELETE_ON_CLOSE Object can be deleted when all its associated connections are closed. IOS_$OF_SPARSE_OK Object can be written as a sparse object. IOS_$OF_ASCII Object contains ASCII data. IOS_$OF_FTNCC Object uses FORTRAN carriage control characters. IOS_$OF_COND Object performs get or put calls conditionally, as if the IOS_$COND_OPT was specified.
IOS_$OBJ_FLAG_SET	A 4-byte integer. A set of object attributes, in IOS_$OBJ_FLAG_T format, indicating which attributes of the specified object are set. For a list of options, see IOS_$OBJ_FLAG_T above.
IOS_$OPEN_OPTIONS_T	A 2-byte integer. Specifies options for an IOS_$OPEN. Any combination of the following predefined values: IOS_$NO_OPEN_DELAY_OPT Return immediately instead of waiting for open to complete. IOS_$WRITE_OPT Permit writing data to a new object. IOS_$UNREGULATED_OPT Permit concurrency (unregulated read and write access.) to the object IOS_$POSITION_TO_EOF_OPT Position stream marker to the end of the object at open. IOS_$INQUIRE_ONLY_OPT Open object for attribute inquiries only. IOS_$READ_INTEND_WRITE_OPT Object has read-intend-write access, other processes can have read but not write access.

IOS_$POS_OPT_T	A 2-byte integer. Specify position to return when inquiring about object position. One of the following predefined values: IOS_$CURRENT Return key for the current stream marker. IOS_$BOF Return key for beginning of the object (BOF) marker. IOS_$EOF Return key for end of the object (EOF) marker.
IOS_$PUT_GET_OPTS_T	A 2-byte integer. Specifies options for put and get operations. Any combination of the following predefined values: IOS_$COND_OPT Read or write data conditionally. If call fails, returns IOS_$xxx_CONDITIONAL_FAILED, where xxx is either GET or PUT. IOS_$PREVIEW_OPT Write data but do not update the stream marker. IOS_$PARTIAL_RECORD_OPT Write a portion of a record but do not terminate it. IOS_$NO_REC_BNDRY_OPT Ignore record (line) boundries.
IOS_$RTYPE_T	A 2-byte integer. Specifies the record type format. One of the following predefined values: IOS_$V1 Variable-length record with count fields. IOS_$F2 Fixed-length records with count fields. IOS_$UNDEF No record structure. IOS_$EXPLICIT_F2 Fixed-length records that IOS_$PUT cannot implicitly change to IOS_$V1. IOS_$F1 Fixed-length records without count fields.

IOS_$SEEK_KEY_T	The full seek key. This is an 8-byte integer value.
IOS_$SEEK_TYPE_T	A 2-byte integer. Specifies the type of seek to perform. One of the following predefined values: IOS_$REC_SEEK Record-oriented seek. IOS_$BYTE_SEEK Byte-oriented seek.
STATUS_$T	A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

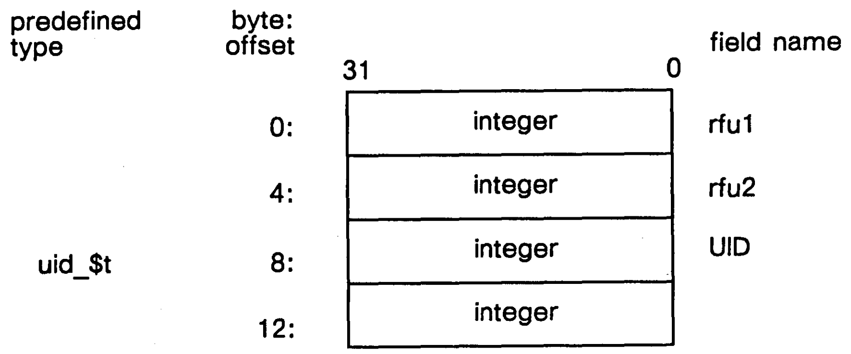
fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

UID_$T	An object type identifier. This is an 8-byte integer value.
XOID_$T	Unique identifier of an object. Used by type managers only. The diagram below illustrates the XOID_$T data type:



Field Description:

rfu1
Reserved for future use.

rfu2
Reserved for future use.

UID
Unique identifier for an object.

IOS_$CHANGE_PATH_NAME

Changes the pathname of an object.

FORMAT

```
IOS_$CHANGE_PATH_NAME (stream-id, new-pathname, new-namelength, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

new-pathname
New name of the object, in NAME_$PNAME_T format. This is an array of up to 256 characters.

new-namelength
Length of "new-pathname." This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$CHANGE_PATH_NAME changes the pathname of an existing object. The stream ID of the object remains the same.

IOS_$CHANGE_PATH_NAME permits you to assign a name to a previously unnamed object and, conversely, to remove a name from a previously named object. (To remove a name, specify a null pathname.)

Note that this call can change the delete-on-close object attribute. For example, if you assign a name to an unnamed object, the operation implicitly changes the delete-on-close attribute to FALSE. Likewise, if you specify a null pathname for a previously named object, the operation implicitly changes the delete-on-close attribute to TRUE. Be aware that this behavior can cause unexpected results in cases where you explicitly change the delete-on-close attribute, and then make an unnamed-to-named name change.

IOS_$CLOSE

Closes a stream.

FORMAT

```
IOS_$CLOSE (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of the stream to be closed, in IOS_$ID_T format. This is a 2-byte integer.

Once IOS_$CLOSE closes the stream, the number used for this stream ID becomes available for reuse. If the object is open on more than one stream, IOS_$CLOSE closes only "stream-id."

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$CLOSE closes the stream so that you can no longer use the stream ID to operate on the object. Closing a stream to an object releases any locks maintained by the stream connection, thus making the object available to other users.

A program can close only the streams that it has opened at the current or lower program levels (that is, streams opened by programs that the calling program has invoked). IOS_$CLOSE returns an error status code if you try to close a stream that was opened at a higher program level.

If an object has the delete-on-close attribute (IOS_$OF_DELETE_ON_CLOSE), IOS_$CLOSE deletes the object. However, the object is not deleted until *all* streams to it are closed. (For details on object attributes, see the IOS_$INQ_$OBJ_FLAGS and IOS_$SET_OBJ_FLAG calls.)

IOS_$CREATE

Creates an object and opens a stream to it.

FORMAT

```
IOS_$CREATE (pathname, namelength, type-uid, create-options,
          open-options, stream-id, status)
```

INPUT PARAMETERS

pathname
Name of the object to be created, in NAME_$PNAME_T format. This is an array of up to 256 characters. To create a temporary object, see the section "Creating an Object in Backup Mode" below.

namelength
Length of "pathname," in bytes. This is a 2-byte integer. To create a temporary object, see the section "Creating an Object in Backup Mode" below.

type-uid
UID of the type to be created, in UID_$T format. This data type is 8 bytes long. See the IOS Data Types section for more information.

If you specify the predefined UID_$NIL, IOS_$CREATE creates an object of the default type, which is currently unstructured ASCII (UASC). You can also specify any of the system's predefined type UIDs listed below, or any valid user-created type UID.

DOMAIN currently supports a set of standard object types which include the following types. (Note that objects created by type managers return manager-specific type UIDs.)

Type UID	Object
UASC_$UID	UASC object
RECORDS_$UID	Record-oriented object
HDR_UNDEF_$UID	Nonrecord-oriented object
OBJECT_FILE_$UID	Object module object (compiler or binder output)
SIO_$UID	Serial line descriptor object
MT_$UID	Magnetic tape descriptor object
PAD_$UID	Saved Display Manager transcript pad
INPUT_PAD_$UID	Display Manager input pad
MBX_$UID	Mailbox object

Type UID	**Object**
DIRECTORY_$UID	Directory
NULLDEV_$UID	Null device

create-options

Specifies the action to be taken if the object already exists, or specifies the creation of an unnamed object, in IOS_$CREATE_MODE_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$NO_PRE_EXIST_MODE	Return the IOS_$ALREADY_EXISTS error status code if an object with the specified name already exists.
IOS_$PRESERVE_MODE	Preserve the contents of the object if an object with the specified name already exists. Then open the object and position the stream marker to the beginning of the object (BOF) unless you set the IOS_$POSITION_TO_EOF open option. Use this mode to append data to an existing object.
IOS_$RECREATE_MODE	Recreate the object if an object with the specified name already exists. Essentially, this option deletes the existing object and creates a new one. The new object will have the default set of attributes for that object type.
IOS_$TRUNCATE_MODE	Open the object and delete the contents if an object with the specified name already exists. Use this mode to create an object to preserve the attributes of the specified object.
IOS_$MAKE_BACKUP_MODE	Create a temporary object with the same type and attributes as the object specified in the pathname if an object with the specified name already exists. Use this mode to create a backup object. (See below for detailed description.)
IOS_$LOC_NAME_ONLY_MODE	Create a temporary unnamed object. Use the pathname to specify the location of the object. IOS_$CREATE will locate the temporary object on the same volume as the object specified in the pathname.

open-options

Open options, in IOS_$OPEN_OPTIONS_T format. This is a 2-byte integer. Specify a combination of the following set of predefined values:

IOS_$NO_OPEN_DELAY_OPT	Return immediately, instead of waiting for the open call to complete.
IOS_$WRITE_OPT	Permit writing data to a new object. If a program tries to write on a stream for which you have not specified this option, it returns an error status. Note that when creating an object, the IOS manager automatically sets this value because it assumes that when you create an object, you will want to write to it.
IOS_$UNREGULATED_OPT	Permit shared (unregulated) concurrency mode.
IOS_$POSITION_TO_EOF_OPT	Position the stream marker at the end of the object (EOF). Use this to append data to an existing object.
IOS_$INQUIRE_ONLY_OPT	Open the object for attribute inquiries only; do not permit reading or writing of data.
IOS_$READ_INTEND_WRITE_OPT	Open the object for read access with the intent to eventually change the object's access to write access. This allows other processes to read the object; but they cannot have write or read-intend-write access.

OUTPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

Subsequent IOS calls use this number to identify the stream opened by this call.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

If the pathname specifies an object that does not exist, IOS_$CREATE creates a new object of the specified type using that pathname and opens a stream to it. If the object already exists, the create mode option specified in the call determines which action IOS_$CREATE will perform.

Both IOS_$CREATE and IOS_$OPEN open a stream to an object. However, IOS_$CREATE creates the object if it does not exist, whereas IOS_$OPEN returns an error if the object does not exist.

Inquiring about Object Attributes

When IOS_$CREATE creates an object, the object has a default set of attributes (the default attributes depend on the type created). These attributes fall into three categories: manager, object, and connection attributes. To determine which attributes the newly created object has, you can use the following calls:

IOS_$INQ_MGR_FLAGS	Returns the attributes that the object's type manager defines.
IOS_$INQ_OBJ_FLAGS	Returns the attributes of the object.
IOS_$INQ_CONN_FLAGS	Returns the attributes of the stream connection.

To change object or connection attributes, use the IOS_$SET_OBJ_FLAGS, and IOS_$SET_CONN_FLAGS calls, respectively. The attributes that you can change depend on the object type. Note that you cannot change manager attributes because the type manager determines them. For details on writing a type manager, see the *Extending the DOMAIN Streams Facility* manual.

Creating a Temporary Object

IOS_$CREATE allows you to create a temporary object two ways. To create a temporary object on your boot volume, specify a null value in "pathname" and a value of 0 in "namelength." To create a temporary object on another volume, specify the pathname of an existing object on that volume with the IOS_$LOC_NAME_ONLY_MODE option in "create-options." IOS_$CREATE creates a temporary unnamed object on the same volume (node) as the object you specify in "pathname."

Creating an Object in Backup Mode

You can create a new, *unnamed* temporary object by specifying the create mode option, IOS_$MAKE_BACKUP_MODE. The call creates the new object with the same type and attributes as the object specified by "pathname" (if it exists), and it is created on the same volume (node).

IOS_$CREATE does not open or modify the object specified by "pathname," it merely examines the object to extract its attributes. Even though IOS_$CREATE does not modify the "pathname," it conceptually replaces the object, so this operation requires write access to the object.

When IOS_$CLOSE closes the stream created with this call, it changes the object specified by "pathname" to "pathname.bak." It changes the new object (formerly the temporary, unamed object) to "pathname," and makes the object permanent.

If a ".bak" version of the object already exists, IOS_$CLOSE deletes it. (The caller must have either D or P rights to delete the object.) If the ".bak" object is locked at the time IOS_$CLOSE is called, the object will be deleted when it is unlocked.

If "pathname" does not exist at the time that IOS_$CREATE is called, then IOS_$CREATE performs the ordinary functions.

IOS_$DELETE

Deletes an object and closes the associated stream.

FORMAT

```
IOS_$DELETE (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of a stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$DELETE deletes the object, then closes the specified stream.

This call actually sets the object attribute IOS_$OF_DELETE_ON_CLOSE to TRUE, then closes the stream. So, if the type manager does not allow an object to set the delete-on-close attribute, the delete call fails. In this case, the call closes the stream, but does not delete the object.

If the object is open on more than one stream, IOS_$DELETE marks the object for deletion, but the object still exists until *all* streams to that object are closed.

IOS_$DUP

Creates a copy of a specified stream ID.

FORMAT

```
return_stream_id = IOS_$DUP (stream_id_to_duplicate, copy_stream_id, status)
```

RETURN VALUE

return_stream_id
Number of the new stream created, in IOS_$ID_T format. This is a 2-byte integer.

INPUT PARAMETERS

stream_id_to_duplicate
Number of the stream to duplicate, in IOS_$ID_T format. This is a 2-byte integer. This stream number remains a valid connection to the object after IOS_$DUP completes successfully.

copy_stream_id
Number of the stream to use as the newly created copy, in IOS_$ID_T format. This is a 2-byte integer.

If "copy_stream_id" is free, IOS_$DUP returns that value in "return_stream_id." If "copy_stream_id" is in use, IOS_$DUP begins searching from that number upward (higher numbers) until it finds a free stream number and returns that number in "return_stream_id."

If the actual number of "copy_stream_id" is insignificant, specify the value 0. This value causes IOS_$DUP to begin searching from the lowest possible stream number and return the first free stream number it finds.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$DUP to create a copy of an existing stream ID. The new stream ID refers to the same connection as the existing stream ID. Note that you must close *both* streams with IOS_$CLOSE before the stream connection actually closes.

You can use IOS_$DUP to keep a stream connection open when passing it to a subroutine. Use IOS_$DUP to create a copy of the stream ID before passing it. This way, the subroutine cannot close the connection to the object because *all* copies of the stream connection must be closed before the connection itself closes.

IOS_$DUP is identical to IOS_$REPLICATE except that IOS_$DUP looks for a free stream number in *ascending* order from the specified stream ID, while IOS_$REPLICATE looks in *descending* order. Note that you use IOS_$DUP or IOS_$REPLICATE to *copy* existing stream ID's, both the existing and new stream ID's remain valid connections. However, you use IOS_$SWITCH to *replace* stream IDs; you "switch" the connection from the existing stream ID to the new stream ID.

IOS_$EQUAL

Determines whether two stream IDs refer to the same object.

FORMAT

```
same IOS_$EQUAL (stream_id, stream_id_too, status)
```

RETURN VALUE

same
Boolean value that indicates whether the specified stream IDs refer to the same object. "Same" is TRUE if the streams refer to the same object, it is FALSE if they do not.

INPUT PARAMETERS

stream_id
Number of a stream being compared, in IOS_$ID_T format. This is a 2-byte integer.

stream_id_too
Number of a stream being compared, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$EQUAL to determine whether two stream IDs refer to the same object. An application program can use this call to avoid using two streams when one is sufficient.

IOS_$FORCE_WRITE_FILE

Forcibly writes an object to permanent storage.

FORMAT

```
IOS_$FORCE_WRITE_FILE (stream-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$FORCE_WRITE_FILE forcibly writes the object to stable storage. Stable storage depends on the object's type, however, in most cases, it is the disk. For example, stable storage for a magnetic tape descriptor is the tape.

Use IOS_$FORCE_WRITE_FILE before closing the stream to ensure that the object is stored safely in the event of a system crash.

IOS_$GET

Moves data from a stream into a buffer.

FORMAT

```
ret-length = IOS_$GET (stream-id, get-options, buffer, buffer-size, status)
```

RETURN VALUE

ret-length

Amount of data moved, in bytes. This is a 4-byte integer.

"Ret-length" equals the amount of data read; "ret-length" equals 0 if IOS_$GET does not return any data.

If the length of the data read exceeds the amount specified in "data-size," IOS_$GET performs the following:

- Reads enough data to fill the requested size
- Sets "ret-length" equal to "data-size"
- Positions the stream marker to the first unread byte
- Returns the IOS_$BUFFER_TOO_SMALL status code to indicate that this condition has occurred

You can inquire about how many bytes remain to be read in the current record by calling IOS_$INQ_REC_REMAINDER.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

get-options

Options that control how IOS_$GET performs the get operation, in IOS_$PUT_GET_OPTS_T format. This is a 2-byte integer. Specify a combination of the following set of predefined values:

IOS_$COND_OPT	Reads data, if available. (For example, data on an SIO line is not always available immediately.) If the data is not available, IOS_$GET returns the IOS_$GET_CONDITIONAL_FAILED status code and sets the return value of "ret-length" to 0.
IOS_$PREVIEW_OPT	Reads data but does not update the stream marker.

IOS_$NO_REC_BNDRY_OPT	Ignores record boundaries while reading data. For example, it ignores NEWLINE characters in a UASC object, which guarantees that the call fills the specified buffer. Some type managers might not support this option.
IOS_$PARTIAL_RECORD_OPT	Not meaningful for this call.

buffer-size
Maximum number of bytes to be moved to the buffer. This is a 4-byte integer.

OUTPUT PARAMETERS

buffer
Buffer to store the data. This is a character array.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

You can use either of IOS_$LOCATE or IOS_$GET to read data from system objects. IOS_$GET copies the data into a buffer, while IOS_$LOCATE returns the virtual address of the data.

In most cases, use the IOS_$LOCATE call to read data because it is faster (IOS_$LOCATE does not perform a copy).

You will want to use IOS_$GET when you need to read more data than can be obtained in one call, because the pointer remains valid for only one call. For example, use IOS_$GET when you need to read and rearrange a number of lines from an object.

IOS_$GET_DIR

Gets the current working or naming directory.

FORMAT

```
IOS_$GET_DIR (pathname, namelength, dir_type, status)
```

INPUT PARAMETERS

dir_type
Option specifying which type of directory to get, in IOS_$DIR_TYPE_T format. Specify one of the predefined values:

IOS_$WDIR	Name of the current working directory.
IOS_$NDIR	Name of the current naming directory.

OUTPUT PARAMETERS

pathname
Name of the directory to get, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelength
Length of "pathname." This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to get the current working or naming directory. It returns the name of the directory in the "pathname" parameter. If you want to change the current working or naming directory, use IOS_$SET_DIR.

IOS_$GET_EC

Gets a pointer to an eventcount for a stream event.

FORMAT

```
IOS_$GET_EC (stream-id, stream-key, eventcount-pointer, status)
```

INPUT PARAMETERS

stream-id
Number of stream on which the eventcount is waiting, in IOS_$ID_T format. This is a 2-byte integer.

stream-key
The key that specifies which type of eventcount to get a pointer to, in IOS_$EC_KEY_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$GET_REC_EC_KEY	An eventcount that advances when the stream contains data for you to get. This eventcount advances whenever there is anything to get from an open stream.
IOS_$PUT_REC_EC_KEY	An eventcount that advances when a previously "full" stream might now be able to accept data. A full stream is a stream that IOS_$PUT will block.

OUTPUT PARAMETERS

eventcount-pointer
A pointer to the eventcount, in EC2_$PTR_T format. This is a 4-byte integer address that points to an array of eventcounts. See the EC2 Data Types section for more information.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$GET_EC is valid for all streams, including those open to objects, pads, mailboxes, and devices. After you use this call to get a stream event, use EC2 calls to read eventcount values and wait for events.

You can wait for two types of events on a stream:

- The IOS-get eventcount indicates that there might be input to get from an open stream.

- The IOS-put eventcount indicates that a previously "full," or blocked, stream might now have enough room to accept the data.

An example of using the IOS-get eventcount is to wait for keyboard input. Whenever the user types data, the system advances the eventcount associated with the user's input pad. If input pad is in normal (or cooked) mode, the eventcount advances after each carriage return, if the input pad is in raw mode, the eventcount advances after each keystroke. (For details on cooked and raw mode, see the Display Manager chapter in the *Programming with General System Calls* manual.)

An example of using the IOS-put eventcount is to wait on an MBX channel that might get blocked. That is, IOS_$PUT blocks streams associated with MBX channels if a server is not ready for the data from the channel. When it's possible to write data without blocking, the system advances the IOS-put eventcount.

For more information on eventcounts, see the *Programming with General System Calls* and the *Programming with System Calls for Interprocess Communication* manuals.

IOS_$GET_HANDLE

Converts a stream ID to a handle pointer.

FORMAT

```
handle = IOS_$GET_HANDLE (stream-id, type-uid, status)
```

RETURN VALUE

handle

Pointer to the handle associated with the stream connection, in UNIV_PTR format. This is a 4-byte integer.

INPUT PARAMETERS

stream-id

Number of the stream that identifies an open stream, in IOS_$ID_T format. This is a 2-byte integer.

type-uid

Type UID of the object that the type manager handles, in UID_$T format. Specify the type UID of the manager you are writing. This data type is 8 bytes long. See the IOS Data Types section for more information.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

NOTE: This call is generally of interest to type manager writers only.

Type manager writers use this call to access an object when implementing an operation that is not predefined by the system. When the type manager implements such an operation, it is referred to as a **direct manager call** because the I/O switch does not route the call between the client call and the manager. Without switching, the manager receives a stream ID from a client. To access the object, the manager must then call IOS_$GET_HANDLE to obtain the object handle associated with the stream ID.

IOS_$GET_HANDLE returns an error if the stream ID is not associated with an object of the type UID specified by "type_uid." Specify the type UID of the manager you are writing so that the manager can be sure it has a stream to an object of its type.

See the *Using the Open System Toolkit to Extend the Streams Facility* manual for more information.

IOS_$INQ_BYTE_POS

Returns the byte position of the stream marker.

FORMAT

```
byte-position = IOS_$INQ_BYTE_POS (stream-id, position-option, status)
```

RETURN VALUE

byte-position
Byte position of the stream marker. This is a 4-byte integer. Note that byte positions are *zero*-based; consequently the byte position of the beginning of an object (BOF) is 0.

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

position-option
Value specifying the byte position to return, in IOS_$POS_OPT_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$CURRENT	Returns the byte position of the current stream marker.
IOS_$EOF	Returns the byte position of the stream marker at the end of the object (EOF). This is the number of bytes in the object.
IOS_$BOF	Return the byte position of the stream marker at the beginning of the object (BOF). This value is always 0.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

To obtain the offset of the stream marker, use IOS_$INQ_BYTE_POS. (Use IOS_$INQ_REC_POS if your object is record-oriented.)

To get the offset of the stream marker at the beginning or end of the object, specify IOS_$BOF or IOS_$EOF, in the "position-option" parameter. Specify IOS_$CURRENT to get the offset of the stream marker from the beginning of the object. Once you have the returned offset, you can move the stream marker to desired location by calling IOS_$SEEK.

This call allows you to perform a nonkeyed seek by specifying an absolute byte position, or by getting an offset from an absolute position, and moving the stream marker to it.

Whether you perform a nonkeyed or keyed seek depends on how the object's data is represented. For example, programs that need to perform "arithmetic" on the data (such as comparing two positions) will use nonkeyed seek operations. Programs that require only the ability to move from one position to another in an object will use keyed seek operations.

IOS_$INQ_CONN_FLAGS

Returns the attributes associated with a connection.

FORMAT

```
conn_flags = IOS_$INQ_CONN_FLAGS (stream-id, status)
```

RETURN VALUE

conn_flags

A set (bit mask) indicating which attributes of the specified connection are set, in IOS_$CONN_FLAG_SET format. This is a 4-byte integer. Any combination of the following set of predefined values, in IOS_$CONN_FLAG_T format, can be returned. If the set contains the value, the connection has the attribute.

IOS_$CF_TTY	Connection behaves like a terminal.
IOS_$CF_IPC	Connection behaves like an interprocess communication (IPC) channel.
IOS_$CF_VT	Connection behaves like a DOMAIN Display Manager pad.
IOS_$CF_WRITE	Connection can be written to.
IOS_$CF_APPEND	Connection's stream marker will be positioned at the end of the object (EOF) before each put call.
IOS_$CF_UNREGULATED	Connection is open for unregulated (shared) concurrency mode.
IOS_$CF_READ_INTEND_WRITE	Connection is open for read access, and can be changed to write access. Other connections can have read access, but not write or read-intend-write access.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to determine which *connection* attributes are in effect for the specified stream.

To change object or connection attributes, use the IOS_$SET_OBJ_FLAGS, and IOS_$SET_CONN_FLAGS calls respectively. Which attributes you can change depends on the object type.

IOS_$INQ_CUR_REC_LEN

Returns the length of the record at the current stream marker.

FORMAT

```
rec-length = IOS_$INQ_CUR_REC_LEN (stream-id, status)
```

RETURN VALUE

rec-length
Length of the current record. This is a 4-byte integer.

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$INQ_CUR_REC_LEN to determine the length of the record at the current stream marker of the specified stream.

The object specified must be record-oriented (for example, RECORDS_$UID); otherwise, IOS_$INQ_CUR_REC_LEN returns an error.

IOS_$INQ_FILE_ATTR

Returns object usage attributes including date and time created, date and time last used, date and time last modified, number of blocks in the object.

FORMAT

```
IOS_$INQ_FILE_ATTR (stream_id, dt-created, dt-modified, dt-used, blocks, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

dt-created
Date and time the object was created, in TIME_$CLOCKH_T format. This is a 4-byte integer.

dt-modified
Date and time the object was last modified, in TIME_$CLOCKH_T format. This is a 4-byte integer.

dt-used
Date and time the object was last used, in TIME_$CLOCKH_T format. This is a 4-byte integer.

blocks
The number of 1024-byte blocks that the object occupies. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$INQ_FILE_ATTR to obtain a time stamp for an object and to determine the amount of space that an object occupies.

IOS_$INQ_FULL_KEY

Returns a full seek key.

FORMAT

```
IOS_$INQ_FULL_KEY (stream-id, position-option, full-key, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

position-option
Value specifying the position to return a full seek key for, in IOS_$POS_OPT_T format. This is a 2-byte integer. Specify only one of the following predefined values:

IOS_$CURRENT	Return the full seek key of the current marker.
IOS_$EOF	Return the full seek key of the end of the object (EOF) marker.
IOS_$BOF	Return the full seek key of the beginning of the object (BOF) marker.

OUTPUT PARAMETERS

full-key
Full seek key to be used in subsequent seeks, in IOS_$SEEK_KEY_T format. This data type is 8 bytes long. See the IOS Data Types section for more information.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$INQ_FULL_KEY returns a seek key based on the position option you specify, the current stream marker position, beginning or end of the object.

Use seek keys to perform random access of data. Typically, you use this call to inquire about a seek key before writing some data, and then store the seek key. To access the data at a later point in time, position the stream marker by calling the IOS_$SEEK_FULL_KEY call with the stored seek key, and get the data with an IOS get operation (IOS_$GET or IOS_$LOCATE).

Use seek keys merely as an index -- do not rely on the *contents* of the keys. The contents of seek keys remain private to the IOS manager, which guarantees only that the seek key returns to the position it describes.

Some object types support seek key positioning, but do not support record or byte positioning. Use seek keys for repositioning if your application does not need the "arithmetic" properties of record- or byte-positioning (that is, the ability to compute positions given positions).

The DOMAIN system offers both short (4-byte) and full (8-byte) seek keys. Because short seek keys require half the storage space of full seek keys, you might want to use short seek keys if your application program stores a large number of seek keys. However, short seek keys are limiting in that you can only indicate *record* boundary positions, while full seek keys allow you to indicate *any* position.

IOS_$INQ_MGR_FLAGS

Returns the attribute set of an object's manager.

FORMAT

```
mgr_flags = IOS_$INQ_MGR_FLAGS (stream-id, status)
```

RETURN VALUE

mgr_flags

A set (bit mask) indicating the attributes of the specified object's manager, in IOS_$MGR_FLAG_SET format. This is a 4-byte integer. Any combination of the following set of predefined values, in IOS_$MGR_FLAG_T format, can be returned. If the set contains the value, the manager has the attribute and can perform the following operations:

IOS_$MF_CREATE	Manager permits type to create objects.
IOS_$MF_CREATE_BAK	Manager permits type to create backup (.bak) objects.
IOS_$MF_IMEX	Manager permits type to export streams to new processes.
IOS_$MF_FORK	Manager permits type to pass streams to forked processes.
IOS_$MF_FORCE_WRITE	Manager permits type to force-write object contents to stable storage (for most types, this is the disk).
IOS_$MF_WRITE	Manager permits objects to be written to.
IOS_$MF_SEEK_ABS	Manager permits objects to perform absolute seeks.
IOS_$MF_SEEK_SHORT	Manager permits objects to perform seeks using short (4-byte) seek keys.
IOS_$MF_SEEK_FULL	Manager permits objects to perform seeks using full (8-byte) seek keys.
IOS_$MF_SEEK_BYTE	Manager permits objects to perform seeks to byte positions.
IOS_$MF_SEEK_REC	Manager permits objects to perform seeks to record positions.
IOS_$MF_SEEK_BOF	Manager permits objects to perform seeks to the beginning of the object.
IOS_$MF_REC_TYPE	Manager supports different record type formats.

IOS_$MF_TRUNCATE	Manager permits objects to be truncated.
IOS_$MF_UNREGULATED	Manager permits objects to have unregulated (shared) concurrency mode.
IOS_$MF_SPARSE	Manager permits objects to be written as sparse objects.
IOS_$MF_READ_INTEND_WRITE	Manager permits objects to have read-intend-write access.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$INQ_MGR_FLAGS to determine what operations an *object's type manager* can perform.

Depending on the nature of the object, a type manager permits some of the operations identified by "mgr-flags." A manager usually will not support operations that are irrelevant for the object type. For example, if you called IOS_$INQ_MGR_FLAGS specifying a stream open on an SIO line, the set returned would not include any IOS_$MF_SEEK attributes, since serial lines do not support seeking.

Note that even if an object's manager permits an operation, the object itself can prevent the operation because the object's *object* and *connection* attributes must permit the operation as well. For example, a manager's attribute set might contain the attribute that permits writing to a file (IOS_$MF_WRITE), but a specific object's connection attribute set might not include the IOS_$CF_WRITE attribute, which permits writing on the connection. In this case, you *cannot* write to that particular object. However, you could possibly write to another object of the same type if the object's IOS_$CF_WRITE attribute is set for its stream connection.

To change object or connection attributes, use the IOS_$SET_OBJ_FLAGS and IOS_$SET_CONN_FLAGS calls, respectively. Which attributes you can change depends on the object type. Note that you cannot change manager attributes because the type manager determines them. For details on writing a type manager, see the *Extending the DOMAIN Streams Facility* manual.

IOS_$INQ_OBJ_FLAGS

Returns the attribute set associated with an object.

FORMAT

```
obj-flags = IOS_$INQ_OBJ_FLAGS (stream-id, status)
```

RETURN VALUE

obj-flags

A set (bit mask) indicating the attributes of the specified object, in IOS_$OBJ_FLAG_SET format. This is a 4-byte integer. Any combination of the following set of predefined values, in IOS_$OBJ_FLAG_T format, can be returned. If the set contains the value, the object has the attribute and can perform the following operations:

IOS_$OF_DELETE_ON_CLOSE	Object will be deleted when all its associated streams close.
IOS_$OF_SPARSE_OK	Object can be written as a sparse object.
IOS_$OF_ASCII	Object contains ASCII data.
IOS_$OF_FTNCC	Object uses FORTRAN carriage control characters.
IOS_$OF_COND	Get or put calls to the object will be performed conditionally, as if the IOS_$COND_OPT was specified on a get or put call.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to determine which *object* attributes are in effect for the object on the specified stream.

To change object or connection attributes, use the IOS_$SET_OBJ_FLAGS, and IOS_$SET_CONN_FLAGS calls respectively. The attributes that you can change depends on the object type.

IOS_$INQ_PATH_NAME

Returns the pathname of the object open on a specified stream.

FORMAT

```
IOS_$INQ_PATH_NAME (stream-id, name-type, pathname, namelength, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

name-type
Format of the returned pathname, in IOS_$NAME_TYPE_T format. Specify one of the following predefined values:

IOS_$ROOT_NAME
Return the absolute pathname, relative to the network root directory (//). For example, "//node/sid/file."

IOS_$WDIR_NAME
Return just the leaf name if the object's pathname is a name in the current working directory. Otherwise, return the absolute pathname.

IOS_$NDIR_NAME
Return just the leaf name if the object's pathname is a name in the current naming directory. Otherwise, return the absolute pathname.

IOS_$NODE_NAME
Return a name relative to the node's entry directory (/) if the object's pathname is a name in the boot volume. Otherwise, return the absolute pathname. For example, "/sid/file."

IOS_$NODE_DATA_FLAG
Return just the leaf name if the object's pathname is a name in the 'node_data directory. Otherwise, return the absolute pathname.

IOS_$LEAF_NAME
Return just the leaf name regardless of the object's pathname. For example, if the object's pathname is "/a/b/c," it returns "c."

IOS_$RESID_NAME
Return the residual part of a pathname if the stream is open using extended naming. (Extended naming allows you to add additional text to the end of a pathname.)

OUTPUT PARAMETERS

pathname
Name of the object associated with the stream ID, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelength

Length of the pathname. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to determine the pathname of an object associated with the specified stream ID. Generally, use this call in cases where a program has been passed a stream ID and needs the associated pathname.

IOS_$INQ_REC_POS

Returns the record position of the stream marker.

FORMAT

```
record_position = IOS_$INQ_REC_POS (stream-id, position-option, status)
```

RETURN VALUE

record-position

Record position of the stream marker. This is a 4-byte integer. Note that record positions are *zero*-based; consequently, the record position of the beginning of the object is 0.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

position-option

Value specifying the record position to return, in IOS_$POS_OPT_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$CURRENT	Return the record position of the current stream marker.
IOS_$EOF	Return the record position of the end of the object (EOF) stream marker. This is the number of records in the object.
IOS_$BOF	Return the record position of the beginning of the object (BOF) stream marker. This value is always 0.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

To obtain the offset of the stream marker for record-oriented objects, use IOS_$INQ_REC_POS. (UseIOS_$INQ_BYTE_POS if your object is *not* record-oriented.)

To get the offset of the stream marker at the beginning or end of the object, specify IOS_$BOF or IOS_$EOF, in the "position-option" parameter. Specify IOS_$CURRENT to get the offset of the stream marker from the beginning of the object. Once you have the returned offset, you can move the stream marker to desired location by calling IOS_$SEEK.

This call allows you to perform a nonkeyed seek by specifying an absolute byte position, or by getting an offset from an absolute position, and moving the stream marker to it.

Whether you perform a nonkeyed or keyed seek depends on how the object's data is represented. For example, programs that need to perform "arithmetic" on the data (such as comparing two positions) will use nonkeyed seek operations. Programs that require only the ability to move from one position to another in an object will use keyed seek operations.

IOS_$INQ_REC_REMAINDER

Returns the number of bytes remaining in the current record.

FORMAT

```
bytes = IOS_$INQ_REC_REMAINDER (stream-id, status)
```

RETURN VALUE

bytes
Number of bytes remaining in the current record. This is a 4-byte integer.

INPUT PARAMETERS

stream-id
Number of the stream on which the file is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$INQ_REC_REMAINDER with the IOS_$GET or IOS_$LOCATE calls. If IOS_$GET or IOS_$LOCATE fills the specified buffer, but has not yet finished reading a record, it returns the IOS_$BUFFER_TOO_SMALL error status code. At this point, use IOS_$INQ_REC_REMAINDER to determine the number of bytes in the record that remain to be read. If the entire record has been read, the value of "bytes" is undefined.

IOS_$INQ_REC_TYPE

Returns the record type of an object.

FORMAT

```
record-type = IOS_$INQ_REC_TYPE (stream-id, status)
```

RETURN VALUE

record-type

Type of record format used in the specified object, in IOS_$RTYPE_T format. This is a 2-byte integer. Returns one of the following predefined values:

IOS_$V1	Variable-length records with count fields.
IOS_$F1	Fixed-length records without count fields.
IOS_$F2	Fixed-length records with count fields.
IOS_$EXPLICIT_F2	Fixed-length records that IOS_$PUT *cannot* implicitly change to variable-length records.
IOS_$UNDEF	No record structure.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$INQ_REC_TYPE to determine how records within an object are formatted. You can change the record type of a record-oriented object by calling IOS_$SET_REC_TYPE.

By default, a record-oriented object has fixed-length records (IOS_$F2). They remain fixed-length records until IOS_$PUT writes records of different lengths. At this point, IOS_$PUT implicitly changes the objects to variable-length type (IOS_$V1). In some cases, you might want to explicitly set the record type to IOS_$EXPLICIT_F2 so that an attempt to write a variable-length record results in an error. To do so, use the corresponding call, IOS_$SET_REC_TYPE.

IOS_$INQ_SHORT_KEY

Returns a short seek key.

FORMAT

```
short-key = IOS_$INQ_SHORT_SEEK (stream-id, position-option, status)
```

RETURN VALUE

short-key
Short seek key to be used in subsequent seeks. This is a 4-byte integer.

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

position-option
Value specifying the position to return, in IOS_$POS_OPT_T format. This is a 2-byte integer. Specify only one of the following predefined values:

IOS_$CURRENT	Return the short seek key of the current marker.
IOS_$EOF	Return the short seek key of the end of the object (EOF) marker.
IOS_$BOF	Return the short seek key of the beginning of the object (BOF) marker.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$INQ_SHORT_KEY returns a seek key based on the position option you specify -- the current stream marker position, beginning or end of the object.

You use seek keys to perform random access of data. Typically, you use this call to inquire about a seek key before writing some data, and then store the seek key. To access the data at a later time, position the stream marker by calling the IOS_$SEEK_SHORT_KEY call with the stored seek key, and get the data with an IOS get operation (IOS_$GET or IOS_$LOCATE).

Use seek keys merely as an index -- do not count on the *contents* of the keys. The contents of seek keys remain private to the IOS manager, which guarantees only that the seek key returns to the position it describes.

Some object types support seek key positioning, but not record or byte positioning. Use seek keys for repositioning if your application does not need the "arithmetic" properties of record- or byte-positioning (that is, the ability to compute positions given positions).

The DOMAIN system offers both short (4-byte) and full (8-byte) seek keys. Because short seek keys require half the storage space of full seek keys, you might want to use short seek keys if your application program stores a large number of seek keys. However, short seek keys are limiting in that you can only indicate *record* boundary positions, while full seek keys allow you to indicate *any* position.

IOS_$INQ_TYPE_UID

Returns the type UID of an object.

FORMAT

```
IOS_$INQ_TYPE_UID (stream-id, type-uid, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

type-uid
Type UID of the object, in UID_$T format. This data type is 8 bytes long. See the IOS Data Types section for more information.

DOMAIN currently supports a set of predefined standard object types which include the following types. (Note that users can also define their own type UIDs by writing a type manager. See the *Using the Open System Toolkit to Extend the Streams Facility* manual for details.)

Type UID	**Object**
UASC_$UID	UASC object
RECORDS_$UID	Record-oriented object
HDR_UNDEF_$UID	Nonrecord-oriented object
OBJECT_FILE_$UID	Object module object (compiler or binder output)
SIO_$UID	Serial line descriptor object
MT_$UID	Magnetic tape descriptor object
PAD_$UID	Saved display manager transcript pad
INPUT_PAD_$UID	Display manager input pad
MBX_$UID	Mailbox object
DIRECTORY_$UID	Directory
NULLDEV_$UID	Null device

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to determine the object's current type UID given its stream ID. You can use the type UID returned by this call as a parameter in the IOS_$CREATE call to create another object of the same type.

IOS_$LOCATE

Reads data from a stream, and returns a pointer to the data.

FORMAT

```
ret-length = IOS_$LOCATE (stream-id, get-options, data-ptr, data-size,
                          status)
```

RETURN VALUE

ret-length

Amount of data read, in bytes. This is a 4-byte integer.

"Ret-length" equals the amount of data read; "ret-length" equals 0 if IOS_$LOCATE does not return any data.

If the length of the data read exceeds the amount specified in "data-size," IOS_$LOCATE performs the following:

- Reads enough data to fill the requested size
- Sets "ret-length" equal to "data-size"
- Positions the stream marker to the first unread byte
- Returns the IOS_$BUFFER_TOO_SMALL status code to indicate that this condition has occurred

You can inquire about how many bytes remain to be read in the current record by calling IOS_$INQ_REC_REMAINDER.

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

get-options

Options that control how IOS_$LOCATE performs the get operation, in IOS_$PUT_GET_OPTS_T format. This is a 2-byte integer. Specify a combination of the following set of predefined values:

IOS_$COND_OPT	Reads data, if available. (For example, data on an SIO line is not always available immediately.) If the data is not available, IOS_$GET returns the IOS_$GET_CONDITIONAL_FAILED status code and sets the return value of "ret-length" to 0.
IOS_$PREVIEW_OPT	Reads data but does not update the stream marker.

IOS_$NO_REC_BNDRY_OPT	Ignores record boundaries while reading data. For example, it ignores NEWLINE characters in a UASC object, which guarantees that the call fills the specified buffer. Some type managers might not support this call.
IOS_$PARTIAL_RECORD_OPT	Not meaningful for this call.

data-size

Maximum amount of data to be read, in bytes. This is a 4-byte integer.

OUTPUT PARAMETERS

data-ptr

A pointer to the located data, in UNIV_PTR format. This is a 4-byte integer. Note that this pointer remains valid only until the program invokes the next IOS call.

If IOS_$LOCATE is unable to return a pointer to the location of the data, it copies the data into a system buffer and then returns the address of the buffer in "data-ptr." (See the USAGE Section below for more details.)

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

You can use either IOS_$LOCATE or IOS_$GET to read data from system objects. IOS_$LOCATE returns a pointer to the data, while IOS_$GET copies the data into a buffer.

In most cases, use the IOS_$LOCATE call to read data because it is faster (IOS_$LOCATE does not perform a copy).

You will want to use IOS_$GET when you need to read more data than can be obtained in one call, because the pointer remains valid for only one call. For example, when you need to read and rearrange a number of lines from an object.

Normally, IOS_$LOCATE locates data and returns a pointer to the data. However, not all managers support the internal buffering necessary for IOS_$LOCATE to work this way. In these cases, IOS_$LOCATE will not be able to return a pointer to the data.

Instead, IOS_$LOCATE actually creates a buffer and then calls IOS_$GET to perform the get call. In this case, IOS_$LOCATE is no more efficient than IOS_$GET. The size of the buffer that IOS_$LOCATE creates is either the length you specify in "data-size," or 1024 bytes, whichever is the smaller.

Use IOS_$SET_LOCATE_BUFFER_SIZE to specify a buffer larger than 1024 bytes, if necessary. In this case, IOS_$LOCATE is no more efficient than IOS_$GET.

See the IOS_$SET_LOCATE_BUFFER_SIZE call description for more information.

IOS_$OPEN

Opens a stream to an existing object.

FORMAT

```
stream-id = IOS_$OPEN (pathname, namelength, open-options, status)
```

RETURN VALUE

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

INPUT PARAMETERS

pathname
Name of the object to be opened, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelength
Length of the pathname. This is a 2-byte integer.

open-options
Options available at open time, in IOS_$OPEN_OPTIONS_T format. This is a 2-byte integer. Specify a combination of the following set of predefined values:

IOS_$NO_OPEN_DELAY_OPT	Return immediately, instead of waiting for the open call to complete.
IOS_$WRITE_OPT	Permit writing data to a new object. If a program tries to write on a stream for which you have not specified this option, it returns an error status.
IOS_$UNREGULATED_OPT	Permit shared (unregulated) concurrency mode.
IOS_$POSITION_TO_EOF_OPT	Position the stream marker at the end of the object (EOF). Use this to append data to an existing object.
IOS_$INQUIRE_ONLY_OPT	Open the object for attribute inquiries only; do not permit reading or writing of data.
IOS_$READ_INTEND_WRITE_OPT	Open the object for read access with the intent to eventually change the object's access to write access. This allows other processes to read the object; but they cannot have write or read-intend-write access.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

This routine opens a stream to the named object. It returns the stream ID to be used in subsequent stream activity with the object. An error occurs if the object does not exist. If the object already exists, IOS_$OPEN does not change its attributes.

IOS_$OPEN does not return information about the object's attributes. To get information about an object, use the calls with the prefix IOS_$INQ. To change an object's attributes, use the calls with the prefix IOS_$SET.

IOS_$PUT

Writes data into an object.

FORMAT

```
IOS_$PUT (stream-id, put-options, buffer, buffer-size, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

put-options
Options that control how IOS_$PUT performs the put operation, in IOS_$PUT_GET_OPTS_T format. This is a 2-byte integer. Specify any combination of the following set of predefined values:

IOS_$COND_OPT	Write a record only if it can be done without blocking. If the call would block, it returns the IOS_$PUT_CONDITIONAL_FAILED error status.
IOS_$PREVIEW_OPT	Write data but do not update the stream marker.
IOS_$PARTIAL_RECORD_OPT	Write a portion of a record but do not terminate it. IOS_$PUT terminates the record when you call IOS_$PUT without specifying this option. If you do not specify this option, IOS_$PUT writes a full record. You can use this option with record-oriented objects only. IOS_$PUT ignores this option if you specify it with any other type of objects.
IOS_$NO_REC_BNDRY_OPT	Not meaningful for this call.

buffer
Buffer to contain the data. This is a character array.

buffer-size
Size of the buffer containing the data, in bytes. This is a 4-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$PUT writes data into an object. Use "put-options," which is in the IOS_$PUT_GET_OPTS_T format, to write the data to the object in different ways.

If the object is record-oriented, you can write data to it record by record. This is the default action (for record-oriented objects) when you specify the default ([]) value in "put-option."

To write a single record with more than one put operation (for example, to write one field at a time), use the IOS_$PARTIAL_RECORD_OPT option. If you specify this option, IOS_$PUT writes the data, but does not terminate the record. IOS_$PUT terminates the record when you call it without specifying this option.

To write to objects which might not always be immediately available (for example, an MBX channel), you perform *conditional* put operations with the IOS_$COND_OPT option.

IOS_$REPLICATE

Creates a copy of a specified stream ID.

FORMAT

```
return_stream_id = IOS_$REPLICATE (stream_id_to_replicate, copy_stream_id,
                                  status)
```

RETURN VALUE

return_stream_id
Number of the new stream created, in IOS_$ID_T format. This is a 2-byte integer.

INPUT PARAMETERS

stream_id_to_replicate
Number of the stream to replicate, in IOS_$ID_T format. This is a 2-byte integer. This stream number remains a valid connection to the object after IOS_$REPLICATE completes successfully.

copy_stream_id
Number of the stream to use as the copy for "stream_id_to_replicate," in IOS_$ID_T format. This is a 2-byte integer.

If "copy_stream_id" is free, IOS_$REPLICATE returns that number in "return_stream_id." If "copy-stream-id" is in use, IOS_$REPLICATE begins searching from that number downward (lower numbers) until it finds a free stream number, and returns that number in "return_stream_id."

If the actual number of the copy stream is insignificant, specify the predefined constant IOS_$MAX. This value causes IOS_$REPLICATE to begin searching at the highest possible stream number and return the first free stream number it finds.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$REPLICATE to create a copy of an existing stream ID. The new stream ID refers to the same connection as the existing stream ID. Note that you must close *both* streams with IOS_$CLOSE before the stream connection actually closes.

IOS_$REPLICATE is identical to IOS_$DUP except that IOS_$REPICATE looks for a free stream in *descending* order from the specified stream ID, while IOS_$DUP looks in *ascending* order. Note that you use IOS_$DUP or IOS_$REPLICATE to *copy* existing stream ID's, both the existing and new stream ID's remain valid connections. However, you use IOS_$SWITCH to *replace* stream IDs; you "switch" the connection from the existing stream ID to the new stream ID.

You can use IOS_$REPLICATE to keep a stream connection open when passing it to a subroutine. Use IOS_$REPLICATE to create a copy of the stream ID before passing it. This way, the subroutine cannot close the connection to the object because *all* copies of the stream connection must be closed before the connection itself gets closed.

IOS_$RELPLICATE is analagous to UNIX DUP.

IOS_$SEEK

Performs an absolute or relative seek using byte or record positioning.

FORMAT

```
IOS_$SEEK (stream-id, abs-rel, seek-type, offset, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

abs-rel
Value specifying the base for the seek operation, in IOS_$ABS_REL_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$RELATIVE	The seek is relative to the current position.
IOS_$ABSOLUTE	The seek is relative to the beginning of the object (BOF).

seek-type
The type of seek to be performed, in IOS_$SEEK_TYPE_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$REC_SEEK	Record-oriented seek.
IOS_$BYTE_SEEK	Byte-oriented seek.

offset
A signed integer offset value indicating the number of records or bytes from the seek base to position the stream marker. This is a 4-byte integer.

If the integer is a positive number, IOS_$SEEK uses BOF as the seek base and searches forward. If the integer is a negative number, IOS_$SEEK uses EOF as the seek base and searches backward. Whether the offset indicates bytes or records depends on the type of seek you specified in "seek-type."

You can get an offset number to use in an absolute seek with the calls IOS_$INQ_BYTE_POS and IOS_$INQ_REC_POS.

Note that both byte and record positions are *zero-based*; consequently, the first byte or record number is 0.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SEEK to seek to an absolute or relative byte or record position within an object.

You can use this call with the IOS_$INQ_BYTE_POS and IOS_$INQ_REC_POS calls to perform absolute position seeks.

IOS_$SEEK_FULL_KEY

Performs a seek using a full (8-byte) seek key.

FORMAT

```
IOS_$SEEK_FULL_KEY (stream-id, full-key, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

full-key
A full seek key, in IOS_$SEEK_KEY_T format. This data type is 8 bytes long. See the IOS Data Types section for more information.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Before performing a full key seek, you must first obtain a full seek key by using the IOS_$INQ_FULL_KEY call. This call allows you to inquire about a seek key before writing some data, and then store the seek key. To access the data at a later time, position the stream marker by calling the IOS_$SEEK_FULL_KEY call with the stored seek key, and then get the data with an IOS get call (IOS_$GET or IOS_$LOCATE).

IOS_$SEEK_SHORT_KEY

Performs a seek using a short (4-byte) seek key.

FORMAT

```
IOS_$SEEK_SHORT_KEY (stream-id, short-key, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

short-key
A short seek key. This is a 4-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Before performing a short key seek, you must first obtain a short seek key by using the IOS_$INQ_SHORT_KEY call. This call allows you to inquire about a seek key before writing some data, and then store the seek key. To access the data at a later time, position the stream marker by calling IOS_$SEEK_SHORT_KEY with the stored seek key, and then get the data with an IOS get call (IOS_$GET or IOS_$LOCATE).

IOS_$SEEK_TO_BOF

Positions the stream marker to the beginning of an object.

FORMAT

```
IOS_$SEEK_TO_BOF (stream-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SEEK_TO_BOF to position the stream marker to the beginning of an object (BOF). Use this call when performing a nonkeyed seek on an object.

IOS_$SEEK_TO_EOF

Positions the stream marker to the end of an object.

FORMAT

```
IOS_$SEEK_TO_EOF (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SEEK_TO_EOF to position the stream marker to the end of an object (EOF). Use this call when performing a nonkeyed seek on an object.

IOS_$SET_CONN_FLAG

Changes the set of connection attributes associated with a stream connection.

FORMAT

```
IOS_$SET_CONN_FLAG (stream-id, conn-flag, on-off, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer..

conn-flag

Flag indicating which attribute of the specified connection you want to change, in IOS_$CONN_FLAG_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$CF_TTY	Connection behaves like a terminal.
IOS_$CF_IPC	Connection behaves like an interprocess communication (IPC) channel.
IOS_$CF_VT	Connection behaves like a DOMAIN Display Manager pad.
IOS_$CF_WRITE	Connection can be written to.
IOS_$CF_APPEND	Connection's stream marker will be positioned at the end of the object (EOF) before each put call.
IOS_$CF_UNREGULATED	Connection is open for unregulated (shared) concurrency mode.
IOS_$CF_READ_INTEND_WRITE	Connection is open for read access, and can be changed to write access. Other connections can have read access, but not write or read-intend-write access.

on-off

Boolean value indicating whether the specified attribute should be included in the set (on), or removed from the set (off).

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SET_CONN_FLAG to change the attributes of a connection. Note that objects do not support *all* connection attributes. To determine the connection's current set of attributes, use IOS_$INQ_CONN_FLAGS before using this call.

To change the set of attributes, you must call IOS_$SET_CONN_FLAG for *each* connection attribute you want to change. To add an attribute to the set, call IOS_$SET_CONN_FLAG, specifying the desired attribute, and set the "on-off" parameter to TRUE. To remove an attribute from the set, use this call, specifying the attribute to remove, and set the "on-off" parameter to FALSE.

Before an object can permit the operation indicated by an attribute, the object's *manager* and *connection* attributes must permit the operation as well. For example, a manager's attribute set might contain the attribute that permits writing to an object (IOS_$MF_WRITE), but a specific object's connection attribute set might not include the IOS_$CF_WRITE attribute, which permits writing to the object. In this case, you *cannot* write to that particular object.

IOS_$SET_DIR

Changes the current working or naming directory.

FORMAT

```
IOS_$SET_DIR (pathname, namelength, dir_type, status)
```

INPUT PARAMETERS

pathname
Name of the directory to set, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelength
Length of "pathname." This is a 2-byte integer.

dir_type
Option specifying which type of directory to set, in IOS_$DIR_TYPE_T format. Specify one of the predefined values:

IOS_$WDIR	Name of the current working directory.
IOS_$NDIR	Name of the current naming directory.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use this call to change the current working or naming directory. You can use IOS_$GET_DIR to get the name of the current working or naming directory.

IOS_$SET_LOCATE_BUFFER_SIZE

Sets the size of the buffer that IOS_$LOCATE allocates.

FORMAT

```
IOS_$SET_LOCATE_BUFFER_SIZE (stream-id, buffer-size, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

buffer-size
Size of the buffer you want to allocate. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Normally, IOS_$LOCATE locates data and returns a pointer to the data. However, not all managers support the internal buffering necessary for IOS_$LOCATE to work this way. In these cases, IOS_$LOCATE will not be able to return a pointer to the data.

Instead, IOS_$LOCATE actually creates a buffer and then calls IOS_$GET to perform the get call. In this case, IOS_$LOCATE is no more efficient than IOS_$GET. The size of the buffer that IOS_$LOCATE creates is either the length you specify in "data-size," or 1024 bytes, whichever is the smaller.

Use IOS_$SET_LOCATE_BUFFER_SIZE to specify a buffer larger than 1024 bytes, if necessary.

For example, if you are using IOS_$LOCATE with a data-size parameter of 2000 bytes, and the manager of the object from which you are reading does not support internal buffering, the IOS_$LOCATE call, by default, will copy as much of the requested data as it can into a 1024-byte-long buffer and return a pointer to that buffer.

However, if you precede the IOS_$LOCATE call with a call to IOS_$SET_LOCATE_BUFFER_SIZE, specifying a buffer-size of 2000, the IOS_$LOCATE call will use a 2000-byte-long buffer and will be able to copy all the requested data into the buffer. This new buffer size will be valid as long as the stream exists.

IOS_$SET_OBJ_FLAG

Changes the set of object attributes associated with an object.

FORMAT

```
IOS_$SET_OBJ_FLAG (stream-id, obj-flag, on-off, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

obj-flag
Flag indicating which attribute of the specified object you want to change, in IOS_$OBJ_FLAG_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$OF_DELETE_ON_CLOSE	Object will be deleted when all its associated streams close.
IOS_$OF_SPARSE_OK	Object can be written as a sparse object.
IOS_$OF_ASCII	Object contains ASCII data.
IOS_$OF_FTNCC	Object uses FORTRAN carriage control characters.
IOS_$OF_COND	Get or put calls to the object will be performed conditionally, as if the IOS_$COND_OPT was specified on a get or put call.

on-off
Boolean value indicating whether the specified attribute should be included in the set (on), or removed from the set (off).

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SET_OBJ_FLAGS to change the attributes of an object. Note that objects do not support *all* object attributes. To determine the object's current attribute set, use the IOS_$INQ_OBJ_FLAGS call.

To change an object's attribute set, you must call IOS_$SET_OBJ_FLAG once for each object attribute you want to change. To add an attribute to the set, call IOS_$SET_OBJ_FLAG, specifying the desired attribute, and set the "on-off" parameter to TRUE. To remove an attribute from the set, use this call, specifying the attribute to remove, and set the "on-off" parameter to FALSE.

Before an object can permit the operation indicated by an attribute, the object's *manager* and *object* attributes must permit the operation as well. For example, a manager's attribute set might contain the attribute that allows the object to perform put and get calls conditionally (IOS_$MF_COND), but a specific object's object attribute set might not include the IOS_$OF_COND attribute. In this case, you *cannot* make conditional put or get calls to that particular object.

IOS_$SET_REC_TYPE

Sets the record type format and (optionally) record length of a file.

FORMAT

```
IOS_$SET_REC_TYPE (stream-id, record-type, record-length, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

record-type

Type of record format to change for the specified object, in IOS_$RTYPE_T format. This is a 2-byte integer. Specify one of the following predefined values:

IOS_$V1	Variable-length records with count fields.
IOS_$F1	Fixed-length records without count fields.
IOS_$F2	Fixed-length records with count fields. However, IOS_$PUT can change the IOS_$F2 type to IOS_$V1 implicitly. (See Usage section below.)
IOS_$EXPLICIT_F2	Fixed-length records that IOS_$PUT *cannot* implicitly change to variable-length records. (IOS_$PUT can change the IOS_$F2 to IOS_$V1 implicitly. See Usage section below.)
IOS_$UNDEF	No record structure.

record-length

Length to set for the fixed-length records of the object. This is a 4-byte integer. Specify this value only if the object is empty.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

By default, a record-oriented object has fixed-length records (IOS_$F2). They remain fixed-length records until IOS_$PUT writes records of different lengths. At this point, IOS_$PUT implicitly changes the objects to variable-length type (IOS_$V1). In some cases, you might want to explicitly set the record type to IOS_$EXPLICIT_F2 so that an attempt to write a variable-length record results in an error. To do so, use this call.

IOS_$SWITCH

Switches a stream from one stream ID to another stream ID.

FORMAT

```
ret-stream-id = IOS_$SWITCH (stream-id-to-switch, new-stream-id, status)
```

RETURN VALUE

ret-stream-id
Number of the new stream ID that replaces the existing stream ID, in IOS_$ID_T format. This is a 2-byte integer.

INPUT PARAMETERS

stream-id-to-switch
Number of the stream to switch, in IOS_$ID_T format. This is a 2-byte integer.

This stream number becomes *invalid* after the IOS_$SWITCH call completes sucessfully.

new-stream-id
Number of the stream to use as the new stream ID, in IOS_$ID_T format. This is a 2-byte integer.

If "new-stream-id" is free, IOS_$SWITCH returns this value in "ret-stream-id." If "new-stream-id" is in use, IOS_$SWITCH begins searching from that value downward (lower numbers) until it finds a free stream number and returns that number in "ret-stream-id."

If the actual number of the replacement stream is insignificant, specify the predefined constant IOS_$MAX. This value causes IOS_$SWITCH to begins searching at highest possible stream number and return the first free number it finds.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

Use IOS_$SWITCH to switch one stream ID for another. The new stream ID refers to the *same* connection as the old stream ID, making the old stream ID invalid.

Note that you use IOS_$SWTICH to *replace* stream IDs; you "switch" the connection from the existing stream ID to the new stream ID. However, you use IOS_$DUP or IOS_$REPLICATE to *copy* existing stream IDs, both the existing and new stream IDs remain valid connections.

IOS_$TRUNCATE

Deletes the contents of an object following the current stream marker.

FORMAT

```
IOS_$TRUNCATE (stream-id, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in IOS_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IOS Data Types section for more information.

USAGE

IOS_$TRUNCATE decreases the value of the object's length attribute to match the stream marker. (Writing data to a stream that lengthens the object implicitly increases this attribute's value.) This call sets the stream marker to the end of the object (EOF), effectively deleting any data in the object past the stream marker. If the stream position is already at EOF, IOS_$TRUNCATE has no effect.

Truncating an object does not close the stream.

ERRORS

IOS_$ALREADY_EXISTS
Object already exists; detected by IOS_$CREATE with IOS_$NO_PREXIST option.

IOS_$BAD_CHAR_SEEK
Attempted character seek before start of current (variable-length) record.

IOS_$BAD_COUNT_FIELD_IN_FILE
Count field for current record is wrong.

IOS_$BAD_FILE_HDR
Wrong stream file header.

IOS_$BAD_LOCATION
Bad location parameter on IOS_$CREATE call..

IOS_$BAD_OPEN_XP
OPEN_XP must reference a stream that is already open in this process.

IOS_$BAD_SHARED_CURSOR_REFCNT
Reference count on a shared object cursor went below zero.

IOS_$BOF_ERR
Attempted seek beyond beginning of object (BOF).

IOS_$BUFFER_TOO_BIG
Buffer size too large on IOS_$GET or IOS_$LOCATE call.

IOS_$BUFFER_TOO_SMALL
Buffer too small on IOS_$GET or IOS_$LOCATE call, warning.

IOS_$CANT_CHANGE_TYPE
Cannot change the type as requested, detected by IOS_$CREATE.

IOS_$CANT_DELETE_OLD_NAME
Added new name, but cannot delete old name.

IOS_$CANT_INITIALIZE
Cannot initialize an object of this type.

IOS_$CANT_SET_ADVISORY_LOCK
Advisory lock already set on this object.

IOS_$CONCURRENCY_VIOLATION
Requested access violates concurrency constraints, object is in use.

IOS_$DEVICE_MUST_BE_LOCAL
Cannot open stream to remote device.

IOS_$DIR_NOT_FOUND
Couldn't find directory in pathname on IOS_$CREATE.

IOS_$END_OF_FILE
End of file.

IOS_$FILE_NOT_EMPTY
Object not empty.

IOS_$FLAG_NOT_SUPPORTED
: Flag not supported for this object type.

IOS_$FROM_ID_NOT_OPEN
: Stream ID to switch not open on IOS_$SWITCH.

IOS_$FULL_REC_UNAVAIL
: IOS_$GET or IOS_$LOCATE requested a full record, but only part of the record was available. The call returns the part that is available along with this warning that there is still more room in the buffer.

IOS_$GET_CONDITIONAL_FAILED
: Cannot read any data because the stream is empty; detected by IOS_$COND_OPT option.

IOS_$ID_OOR
: Stream ID is out-of-range or invalid.

IOS_$ILLEGAL_NAME_REDEFINE
: Attempted name change would require object to be moved, detected by IOS_$CHANGE_PATH_NAME.

IOS_$ILLEGAL_OBJ_TYPE
: Cannot open a stream for this type of object.

IOS_$ILLEGAL_OPERATION
: Operation illegal on named stream.

IOS_$ILLEGAL_PAD_CREATE_TYPE
: Cannot perform this operation on a pad type.

IOS_$ILLEGAL_PARAM_COMB
: Illegal parameter combination for this operation.

IOS_$ILLEGAL_W_VAR_LGTH_RECS
: Operation illegal with variable-length records.

IOS_$INQ_ONLY_ERROR
: Can only open this operation for inquiries only.

IOS_$INSUFFICIENT_RIGHTS
: Insufficient rights for requested access to object.

IOS_$INSUFF_MEMORY
: Not enough address space.

IOS_$INTERNAL_FATAL_ERR
: Internal fatal error on table re-verify operation.

IOS_$INTERNAL_MM_ERR
: Internal fatal error in stream memory management (windowing).

IOS_$INVALID_DATA
: Cannot write this data to object.

IOS_$NAME_NOT_FOUND
: Name not found.

IOS_$NAME_REQD
: Must specify name on IOS_$OPEN.

IOS_$NEED_MOVE_MODE
IOS_$LOCATE operation refused, try IOS_$GET.

IOS_$NEVER_CLOSED
System (or process) crash prevented complete close of object.

IOS_$NO_ADVISORY_LOCK_SET
No advisory lock to unlock.

IOS_$NO_AVAIL_TARGET
No available target stream to switch to on IOS_$SWITCH.

IOS_$NO_MORE_STREAMS
No more available stream IDs.

IOS_$NO_RIGHTS
No rights to access object.

IOS_$NO_TABLE_SPACE
Internal error.

IOS_$NOT_A_DIRECTORY
Name specified is not a directory detected by IOS_$GET_DIR or _$SET_DIR.

IOS_$NOT_AT_REC_BNDRY
Cannot perform operation with short key -- must be at a record boundary.

IOS_$NOT_OPEN
Operation attempted on unopened stream.

IOS_$OBJ_DELETED
Object has been deleted while open on this stream.

IOS_$OBJECT_NOT_FOUND
Object associated with this name not found even though name exists.

IOS_$OBJECT_READ_ONLY
Cannot open this object for writing.

IOS_$OUT_OF_SHARED_CURSORS
Internal error.

IOS_$PART_REC_WARN
Partial record at EOF on IOS_$CLOSE -- warning only.

IOS_$PERM_FILE_NEEDS_NAME
Only temporary objects can be unnamed, you must name a permanent object.

IOS_$PUT_BAD_REC_LEN
Attempted an IOS_$PUT on a record of the wrong length.

IOS_$PUT_CONDITIONAL_FAILED
Cannot write any data because the stream is full, detected by IOS_$COND_OPT option.

IOS_$READ_ONLY_ERR
Attempted to write to read-only stream.

IOS_$RESOURCE_LOCK_ERR
Unable to lock resources required to process request.

IOS_$SIO_NOT_LOCAL
: No stream found in conditional put, or cannot open a remote SIO line.

IOS_$SOMETHING_FAILED
: Cannot locate attribute set inquiring about manager, connection or object attributes; or cannot change the connection or object attribute requested.

IOS_$TARGET_INUSE
: Target ID already in use on IOS_$SWITCH, no available stream IDs.

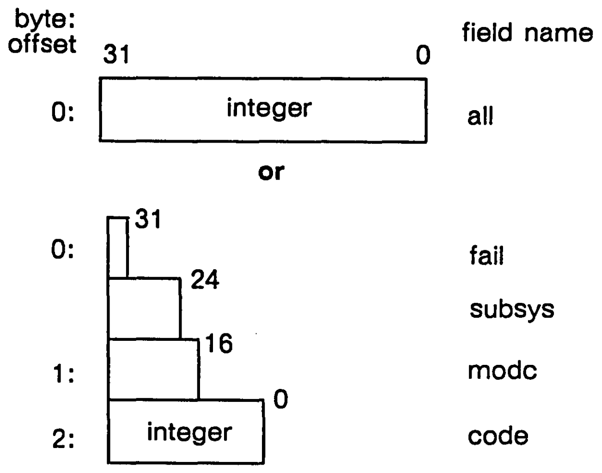
IOS_$XP_BUF_TOO_SMALL
: Buffer supplied to IOS_$EXPORT too small.

IPC

This section describes the data types, the call syntax, and the error codes for the IPC programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

IPC_$DATA_T	An array of up to 1024 characters. The data portion of an IPC datagram.
IPC_$HDR_INFO_T	An array of up to 128 characters. The header portion of an IPC datagram.
IPC_$SOCKET_HANDLE_T	An array of 20 characters. A handle for an IPC socket.
NAME_$PNAME_T	An array of up to 256 characters. A DOMAIN pathname.
STATUS_$T	A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

IPC_$CLOSE

Closes an IPC socket.

FORMAT

```
IPC_$CLOSE (pathname, length, status)
```

INPUT PARAMETERS

pathname
Pathname for the file where the socket handle is stored, in NAME_$PNAME_T format. This is an array of up to 256 characters. Specify a file that was created by a previous IPC_$CREATE call.

length
Length of the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$CLOSE closes a socket and removes its handle from the file where the handle is stored. IPC_$CLOSE does not, however, delete the socket handle file. To delete this file, use IPC_$DELETE.

IPC_$CREATE

Creates a file where an IPC socket handle can be stored.

FORMAT

```
IPC_$CREATE (pathname, length, status)
```

INPUT PARAMETERS

pathname
Pathname for a file where a socket handle can be stored, in NAME_$PNAME_T format. This is an array of up to 256 characters.

length
Length of the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$CREATE creates a special type of DOMAIN system object that is used only for socket handles. When you open a socket, the system obtains a handle for the socket and stores this handle in the file that you specify. You can open a socket only if you have previously used IPC_$CREATE to create a file for the handle.

IPC_$DELETE

Deletes a file that was used to store an IPC socket handle.

FORMAT

```
IPC_$DELETE (pathname, length, status)
```

INPUT PARAMETERS

pathname
Pathname for the file where the socket handle was stored, in NAME_$PNAME_T format. This is an array of up to 256 characters.

length
Length of the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$DELETE deletes a file that the system used to store a handle for an open socket. You must call IPC_$CLOSE to close the socket before you can delete the file containing the socket's handle.

IPC_$GET_EC

Gets a pointer to the eventcount associated with an IPC socket.

FORMAT

```
IPC_$GET_EC (handle, ec-ptr, status)
```

INPUT PARAMETERS

handle
Handle for the socket whose eventcount you are getting, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters.

OUTPUT PARAMETERS

ec-ptr
Pointer to the eventcount, in EC2_$PTR_T format. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$GET_EC gets a pointer to the eventcount associated with an IPC socket. You can use this eventcount to wait for incoming datagrams. Use EC2 system calls to read the eventcount value and wait for datagrams.

IPC_$GET_EC is useful when you want to wait for messages arriving in more than one socket. To wait for messages from only one socket, use IPC_$WAIT or IPC_$SAR.

IPC_$OPEN

Opens an available IPC socket, obtains its handle, and places the handle in a file.

FORMAT

```
IPC_$OPEN (pathname, length, depth, handle, status)
```

INPUT PARAMETERS

pathname
Pathname for the file in which to store the handle, in NAME_$PNAME_T format. This is an array of up to 256 characters. Specify a file that you have created with a previous IPC_$CREATE call.

length
Length of the pathname. This is a 2-byte integer.

depth
Depth of the socket. The depth defines how many datagrams a socket can hold. Allowable values are one through four.

OUTPUT PARAMETERS

handle
Handle for the open socket, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$OPEN opens an available IPC socket on your program's local node. In addition, IPC_$OPEN places the socket's handle in the file you specify. After opening a socket, you can receive datagrams in it. A program must use your socket's handle to send you a message.

User programs running on a node can open a maximum of eight sockets on that node. Only one program at a time can open any socket.

You must use IPC_$CREATE to create a file for the socket handle before you can open a socket.

IPC_$RCV

Gets a datagram that has been received in an IPC socket. This call copies the datagram to the buffers that you specify.

FORMAT

```
IPC_$RCV (handle, hdr-buflen, data-buflen, from-handle,
          hdr-buf, hdr-length, data-buf, data-length, status)
```

INPUT PARAMETERS

handle
Handle for the socket that received the datagram, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters.

hdr-buflen
Length of the buffer where the datagram header will be copied. This is a 2-byte integer. This value defines the maximum number of header bytes that IPC_$RCV will get. An IPC datagram can contain up to 128 header bytes. Specify a length that can accommodate the longest header you expect to receive.

data-buflen
Length of the buffer where the data portion of the datagram will be copied. This is a 2-byte integer. This value defines the maximum number of data bytes that IPC_$RCV will get. The data portion of an IPC datagram can contain up to 1024 bytes. Specify a length that can accommodate the longest data you expect to receive.

OUTPUT PARAMETERS

from-handle
Handle for the socket where the datagram originated, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters. Use this handle to send a reply to the datagram you are currently getting.

hdr-buf
Buffer where the datagram header is copied. This buffer can contain up to 128 bytes.

hdr-length
Length, in bytes, of the header that is copied. This is a 2-byte integer.

data-buf
Buffer where the data portion of the datagram is copied. This buffer can contain up to 1024 bytes.

data-length
Length, in bytes, of the data that is copied. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

IPC_$RCV gets datagrams that have been received in a socket and copies them to your buffers. This call returns only the number of header and data bytes that you specify, even if the actual datagram (in the socket) contains more bytes.

IPC_$RCV gets datagrams in the order in which they arrive in the socket queue. If the socket queue is full when an incoming datagram arrives, the datagram is lost. You can use IPC_$RCV to get datagrams only from a socket that you have previously opened with IPC_$OPEN.

Usually, you wait for a datagram to arrive in a socket, and then call IPC_$RCV to get the datagram. If you call IPC_$RCV when the socket is empty, the call returns immediately with the status IPC_$SOCKET_EMPTY.

IPC_$RESOLVE

Obtain the handle for an open socket.

FORMAT

```
IPC_$RESOLVE (pathname, length, handle, status)
```

INPUT PARAMETERS

pathname
Pathname for the file containing the socket handle, in NAME_$PNAME_T format. This is an array of up to 256 characters. Specify a file that was created by a previous IPC_$CREATE call.

length
Length of the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

handle
Handle for the socket, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$RESOLVE returns the handle associated with an open socket. Use this call if you know a socket's pathname, but you need the socket handle to send a datagram.

IPC_$RESOLVE returns the error IPC_$SOCKET_NOT_OPEN if the handle file does not contain the handle for an open socket.

IPC_$SAR

Performs a single send/await-reply operation. This call sends a datagram, waits a specified amount of time for a reply, and copies the reply to the buffers you specify.

FORMAT

```
IPC_$SAR (retry-time, to-handle, in-hdr-buf, in-hdr-length, in-data-buf,
         in-data-length, out-hdr-buflen, out-data-buflen, out-hdr-buf,
         out-hdr-length, out-data-buf, out-data-length, status)
```

INPUT PARAMETERS

retry-time
Number of quarter-seconds to wait for a reply. This is a 2-byte integer.

to-handle
Handle for the destination socket, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters. The destination socket is where you are sending the datagram.

in-hdr-buf
Buffer that contains the header for the datagram you are sending. This buffer can contain up to 128 bytes.

in-hdr-length
Length, in bytes, of the header you are sending. This is a 2-byte integer.

in-data-buf
Buffer that contains the data portion of the datagram you are sending. This buffer can contain up to 1024 bytes.

in-data-length
Length, in bytes, of the data you are sending. This is a 2-byte integer.

out-hdr-buflen
Length of the buffer where the reply datagram header will be copied. This is a 2-byte integer. This value defines the maximum number of header bytes that IPC_$SAR will get from the reply datagram. The reply can contain up to 128 header bytes. Specify a length that can accommodate the longest header you expect to receive.

out-data-buflen
Length of the buffer where the data portion of the reply datagram will be copied. This is a 2-byte integer. This value defines the maximum number of data bytes that IPC_$SAR will get from the reply datagram. The data portion of a reply can contain up to 1024 bytes. Specify a length that can accommodate the longest data you expect to receive.

OUTPUT PARAMETERS

out-hdr-buf
Buffer where the header for the reply datagram is copied. This buffer can contain up to 128 bytes.

out-hdr-length
Length, in bytes, of the header that is copied. This is a 2-byte integer.

out-data-buf
Buffer where the data portion of the reply datagram is copied. This buffer can contain up to 1024 bytes.

out-data-length
Length, in bytes, of the data that is copied. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

Use IPC_$SAR to send a datagram to another process and wait a specified time for a reply. If the reply datagram does not arrive within the specified time, IPC_$SAR returns the status code IPC_$TIMEOUT.

IPC_$SAR returns only the number of header and data bytes that you specify, even if the actual datagram (in the socket) contains more bytes.

When you send a datagram that contains less than 128 bytes of information, you can place all the information in the header buffer. Then specify the data length as zero. It takes less time to send a datagram that contains only a header.

IPC_$SEND

Sends a datagram to an IPC socket.

FORMAT

```
IPC_$SEND (to-handle, reply-handle, hdr-buf, hdr-length,
          data-buf, data-length, status)
```

INPUT PARAMETERS

to-handle
Handle for the destination socket, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters. The destination socket is where you are sending the datagram.

reply-handle
Handle for the reply socket, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters. The reply socket is where you can receive a reply.

hdr-buf
Buffer that contains the header for the datagram you are sending. This buffer can contain up to 128 bytes.

hdr-length
Length, in bytes, of the datagram header. This is a 2-byte integer.

data-buf
Buffer that contains the data portion of the datagram you are sending. This buffer can contain up to 1024 bytes.

data-length
Length, in bytes, of the data portion of the datagram. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$SEND sends a datagram to the socket that you specify. To obtain a socket handle from a pathname, use IPC_$RESOLVE.

Even if IPC_$SEND completes successfully, there is no guarantee that the datagram will be received by the process you are sending it to. The programs using IPC datagrams are responsible for verifying that datagrams are successfully received. Note that you can use IPC_$SAR to perform a send/await reply operation with a single call.

When you send a datagram that contains less than 128 bytes of information, you can place all the information in the header buffer. Then specify the data length as zero. It takes less time to send a datagram that contains only a header.

IPC_$WAIT

Waits for a specified amount of time to receive a datagram in an IPC socket.

FORMAT

```
IPC_$WAIT (handle, wait-time, status)
```

INPUT PARAMETERS

handle
Handle for the socket that you are waiting to receive data in, in IPC_$SOCKET_HANDLE_T format. This is an array of 20 characters.

wait-time
Number of quarter-seconds to wait for a reply. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the IPC Data Types section for more information.

USAGE

IPC_$WAIT waits for a specified amount of time to receive a datagram in a socket. If a datagram is received before the time elapses, the call returns with the status STATUS_$OK. To get the datagram, use IPC_$RCV.

If IPC_$WAIT times out before a datagram is received, the call returns with the status IPC_$TIMEOUT. If you call IPC_$WAIT and there is a datagram already in the socket, the call returns immediately with a success status.

Note that you can use IPC_$SAR to perform a send/await reply operation with a single call. Also, if you want to wait for datagrams in more than one socket, use IPC_$GET_EC to get pointers to the appropriate eventcounts. Then use eventcount calls (EC2) to wait for datagrams.

ERRORS

IPC_$OK
Successful completion.

IPC_$NOMORE_SOCKETS
All the sockets are in use.

IPC_$NOT_IPC_OBJ
The specified pathname does not belong to an IPC object.

IPC_$NOT_OWNER
You did not open the socket so you cannot close it.

IPC_$RANGE_ERROR
Supplied socket number is outside legal range.

IPC_$SOCKET_ALREADY_OPEN
Specified socket is already open.

IPC_$SOCKET_EMPTY
There are no datagrams in the socket.

IPC_$SOCKET_NOT_OPEN
The specified socket is not open.

IPC_$TIMEOUT
The call timed out before a datagram was received.

IPC_$TOO_DEEP
Supplied socket depth is too big.

IPC_$TOO_MUCH_DATA
The data is too long to send.

STATUS_$OK
Successful completion.

MBX

This section describes the data types, the call syntax, and the error codes for the MBX programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

MBX_$CHN_MAX	255	Maximum number of channels that can be open to a mailbox.
MBX_$FIL_MAX	257*32768	Maximum mailbox size.
MBX_$MIN_CHN_SIZE	64	The minimum size of a channel buffer.
MBX_$MSG_MAX	1024	A mailbox message that is 1024 bytes long.
MBX_$MSG_TN	1023	For use when declaring a zero-based array that is MBX_$MSG_MAX bytes long.
MBX_$MSG_WMAX	512	A mailbox message that is 512 words long.
MBX_$MSG_WTN	511	For use when declaring a zero-based array that is MBX_$MSG_WMAX words long.
MBX_$REC_DATA_MAX	32760	The maximum length of the data portion of a mailbox message.
MBX_$REC_MSG_MAX	32766	The maximum length of a server message, including the header and data portions.
MBX_$SERV_MSG_HDR_LEN	6	Length of the mailbox header for a server message.
MBX_$SERV_MSG_MAX	1030	A server message that contains 1024 bytes of data plus a 6-byte header.
MBX_$VERSION	1	Current version of MBX.

DATA TYPES

EC2_$PTR_T	A 4-byte integer. A pointer to an eventcount.
MBX_$CHAN_NUM_T	A channel number. Possible values are integers from 0 through MBX_$CHN_MAX.

MBX_$CHAN_SET_T	A set of channel numbers of type MBX_$CHAN_NUM_T. The following Pascal example specifies channels 1, 4, and 7:

```
VAR
    chan_set : mbx_$chan_set_t
                   .
                   .
                   .

    chan_set := [ 1, 4, 7]
```

In a FORTRAN program, declare an 8-element array of 4-byte integers to indicate a channel set. Use the array as a mask in which the bits represent mailbox channels.

MBX_$EC_KEY_T	A 2-byte integer. A mailbox eventcount. One of the following pre-defined values:

MBX_$GETREC_EC_KEY
An eventcount that advances when the mailbox contains messages for you to get.

MBX_$PUTREC_EC_KEY
An eventcount that advances when enough room exists in the channel to hold the last message you unsuccessfully tried to put there.

MBX_$MTYPE_T	A 2-byte integer. A message type. One of the following pre-defined values:

MBX_$ACCEPT_OPEN_MT
A response from a server to accept a client's open request.

MBX_$CHANNEL_OPEN_MT
A request from a client to open a channel to a mailbox.

MBX_$DATA_MT
A data transmission.

MBX_$DATA_PARTIAL_MT
A partial data transmission.

MBX_$EOF_MT
An end of transmission notice.

MBX_$REJECT_OPEN_MT
A response from a server to reject a client's open request.

MBX_$NAME_T	An array of up to 256 characters. A mailbox name.

MBX_$MSG_HDR_T

A mailbox message header. The diagram below illustrates the MBX_$MSG_HDR_T data type:

predefined type	byte: offset		field name
	0:	integer	cnt
mbx_$mtype_t	2:	integer	mt
	4:	integer	chan

Field Description:

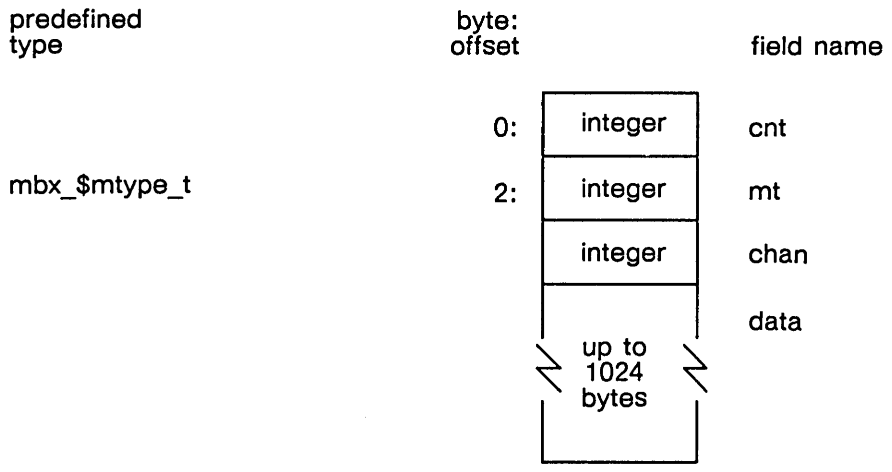
cnt
The total number of bytes in the message, including the header.

mt
A value representing a message type. This value is one of the predefined values of type MBX_$MTYPE_T.

chan
The channel of the client that sent the message, or that should receive the message.

MBX_$SERVER_MSG_T

A server message with up to 1024 data bytes. The diagram below illustrates the MBX_$SERVER_MSG_T data type:



Field Description:

cnt
The total number of bytes in the message, including the header.

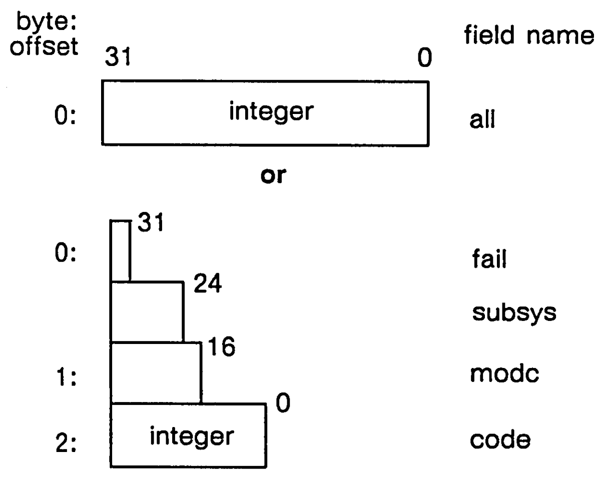
mt
A value representing a message type. This value is one of the predefined values of type MBX_$MTYPE_T.

chan
The channel of the client that sent the message, or that should receive the message.

data
The data portion of the message. This field can contain up to 1024 bytes.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

MBX_$CLIENT_WINDOW

Returns the buffer size for the mailbox that a client is using.

FORMAT

```
size = MBX_$CLIENT_WINDOW (handle, status)
```

RETURN VALUE

size

Buffer size for the mailbox. This is a 4-byte integer.

This value defines a window size when a client sends messages to a remote server. That is, the client cannot send messages that are larger than the mailbox's buffer.

INPUT PARAMETERS

handle

Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

When a client sends a message, the message is stored in a channel buffer until the server gets the message. The buffer size defines the maximum number of message bytes that the channel can hold at one time.

A client can use MBX_$CLIENT_WINDOW to get the size of the channel buffer. To get the size, MBX_$CLIENT_WINDOW queries the MBX_HELPER on the server's node. Note that MBX_$CLIENT_WINDOW returns the actual buffer size, not the number of unused bytes in the buffer.

MBX_$CLIENT_WINDOW only works correctly when the server you are inquiring about is on a node with SR9 or later software. If you call MBX_$CLIENT_WINDOW and the server is on a node with pre-SR9 software, MBX_$CLIENT_WINDOW returns the value 1158. This value is returned, even if the mailbox's actual buffer size is smaller. Therefore, this call does not provide a reliable way to determine the window size when sending messages to a server that is running on a node with pre-SR9 software.

MBX_$CLIENT_WINDOW is for use only by mailbox clients. A server should use MBX_$SERVER_WINDOW.

MBX_$CLOSE

Closes a mailbox or a channel.

FORMAT

```
MBX_$CLOSE (handle, status)
```

INPUT PARAMETERS

handle

Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

Both clients and servers can use MBX_$CLOSE. When called from a client, MBX_$CLOSE tells the server that the client is no longer using the channel. When called from a server, MBX_$CLOSE closes the mailbox.

After a client calls MBX_$CLOSE, the server should call MBX_$DEALLOCATE to deallocate the channel and free it for use by other clients. No other client can use the channel until it has been deallocated by the server.

If a server closes a mailbox while there are still active clients, the clients get errors on subsequent attempts to use the mailbox.

MBX_$COND_GET_REC_CHAN

Attempts to get a mailbox message from a specified channel.

FORMAT

```
MBX_$COND_GET_REC_CHAN (handle, channel, bufptr, buflen, retptr, retlen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle you obtained from MBX_$CREATE_SERVER.

channel
Channel to read from. This is a 2-byte integer. The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN.

bufptr
Pointer to a data buffer where the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter **retptr** to reference the message.

buflen
The number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$COND_GET_REC_CHAN can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$COND_GET_REC_CHAN requests a message from a specified channel. If there is no message, the call returns immediately with the status MBX_$CHANNEL_EMPTY. You can use an eventcount to tell when the status of the mailbox has changed. You get a mailbox eventcount with MBX_$GET_EC.

Only a server can use MBX_$COND_GET_REC_CHAN. To perform a conditional get operation from a client, use MBX_$GET_CONDITIONAL.

MBX_$COND_GET_REC_CHAN_SET

Attempts to get a mailbox record from a set of clients.

FORMAT

```
MBX_$COND_GET_REC_CHAN_SET (handle, chan-set, bufptr, buflen, retptr, retlen,
                            status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

chan-set
Set of channels to read from, in MBX_$CHAN_SET_T format. This is an 8-element array of 4-byte integers. See the MBX Data Types section for more information.

The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN. The channel number can range from 1 through MBX_$CHN_MAX.

bufptr
Pointer to a data buffer whe\re the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter **retptr** to reference the message.

buflen
Number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$COND_GET_REC_CHAN_SET can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$COND_GET_REC_CHAN_SET requests a message from a specified set of channels. If there is no message, the call returns immediately with the status MBX_$CHANNEL_EMPTY. You can use an eventcount to tell when the status of the mailbox has changed. You get a mailbox eventcount with MBX_$GET_EC.

Only a mailbox server can use this call. To perform a conditional get operation from a client, use MBX_$GET_CONDITIONAL.

MBX_$CREATE_SERVER

Creates and opens a server's mailbox.

FORMAT

```
MBX_$CREATE_SERVER (name, namelen, bufsize, maxchan, handle, status)
```

INPUT PARAMETERS

name

Name of the mailbox, in MBX_$NAME_T format. This is an array of up to 256 characters. Specify the name as a pathname to the mailbox file. If you use the name of a file that already exists, this call deletes the contents of the file. If the file already exists and it is in use, then the call returns an error.

namelen

Number of characters in the name. This is a 2-byte integer.

bufsize

Number of message bytes that the server and client can each store in a channel. This is a 2-byte integer. You must specify a buffer size of at least MBX_$MIN_CHN_SIZE (64 bytes). This allocates 128 bytes for each channel -- 64 bytes apiece for the server and the client buffers. The maximum buffer size is 32767.

The buffer size should be large enough to store the largest message you plan to send from a server or a client. Note that the maximum message length is MBX_$REC_MSG_MAX (32767), which includes 32761 data bytes plus a 6-byte header. If you specify a buffer size of less than MBX_$REC_MSG_MAX, you impose a lower limit on the total length of messages that pass through the mailbox.

Note that if you specify a buffer size that is greater than 1158, and the server is communicating with clients on remote nodes, the length of the transmitted messages may be limited by the MBX_HELPER on the client node. When a server puts a message into the mailbox and the message is intended for a remote client, the message passes through the system mailbox maintained by the remote node's MBX_HELPER. By default, this mailbox has a buffer size of 1158 bytes. To allow the remote node's mailbox to handle larger messages, use the -DATASIZE option to specify a larger buffer size when you start the MBX_HELPER. Specify a value that is at least as large as the largest message the server will send.

maxchan

Maximum number of channels that can be simultaneously open to the mailbox. This is a 2-byte integer. You can allow up to MBX_$CHN_MAX (255) channels.

OUTPUT PARAMETERS

handle

Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Subsequent calls use this handle to send and receive messages.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

A server uses MBX_$CREATE_SERVER to create a mailbox. Once the mailbox is open, clients use MBX_$OPEN to open communications channels to the mailbox.

In a secure network, a mailbox gets an access control list (ACL) that is determined by the ACL of the directory in which the mailbox is created. If servers and clients on different nodes use the mailbox, be sure that the server's MBX_HELPER has read and write access to the mailbox.

MBX_$DEALLOCATE

Releases a channel for use by another client.

FORMAT

```
MBX_$DEALLOCATE (handle, channel, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

channel
Channel to deallocate. This is a 2-byte integer. The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

Only a server can call MBX_$DEALLOCATE. A client uses MBX_$CLOSE to indicate the end of transmission over a channel. However, the server must deallocate the channel before another client can use it.

A server can deallocate a channel while a client is still using it; this both closes and deallocates the channel. The next time the client tries to use the channel, the client receives the error MBX_$CHANNEL_NOT_OPEN.

MBX_$GET_CONDITIONAL

Attempts to get a mailbox message.

FORMAT

```
MBX_$GET_CONDITIONAL (handle, bufptr, buflen, retptr, retlen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

bufptr
Pointer to a data buffer where the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter retptr to reference the message.

buflen
Number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$GET_CONDITIONAL can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$GET_CONDITIONAL gets a message if one is waiting. Otherwise, the call returns immediately with a completion status of MBX_$CHANNEL_EMPTY. You can use an eventcount to tell when the status of the mailbox has changed. You get a mailbox eventcount with MBX_$GET_EC.

Both servers and clients can use MBX_$GET_CONDITIONAL. When a server calls MBX_$GET_CONDITIONAL, the mailbox manager uses a scheduling algorithm to determine the channels to search for the next message. This algorithm guarantees fair service to each open channel.

MBX_$GET_EC

Gets a pointer to an eventcount for a mailbox event.

FORMAT

```
MBX_$GET_EC (mbx-handle, mbx-key, eventcount-pointer, status)
```

INPUT PARAMETERS

mbx-handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

mbx-key
Type of eventcount to get a pointer to, in MBX_$EC_KEY_T format. This is a 2-byte integer. Specify one of these predefined values:

MBX_$GETREC_EC_KEY	An eventcount that advances when the mailbox may contain messages for you to get. For a server, this eventcount may advance whenever there is anything to get from any open channel.
MBX_$PUTREC_EC_KEY	An eventcount that advances when there may be enough room in the channel to hold the last message you unsuccessfully tried to put there. A mailbox server sees only one MBX_$PUTREC_EC_KEY eventcount for the entire mailbox. If puts fail with MBX_$NO_ROOM_IN_CHANNEL on two channels of the same mailbox, the event's completion simply says that at least one channel may now take the message. One or both channels may now be capable of taking the respective message.

OUTPUT PARAMETERS

eventcount-pointer
A pointer to an eventcount, in EC2_$PTR_T format. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

After you use MBX_$GET_EC to get a mailbox eventcount, use EC2 calls to read eventcount values and wait for events.

MBX_$GET_REC

Gets a message from a mailbox.

FORMAT

```
MBX_$GET_REC (handle, bufptr, buflen, retptr, retlen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

bufptr
Pointer to a data buffer where the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter retptr to reference the message.

buflen
Number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$GET_REC can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$GET_REC gets a mailbox message. If there is no message in the mailbox, the call waits for one.

Both servers and clients can use MBX_$GET_REC. When a server calls MBX_$GET_REC, the mailbox manager uses a scheduling algorithm to determine the channel to search for the next message. This algorithm guarantees fair service to each open channel.

MBX_$GET_REC_CHAN

Gets a mailbox message from a specified channel.

FORMAT

```
MBX_$GET_REC_CHAN (handle, channel, bufptr, buflen, retptr, retlen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

channel
Channel to read from. This is a 2-byte integer. The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN.

bufptr
Pointer to a data buffer where the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter retptr to reference the message.

buflen
Number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$GET_REC_CHAN can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$GET_REC_CHAN requests a message from the specified mailbox and channel. If there is no message, the call waits for one. Only a mailbox server can use this call. To perform a get operation from a client, use MBX_$GET_REC.

MBX_$GET_REC_CHAN_SET

Gets a mailbox message from a specified set of channels.

FORMAT

```
MBX_$GET_REC_CHAN_SET (handle, chan-set, bufptr, buflen, retptr, retlen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

chan-set
Set of channels to read from, in MBX_$CHAN_SET_T format. This is an 8-element array of 4-byte integers. See the MBX Data Types section for more information.

The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN. The channel number can range from 1 through MBX_$CHN_MAX.

bufptr
Pointer to a data buffer where the message can be copied. This is a 4-byte integer.

Your program must allocate a data buffer, although the mailbox manager does not always copy messages to this buffer. Use the output parameter retptr to reference the message.

buflen
Number of bytes in the data buffer. This is a 4-byte integer. For a server, MBX will never return more than 32766 bytes. For a client, MBX will never return more than 32760 bytes.

OUTPUT PARAMETERS

retptr
Pointer to the buffer where the message is copied. This is a 4-byte integer.

retlen
Either the number of bytes in the returned message, or the number of message bytes waiting to be returned. This is a 4-byte integer.

MBX_$GET_REC_CHAN_SET can get as many bytes as you specify in buflen. If the message is less than or equal to buflen, then the call gets the entire message and retlen specifies the message length. If the message is greater than buflen, then the call gets the number of bytes specified in buflen. If this occurs, then retlen contains a negative value, the absolute value of which is the number of bytes remaining in the message. Get the remaining data with another call.

Note that a server sees the message header each time it gets a piece of the message. The count field contains the total length of the message -- not the length of the returned piece.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$GET_REC_CHAN_SET requests a message from the specified mailbox and set of channels. If there is no message, the call waits for one. Only a mailbox server can use this call. To perform a get operation from a client, use MBX_$GET_REC.

MBX_$OPEN

Opens a client channel to a mailbox.

FORMAT

```
MBX_$OPEN (name, namelen, bufptr, buflen, handle, status)
```

INPUT PARAMETERS

name
Name of the mailbox, in MBX_$NAME_T format. This is an array of up to 256 characters. Specify the name as a pathname to the mailbox created by the server.

namelen
Number of characters in the name. This is a 2-byte integer.

bufptr
Pointer to a buffer containing data to be sent with the open request. This is a 4-byte integer. If you are not sending data, specify a nil pointer.

buflen
Number of bytes of data you are sending. This is a 4-byte integer. If you are not sending data, specify a length of 0.

The maximum amount of data you can send with MBX_$OPEN is MBX_$MSG_MAX (1024) bytes, even if the mailbox message buffer is larger.

OUTPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Subsequent calls use this handle to send and receive messages.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$OPEN opens a channel from a client to an existing mailbox. Only a client can use this call.

This call makes the mailbox manager send the server a channel open request. The server must respond by accepting or rejecting the request. After the server responds, MBX_$OPEN returns a status code indicating whether the call was successful. The client does not see the acceptance or rejection as a message, but as the completion status of the MBX_$OPEN call.

MBX_$PUT_CHR

Sends a partial message from a client.

FORMAT

```
MBX_$PUT_CHR  (handle, bufptr, buflen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$OPEN.

bufptr
Pointer to the buffer that contains the message to be sent. This is a 4-byte integer.

buflen
Length of the message, in bytes. This is a 4-byte integer. For a client, the buffer can contain up to 32760 bytes.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$PUT_CHR is equivalent to MBX_$PUT_REC, except that MBX_$PUT_CHR informs the server that the message is a partial message. If the mailbox is full, this call waits until the mailbox has room for the message.

Only a client can call MBX_$PUT_CHR. A server can send a partial data message by using MBX_$PUT_REC or MBX_$PUT_REC_COND, and specifying a message type of MBX_$DATA_PARTIAL_MT. When the client gets such a message, the get call returns a status of MBX_$PARTIAL_RECORD to the client.

MBX_$PUT_CHR_COND

Attempts to send a partial message from a client.

FORMAT

```
MBX_$PUT_CHR_COND  (handle, bufptr, buflen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$OPEN.

bufptr
Pointer to the buffer that contains the message to be sent. This is a 4-byte integer.

buflen
Length of the message, in bytes. This is a 4-byte integer. For a client, the buffer can contain up to 32760 bytes.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$PUT_CHR_COND is equivalent to MBX_$PUT_REC_COND, except that MBX_$PUT_CHR_COND informs the server that the message is a partial message.

If the client's buffer is full, MBX_$PUT_CHR_COND returns immediately with a completion status of MBX_$NO_ROOM_IN_CHANNEL. You can use an eventcount to tell when the status of the mailbox eventcount has changed. You get a mailbox eventcount with MBX_$GET_EC.

Only a client can call MBX_$PUT_CHR_COND. A server can send a partial data message by using MBX_$PUT_REC or MBX_$PUT_REC_COND, and specifying a message type of MBX_$DATA_PARTIAL_MT. When the client gets such a message, the get call returns a status of MBX_$PARTIAL_RECORD to the client.

MBX_$PUT_REC

Puts a record in the mailbox.

FORMAT

```
MBX_$PUT_REC (handle, bufptr, buflen, status)
```

INPUT PARAMETERS

handle
Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

bufptr
Pointer to the buffer that contains the message to be sent. This is a 4-byte integer.

buflen
Length of the message, in bytes. This is a 4-byte integer. For a server, the message can contain up to 32766 bytes. For a client, the buffer can contain up to 32760 bytes.

If a server puts a message that is larger than 1158 bytes, and the client is on a remote node, the client node's MBX_HELPER must be able to handle the message. To handle the message, the client node's MBX_HELPER must have a queue data size that is at least as large as the message. Use MBX_$SERVER_WINDOW to determine the client node's queue data size.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

This call can be used by either servers or clients. Note, however, that servers and clients have different message formats. A server must include the 6-byte message header when sending a message. In contrast, a client sends only data.

If the channel is full, this call waits until there is room for the message.

MBX_$PUT_REC_COND

Attempts to put a message into a mailbox.

FORMAT

```
MBX_$PUT_REC_COND (handle, bufptr, buflen, status)
```

INPUT PARAMETERS

handle

Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER or MBX_$OPEN.

bufptr

Pointer to the buffer that contains the message to be sent. This is a 4-byte integer.

buflen

Length of the message, in bytes. This is a 4-byte integer. For a server, the message can contain up to 32766 bytes. For a client, the buffer can contain up to 32760 bytes.

If a server puts a message that is larger than 1158 bytes, and the client is on a remote node, the client node's MBX_HELPER must be able to handle the message. To handle the message, the client node's MBX_HELPER must have a queue data size that is at least as large as the message. Use MBX_$SERVER_WINDOW to determine the client node's queue data size.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

MBX_$PUT_REC_COND can be used by either servers or clients. Note, however, that servers and clients have different message formats. A server must include the 6-byte message header when sending a message. In contrast, a client sends only data.

If the channel is full, MBX_$PUT_REC_COND returns immediately, with a completion status of MBX_$NO_ROOM_IN_CHANNEL. You can use an eventcount to tell when the status of the mailbox eventcount has changed. You get a mailbox eventcount with MBX_$GET_EC.

MBX_$SERVER_WINDOW

Returns the buffer size for the mailbox maintained by the MBX_HELPER on a remote client's node.

FORMAT

```
size = MBX_$SERVER_WINDOW (handle, channel, status)
```

RETURN VALUE

size

Buffer size for the mailbox maintained by the MBX_HELPER on the remote client's node. This is a 4-byte integer.

This value defines a window size when a server sends messages to a remote client. That is, the server cannot send messages that are larger than the buffer for the remote MBX_HELPER's mailbox.

INPUT PARAMETERS

handle

Identifier for the mailbox, in UNIV_PTR format. This is a 4-byte integer. Use the handle returned by MBX_$CREATE_SERVER.

channel

Channel belonging to the client whose window size you are inquiring about. This is a 2-byte integer. The mailbox manager assigns a channel number to a client when the client calls MBX_$OPEN.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MBX Data Types section for more information.

USAGE

When a server puts a message into a mailbox and the message is intended for a client on a remote node, the message must pass through a system mailbox maintained by the client node's MBX_HELPER. Thus, the largest message that a server can send depends on the buffer size for the remote client's system mailbox. This buffer size was defined when the client node's MBX_HELPER was started. (The MBX_HELPER's -DATASIZE option defines a buffer size for the system mailbox.)

A server can use MBX_$SERVER_WINDOW to determine the buffer size for the remote client's system mailbox. To get this value, MBX_$SERVER_WINDOW queries the MBX_HELPER on the client's node. Note that MBX_$SERVER_WINDOW returns the actual buffer size, not the number of unused bytes in the buffer.

Note that if a server is communicating with clients on different nodes, the buffer size can differ on each node. Therefore, the server must use MBX_$SERVER_WINDOW to obtain the buffer size on each node.

MBX_$SERVER_WINDOW correctly returns the buffer size for clients on nodes with SR9 or later software. However, if you call MBX_$SERVER_WINDOW and the specified client is on a pre-SR9 node, then the call always returns the value 1158. 1158 is the minimum buffer size for mailboxes maintained by pre-SR9 MBX_HELPERs.

MBX_$SERVER_WINDOW is for use only by mailbox servers. A client should use MBX_$CLIENT_WINDOW.

ERRORS

MBX_$BAD_KEY
: Bad key.

MBX_$BUFFER_TOO_SMALL
: A server requested a message using a buffer smaller than 6 bytes. There must be enough room for the message header in all server message requests.

MBX_$CHANNEL_EMPTY
: There are no messages waiting in the channel. Received in response to an MBX_$GET_COND or MBX_$COND_GET_CHAN request.

MBX_$CHANNEL_NOT_OPEN
: For a server, the channel number given referred to a channel that is not presently open; for a client, the server has deallocated the client's channel.

MBX_$CLIENT_NO_RIGHTS
: The client can't access the local MBX_HELPER's SYSMBX.

MBX$CLIENT_SERVER_DEADLOCK
: A server tried to open a channel to itself; this is illegal.

MBX_$EOF
: The client has sent a message of type MBX_$EOF_MT. Received in response to an MBX_$GET_REC or MBX_$GET_COND request.

MBX_$FILE_IN_USE
: An MBX_$CREATE_SERVER or MBX_$OPEN request was made giving a mailbox pathname that is the pathname for a file presently in use.

MBX_$HANDLE_NOT_VALID
: The handle given does not point to a mailbox.

MBX_$HELPER_NO_RIGHTS
: The MBX_HELPER on the server's node can't access the server's mailbox.

MBX_$ILL_HANDLE
: The handle given is not a legal handle.

MBX_$MSG_TOO_BIG_FOR_CHANNEL
: An MBX_$PUT_CONDITIONAL or MBX_$PUT_REC request tried to send a message bigger than the maximum specified when the server created the mailbox.

MBX_$NO_MORE_CHANNELS
: An MBX_$OPEN was made to a mailbox with no free channels.

MBX_$NO_MORE_RESOURCES
: An MBX_$CREATE_SERVER or MBX_$OPEN request was made, and the process has insufficient resources (for example, address space) left to open the mailbox or the channel.

MBX_$NO_ROOM_IN_CHANNEL
: There is not enough room in the channel for the message. Received in response to an MBX_$PUT_CONDITIONAL request.

MBX_$NO_SERVERS
: An MBX_$OPEN was made to a mailbox without an active server.

MBX_$OPEN_REJECTED
The server rejected an MBX_$OPEN request.

MBX_$PARTIAL_RECORD
Returned data does not contain a complete record.

MBX_$REM_RCV_TIMOUT
A remote operation was attempted, and the network has failed.

MBX_$REM_SEND_FAILED
A remote operation was attempted, and the network has failed.

MBX_$REM_SERVICE_UNAVAILABLE
An MBX_$OPEN open request was made from a remote node when the MBX_HELPER program was not running on that node or the server's node.

MBX_$REMOTE_SERVICE_DENIED
An MBX_$OPEN request was made from a remote node, and there are not enough network services free to handle the request.

MBX_$SEQUENCED_SEND_FAILED
An internal error occurred while sending a message that is larger than 1158 bytes.

MBX_$SIZE_TOO_LARGE
MBX_$CREATE_SERVER request asked for a mailbox larger than the maximum.

MBX_$SIZE_TOO_SMALL
An MBX_$CREATE_SERVER request was made with a buffer size smaller than the minimum.

MBX_$TOO_MANY_CHANNELS
An MBX_$CREATE_SERVER request was made asking for more than the maximum number of channels.

MBX_$UNEXPECTED_CNTL_MSG
Received by a client when the last message the server sent on that channel had a message type of MBX_$ACCEPT_OPEN_MT, MBX_$REJECT_OPEN_MT, or MBX_$CHANNEL_OPEN_MT when such a message type was inappropriate. (MBX_$CHANNEL_OPEN_MT should never be used. The other two message types are only used in response to a message of type MBX_$CHANNEL_OPEN_MT.) Received in response to an MBX_$GET_REC or MBX_$GET_COND request.

MBX_$UNKNOWN_RQST
The client and server are using different versions of the mailbox manager (although the two versions have the same version number), and one of them made a request not recognized by the other manager.

MBX_$WRONG_VERSION_NUMBER
An MBX_$OPEN request was made using a mailbox manager with a different version number than the one used to create the mailbox.

STATUS_$OK
Successful completion.

MS

This section describes the data types, the call syntax, and the error codes for the MS programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

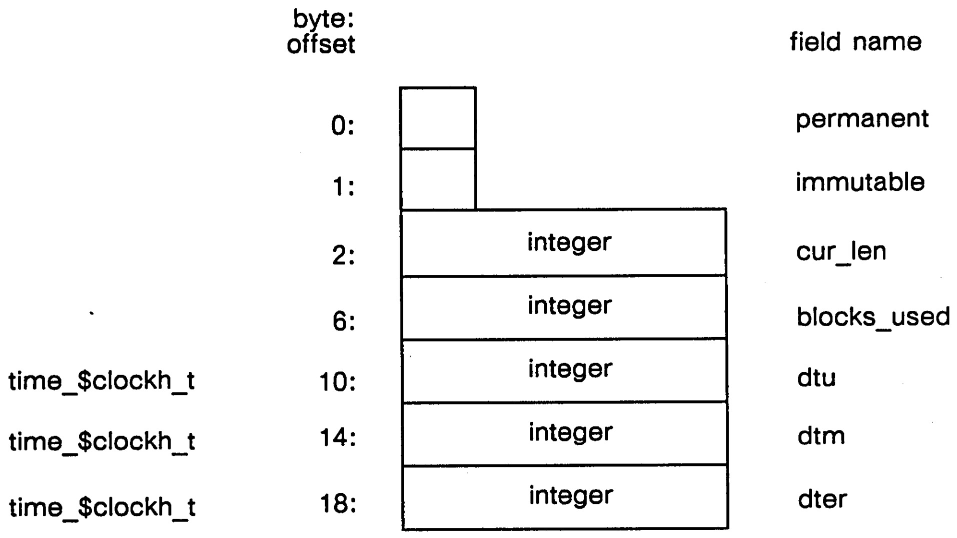
MS_$EXTEND	TRUE	The object can be extended.
MS_$NO_EXTEND	FALSE	The object cannot be extended.

DATA TYPES

MS_$ACC_MODE_T	A 2-byte integer. Access mode for an object. One of the following predefined values: MS_$R Read access. MS_$RX Read and execute access. MS_$WR Read and write access. MS_$WRX Write and execute access. MS_$RIW Read with intent to write.
MS_$ACCESS_T	A 2-byte integer. Usage patterns for accessing a file. One of the following predefined values: MS_$NORMAL Normal use. MS_$RANDOM Random access use. MS_$SEQUENTIAL Sequential access use.
MS_$ADVICE_OPT_T	Reserved for future use.
MS_$ADVICE_T	Four bytes that are reserved for future use.

MS_$ATTRIB_T

An attribute record. The diagram below illustrates the MS_$ATTRIB_T data type:



Field Description:

permanent
A boolean value that indicates whether the object is permanent (TRUE) or temporary (FALSE)

immutable
A boolean value that indicates whether the object can be modified. The value TRUE means that the object is immutable. The value FALSE means that the object is not immutable and can therefore be modified.

cur_len
Current length, in bytes, of the object.

blocks_used
The number of blocks used for the object.

dtu
Date-time used, in TIME_$CLOCKH_T format.

dtm
Date-time modified, in TIME_$CLOCKH_T format.

dtcr
Date-time created, in TIME_$CLOCKH_T format.

MS_$CONC_MODE_T

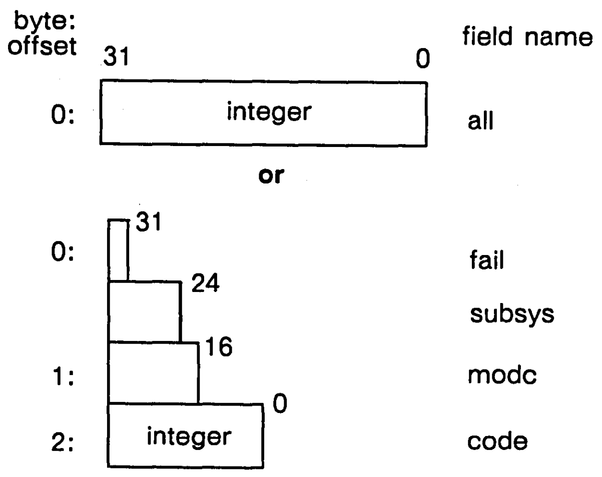
A 2-byte integer. Concurrency mode for an object. One of the following predefined values:

MS_$NR_XOR_1W
Allows one writer or any number of readers.

MS_$COWRITERS
Allows any number of readers and/or writers.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

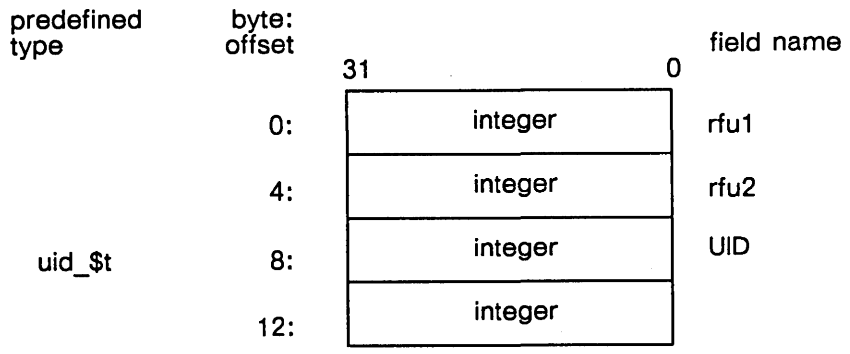
subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

XOID_$T

Unique identifier of an object. Used by type managers only. The diagram below illustrates the XOID_$T data type:



Field Description:

rfu1
Reserved for future use.

rfu2
Reserved for future use.

UID
Unique identifier for an object.

MS_$ADVICE

Provides the operating system with information on how you plan to access an object. This information helps the system optimize performance when managing the object.

FORMAT

```
MS_$ADVICE (address, length, access, options, record-length, status)
```

INPUT PARAMETERS

address

Pointer to the first byte to provide advice for, UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$CRMAPL, MS_$MAPL, or MS_$REMAP.

length

Number of bytes to provide advice for. This is a 4-byte integer.

access

Method of accessing the object, in MS_$ACCESS_T format. Specify only one of the following predefined values:

MS_$NORMAL	You do not have a predicted manner for accessing the object. This is the default if a program never uses MS_$ADVICE.
MS_$RANDOM	You access the object randomly.
MS_$SEQUENTIAL	You access the object sequentially.

options

Reserved for future use, in MS_$ADVICE_T format. This is a 4-byte integer. In Pascal, specify this parameter using the empty set []. In C and FORTRAN, declare a variable and initialize it to 0.

record-length

Number of bytes in a record in the mapped object. This is a 4-byte integer. If you do not know the record length, or if the object is not record-structured, specify 0.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information.

USAGE

MS_$ADVICE provides the operating system with information on how you plan to access an object. When you work with a mapped object, the system brings pages into memory as needed. By using MS_$ADVICE, you can change the number of pages that the system gets when a page fault occurs. This helps the system provide better performance when managing the object on your behalf.

Although it is not required that you use MS_$ADVICE, you should use it whenever you have a predicted type of file access. In addition, you can use MS_$ADVICE more than once to change the advice for a mapped object.

If you remap an object with MS_$REMAP, the advice in effect for the first part of the currently mapped section is propagated to the newly mapped section.

MS_$ATTRIBUTES

Returns the selected attributes of a mapped object.

FORMAT

```
MS_$ATTRIBUTES (address, attrib-buf, attrib-len, attrib-max, status)
```

INPUT PARAMETERS

address
Pointer to the first byte of the currently mapped portion of the object, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

OUTPUT PARAMETERS

attrib-buf
Buffer in which to receive the attributes, in MS_$ATTRIB_T format. This data type is 22 bytes long. See the MS Data Types section for more information.

attrib-len
Length of the attributes returned in the attributes buffer. This is a 2-byte integer.

INPUT PARAMETERS

attrib-max
Length of the attributes buffer. This is a 2-byte integer. Specify the length of the attributes buffer in the **attrib_buf** parameter. This value defines the maximum amount of information that MS_$ATTRIBUTES can return.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information.

USAGE

Use MS_$ATTRIBUTES to get information about selected attributes of a mapped object.

MS_$CRMAPL

Creates, maps, and locks an object.

FORMAT

```
address = MS_$CRMAPL (name, name-length, start, desired-length,
                      concurrency, status)
```

RETURN VALUE

address

Pointer to the first mapped byte of the object, in UNIV_PTR format. This is a 4-byte integer.

The first mapped byte is not necessarily the first byte of the object; it is the byte you specify in the start parameter.

INPUT PARAMETERS

name

Pathname of the object to be mapped, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name-length

Length of the pathname. This is a 2-byte integer.

start

First byte to be mapped. This is a 4-byte positive integer. To specify the first byte in an object, provide a start value of 0.

desired-length

Number of bytes to map, including the start byte. This is a 4-byte positive integer.

concurrency

Concurrency mode for the object, in MS_$CONC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$NR_XOR_1W	Allows one writer or any number of readers.
MS_$COWRITERS	Allows any number of readers and/or writers.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Object created.
MS_$BAD_ACCESS	Illegal concurrency value.
MS_$IN_USE	Object is currently locked.
MS_$NO_SPACE	Insufficient virtual address space to map.
NAME_$ALREADY_EXISTS	Name given already exists.
Other naming server errors	See the NAME_$ error codes.

USAGE

MS_$CRMAPL creates a file only if the name you specify does not already exist. The call implicitly uses an MS manager access mode of MS_$WR. Thus the object is always mapped for write access. You can get an exclusive write lock (if you specify a concurrency of MS_$WR_XOR_1W) or you can get a shared write lock (if you specify a concurrency of MS_$COWRITERS.) See the description of MS_$MAPL for more information on locks.

MS_$CRMAPL always uses an extend value of TRUE. Thus you can extend the object to the length you specify in the desired-length parameter.

MS_$FW_FILE

Forces the system to write a mapped file onto disk.

FORMAT

```
MS_$FW_FILE (address, status)
```

INPUT PARAMETERS

address

Pointer to the first byte of the currently mapped portion of the object, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information.

USAGE

When you work with mapped objects, the system uses a predefined set of conditions to determine when to write information (stored in memory) onto the disk. However, if you need to supplement the system's actions, you can use MS_$FW_FILE to force the system to write an object onto disk.

When you use MS_$FW_FILE, the system force writes the entire object, even if the currently mapped portion does not begin at byte 0. However, the system writes only the changed portions of the object onto the disk.

When you force-write a permanent object, the system also force-writes the directory where the object is cataloged.

MS_$MAPL

Maps the specified portion of a file-system object into an available region of the process address space. This call also locks the object.

FORMAT

```
address = MS_$MAPL (name, name-length, start, desired-length, concurrency,
                    access, extend, length-mapped, status)
```

RETURN VALUE

address
Pointer to the first mapped byte of the object, in UNIV_PTR format. This is a 4-byte integer.

The first mapped byte is not necessarily the first byte of the object; it is the byte you specify in the start parameter.

INPUT PARAMETERS

name
Pathname of the object to be mapped, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name-length
Length of the pathname. This is a 2-byte integer.

start
First byte to be mapped. This is a 4-byte positive integer. To specify the first byte of an object, provide a start value of 0.

desired-length
Number of bytes to map, including the start byte. This is a 4-byte positive integer.

concurrency
Concurrency mode for the object, in MS_$CONC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$NR_XOR_1W	Allows one writer or any number of readers.
MS_$COWRITERS	Allows any number of readers and/or writers.

access

The access mode desired, in MS_$ACC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$R	Read access.
MS_$RX	Read and execute access.
MS_$WR	Read and write access.
MS_$WRX	Write and execute access.
MS_$RIW	Read with intent to write.

The access requested must be a subset of the access permitted by the protection for the object.

extend

A Boolean value that indicates whether the object can be extended. The value TRUE indicates that the length given in the desired-length parameter should be mapped, even if the object is shorter. Writing beyond the end of the object, but within the space mapped, extends the object. FALSE indicates that the amount mapped should be no greater than the actual length of the file.

OUTPUT PARAMETERS

length-mapped

Number of bytes actually mapped. This is a 4-byte positive integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Object created.
MS_$BAD_ACCESS	Access type is illegal.
MS_$IN_USE	Object could not be locked.
NAME_$NOT_FOUND	No object exists with the given name.
Other naming-server errors	See the NAME_$ error codes.

USAGE

Use MS_$MAPL to map files that contain data in a user-defined format. For example, font files are a case where MS_$MAPL is appropriate. Do not use MS_$MAPL to access DOMAIN record structured files; use STREAM_$ calls to access these files.

MS_$MAPL locks a file, in addition to mapping it. The lock is determined by the concurrency and access modes that you specify. MS_$MAPL can obtain the following types of locks:

Lock	*Concurrency Mode*	*Access Mode*
`Protected Read`	`MS_$NR_XOR_1W`	`MS_$R or MS_$RX`
`Protected RIW`	`MS_$NR_XOR_1W`	`MS_$RIW`
`Shared Read`	`MS_$COWRITERS`	`MS_$R or MS_$RX or MS_$RIW`
`Exclusive Write`	`MS_$NR_XOR_1W`	`MS_$WR or MS_$WRX`
`Shared Write`	`MS_$COWRITERS`	`MS_$WR or MS`

Once you have locked a file, the MS manager allows other processes to map the file only if these processes request a lock that is compatible with your lock. The following lock combinations are allowed and prohibited. (Y means that the combination is allowed; N means that the combination is prohibited.)

Existing Lock	*Requested Lock*				
	Protected Read	*Protected RIW*	*Shared Read*	*Exclusive Write*	*Shared Write*
Protected Read	Y	Y	Y	N	N
Protected RIW	Y	N	Y	N	N
Shared Read	Y	Y	Y	N	Y *
Exclusive Write	N	N	N	N	N
Shared Write	N	N	Y *	N	Y *

* These locks are allowed only if the processes are on the same node.

MS_$MAPL_STREAM

Maps the specified filesystem object, given its xoid, into an available region of the process address space. This call also locks the object and protects the mapping on a UNIX EXEC call. For type managers only.

FORMAT

```
address-ptr := MS_$MAPL_STREAM (xoid, start, desired-length, concurrency,
                                 access, extend, length-mapped, status)
```

RETURN VALUE

address-ptr

Pointer to the first mapped byte of the object, in UNIV_PTR format. This is a 4-byte integer.

The first mapped byte is not necessarily the first byte of the object; it is the byte you specify in the start parameter.

INPUT PARAMETERS

xoid

Xoid, or unique identifier of an object in XOID_$T format. This data type is 16-bytes long. See the MS Data Types section for details.

start

First byte to be mapped. This is a 4-byte positive integer. To specify the first byte of an object, provide a start value of 0.

desired-length

Number of bytes to map, including the start byte. This is a 4-byte positive integer.

concurrency

Concurrency mode for the object, in MS_$CONC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$NR_XOR_1W	Allows one writer or any number of readers.
MS_$COWRITERS	Allows any number of readers and/or writers.

access

The access mode desired, in MS_$ACC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$R	Read access.
MS_$RX	Read and execute access.
MS_$WR	Read and write access.
MS_$WRX	Write and execute access.
MS_$RIW	Read with intent to write.

The access requested must be a subset of the access permitted by the protection for the object.

extend

A Boolean value that indicates whether the object can be extended. The value TRUE indicates that the length given in the desired-length parameter should be mapped, even if the object is shorter. Writing beyond the end of the object, but within the space mapped, extends the object. FALSE indicates that the amount mapped should be no greater than the actual length of the file.

OUTPUT PARAMETERS

length-mapped

Number of bytes actually mapped. This is a 4-byte positive integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Object created.
MS_$BAD_ACCESS	Access type is illegal.
MS_$IN_USE	Object could not be locked.

USAGE

Use MS_$MAPL_STREAM to map objects that you access through a type manager. Note that you use MS_$MAPL_STREAM *only* through a type manager. For details, see the *Extending the DOMAIN Streams Facility* manual.

MS_$MAPL_STREAM protects the mapping on a UNIX EXEC call. Conversely, with MS_$MAPL, the UNIX EXEC call unmaps any objects on any open streams.

MS_$MAPL_STREAM also locks the object. The lock is determined by the concurrency and access modes that you specify. MS_$MAPL_STREAM can obtain the following types of locks:

Lock	*Concurrency Mode*	*Access Mode*
Protected Read	MS_$NR_XOR_1W	MS_$R or MS_$RX
Protected RIW	MS_$NR_XOR_1W	MS_$RIW
Shared Read	MS_$COWRITERS	MS_$R or MS_$RX or MS_$RIW
Exclusive Write	MS_$NR_XOR_1W	MS_$WR or MS_$WRX
Shared Write	MS_$COWRITERS	MS_$WR or MS

Once you have locked an object, the MS manager allows other processes to map the object only if these processes request a lock that is compatible with your lock. The following lock combinations are allowed and prohibited. (Y means that the combination is allowed; N means that the combination is prohibited.)

Existing Lock	*Requested Lock*				
	Protected Read	*Protected RIW*	*Shared Read*	*Exclusive Write*	*Shared Write*
Protected Read	Y	Y	Y	N	N
Protected RIW	Y	N	Y	N	N
Shared Read	Y	Y	Y	N	Y *
Exclusive Write	N	N	N	N	N
Shared Write	N	N	Y *	N	Y *

* These locks are allowed only if the processes are on the same node.

MS_$RELOCK

Changes the lock on an object.

FORMAT

```
MS_$RELOCK (virtual-address, access, status)
```

INPUT PARAMETERS

virtual-address

Pointer to the first mapped byte of the object whose lock you want to change, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by an earlier call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

access

New access mode, in MS_$ACC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$R	Read access.
MS_$WR	Read and write access.
MS_$RIW	Read with intent to write.

If you specify an access mode of MS_$RIW when you first lock an object, you cannot relock the object with MS_$R access.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Completed successfully.
MS_$NOT_MAPPED	No object is mapped at the supplied virtual address.
MS_$BAD_ACCESS	Access mode given is incorrect.
FILE_$ILLEGAL_LOCK_RQST	Illegal lock request (file server); the access mode given is incorrect.

USAGE

MS_$RELOCK changes the lock on an object. With MS_$RELOCK, you specify a new access type. This new access, in combination with the current concurrency mode, forms a new lock. You can relock a file in the following ways:

Current Lock	*Changes*
Protected read	Change to exclusive write by specifying the access mode MS_$WR or MS_$WRX.
	Change to protected RIW by specifying the access mode MS_$RIW.
Protected RIW	Change to exclusive write by specifying the access mode MS_$WR or MS_$WRX.
	Cannot change to protected read by specifying the access mode MS_$R.
Shared read	Change to shared write by specifying the access mode MS_$WR or MS_$WRX.
Exclusive write	Change to protected read by specifying the access mode MS_$R.
	Change to protected RIW by specifying the access mode MS_$RIW.
Shared write	Change to shared read by specifying MS_$R or MS_$RIW.

See the description of MS_$MAPL for a list of the concurrency/access combinations for each lock.

MS_$REMAP

Maps a different portion of a previously mapped object.

FORMAT

```
address  =  MS_$REMAP (old-address, start, desired-length,
                       remapped-length, status)
```

RETURN VALUE

address
Pointer to the first byte of the new mapped section, in UNIV_PTR format. This is a 4-byte integer.

INPUT PARAMETERS

old-address
Pointer to the first byte of the currently mapped portion of the object, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

start
First byte to be mapped. This is a 4-byte integer.

desired-length
Number of bytes to remap. This is a 4-byte integer.

OUTPUT PARAMETERS

remapped-length
Number of bytes remapped. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Completed successfully.
MS_$NOT_MAPPED	No object is mapped at the given virtual address.
MS_$BAD_LENGTH	Desired-length is invalid.

USAGE

This call maps a different portion of an already mapped object and unmaps the previously mapped portion. This call is useful for moving a sliding window over a big file.

When you remap a file, certain attributes of the mapping (extend, access, concurrency) are left the same as in the original mapping. If you used MS_$ADVICE to provide file access advice, the advice in effect for the first part of the currently mapped section is propagated to the newly mapped section. Also, MS_$REMAP does not change the lock mode of the object.

MS_$TRUNCATE

Truncates a mapped object to the specified length.

FORMAT

```
MS_$TRUNCATE (address, length, status)
```

INPUT PARAMETERS

address
Pointer to the first byte of the currently mapped portion of the object, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

length
Number of bytes to keep in the mapped object, starting at the first byte in the object. This is a 4-byte integer. Everything after this length is truncated.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information.

USAGE

MS_$TRUNCATE shortens a mapped file to the length that you specify. In addition, you can use MS_$TRUNCATE to define a length for a file, even if you are not throwing away data. For example, when you unmap a file, the system may set the file length to a page-aligned value. (That is, the length will be a multiple of 1024.) However, you can use MS_$TRUNCATE to shorten the file to a nonpage-aligned value.

For example, if a file contains only 20 bytes of data, you can use MS_$TRUNCATE to set the file length to 20. When you unmap the file, the length will be 20 rather than 1024. Use MS_$ATTRIBUTES to determine the current file length and number of blocks used.

MS_$UNMAP

Unmaps a previously mapped object.

FORMAT

```
MS_$UNMAP (address, length, status)
```

INPUT PARAMETERS

address
Pointer to the first byte of the currently mapped portion of the object, in UNIV_PTR format. This is a 4-byte integer. Use the pointer returned by the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

length
Number of mapped bytes. This is a 4-byte integer. Use the length you requested in the most recent call to MS_$MAPL, MS_$CRMAPL, or MS_$REMAP.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MS Data Types section for more information. Possible values are:

STATUS_$OK	Completed successfully.
MS_$NOT_MAPPED	Address and length given do not refer to an object mapped with MS_$MAPL.

USAGE

MS_$UNMAP unmaps and unlocks an object mapped and locked with MS_$MAPL or MS_$CRMAPL. You cannot unmap a subset of the object.

If the original object is on a remote node, changes made in the mapped version are written back to the original object when MS_$UNMAP is executed. If the original object is on the local node, changes made in the mapped version of the object will be written back to the original object when the space they occupy in memory is needed.

If the object was mapped with the extend parameter equal to TRUE, and your program modified part of the extension space, the original object is extended to include those modifications. Parts of the extension space beyond the last modification are not added to the original object.

An object locked by several calls to MS_$MAPL by different processes will remain locked until all the processes have unmapped the object.

ERRORS

MS_$BAD_ACCESS
: Unsupported access rights requested.

MS_$BAD_LENGTH
: Bad length.

MS_$IN_USE
: Object is locked by another process or in an incompatible mode.

MS_$INSUFFICIENT_RIGHTS
: You have some access rights to the object, but not the ones you requested.

NAME_$NAME_NOT_FOUND
: No object exists with the given name.

MS_$NO_RIGHTS
: You do not have any access rights to the object.

MS_$NO_SPACE
: No space.

MS_$NOT_MAPPED
: No object mapped at the virtual address supplied.

MS_$OBJECT_NOT_FOUND
: The object does not exist, or it is not accessible over the network.

MS_$WRONG_LEVEL
: Attempt to release segment mapped by previous level.

STATUS_$OK
: Successful completion.

Other naming server errors.
: See the NAME_$ error codes.

MTS

This section describes the data types, the call syntax, and the error codes for the MTS programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

MTS_$CURRENT_FILE_SEQ	File sequence number for current file.
MTS_$END_FILE_SEQ	File sequence number for last tape file.
MTS_$FIRST_A	MTS_$UNIT_A First attribute in MTS_$ATTR_T.
MTS_$LAST_A	MTS_$BUFFER_OFFSET_A Last attribute in MTS_$ATTR_T.

DATA TYPES

MTS_$ATTR_T

A 2-byte integer. THE User-modifiable tape file attributes. One of the following pre-defined values:

MTS_$UNIT_A
Tape unit number.

MTS_$LABELED_A
Labeled volume.

MTS_$REOPEN_VOL_A
Reopen volume.

MTS_$CLOSE_VOL_A
Close file and volume.

MTS_$SAVE_VOL_POS_A
Save position on close.

MTS_$VOL_DEVICE_A
Device type.

MTS_$VOL_ID_A
Volume ID.

MTS_$VOL_ACCESS_A
Volume accessibility.

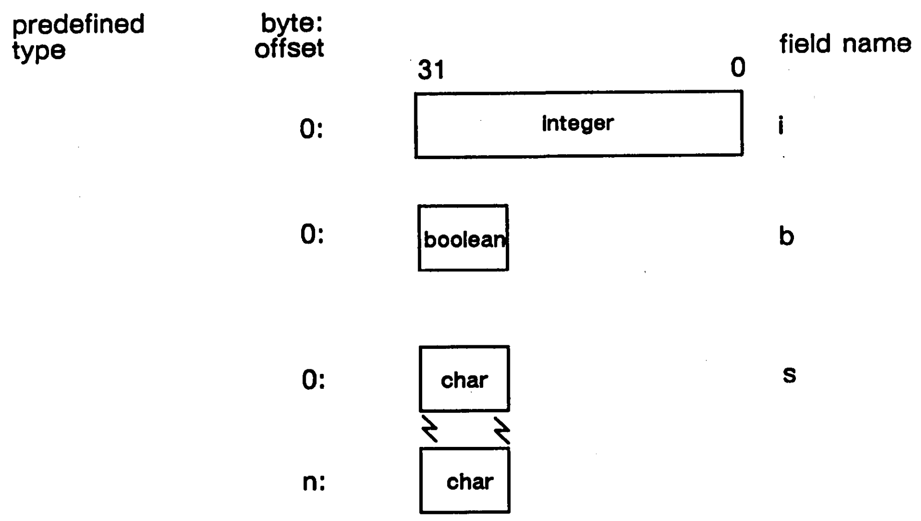
MTS_$OWNER_ID_A
Owner ID.

MTS_$FILE_SEQUENCE_A
File sequence number.

MTS_$RECORD_FORMAT_A
Record format.

MTS_$BLOCK_LENGTH_A
Block length.

	MTS_$RECORD_LENGTH_A Maximum record length.
	MTS_$ASCII_NL_A ASCII newline head-length
	MTS_$FILE_SECTION_A File section number.
	MTS_$FILE_ID_A File ID.
	MTS_$FILE_SET_ID_A File set ID.
	MTS_$GENERATION_A Generation number.
	MTS_$GENERATION_VERSION_A Generation version number.
	MTS_$CREATE_DATE_A Creation date.
	MTS_$EXP_DATE_A Expiration date.
	MTS_$FILE_ACCESS_A File accessibility.
	MTS_$SYSTEM_CODE_A System code.
	MTS_$SYSTEM_USE_A System use.
	MTS_$BUFFER_OFFSET_A Buffer offset.
MTS_$ATTR_VALUE_T	Attribute values. The diagram below illustrates the MTS_$ATTR_VALUE_T data type:



Field Description:

i
An integer value.

b
A Boolean value.

s
A character string.

MTS_$DEVICE_T

A 2-byte integer. Type of device. One of the following pre-defined values:

MTS_$MT
Magtape device.

MTS_$NOT_REALLY
Not currently supported.

MTS_$CT
Cartridge tape device.

MTS_$HANDLE_T

A 4-byte integer. A handle to a tape descriptor file.

MTS_$RW_T

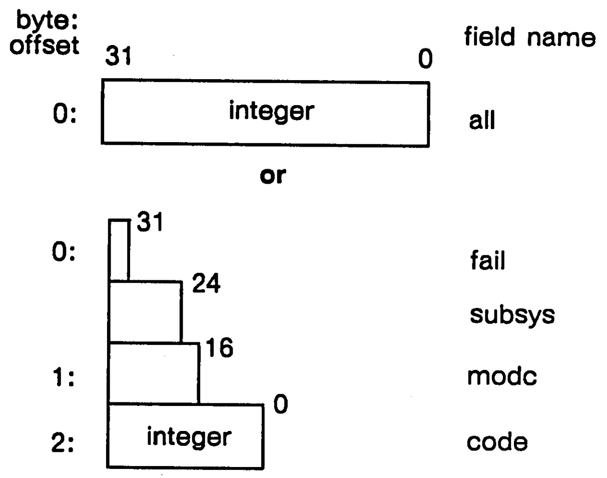
A 2-byte integer. Read or write status. One of the following pre-defined values:

MTS_$READ
Read operation.

MTS_$WRITE
Write operation.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

Table MTS-1. Magnetic Tape Volume and File Attributes

Mnemonic	*Type*	*Default*	*Definition*
MTS_$UNIT_A	int	0	magtape unit number (normally 0)

MTS_$LABELED_A	t/f	true	true = ANSI labeled volume
			false = unlabeled volume

MTS_$REOPEN_VOL_A	t/f	false	true = reopen previously used volume (suppresses rewind) false = do not reopen
MTS_$CLOSE_VOL_A	t/f	true	true = volume closed when file is closed false = leave volume open
MTS_$SAVE_VOL_POS_A	t/f	false	true = saves volume position when volume is closed (for reopen) false = rewind volume when closed

MTS_$VOL_COMMENT_A	int	0	type of device:
			tfp_$mt=0 for magtape
			tfp_$ct=3 for cartridge

MTS_$VOL_ID_A	char	-auto	volume identifier (labeled volumes) (Automatically generated.) Six-character string maximum.
MTS_$VOL_ACCESS_A	char	" "	volume accessibility (labeled volumes only). The default is the space character.
MTS_$VOL_OWNER_ID_A	char	-auto	volume owner (labeled volumes) Maximum string length is 14.
MTS_$FILE_SEQUENCE_A	int "cur" "end"	1	file sequence number. Possible values are an integer, "cur" for current file, "end" for new file at end of labeled volume.
MTS_$RECORD_FORMAT_A	char	D	record format. Possible values: "F" = fixed length "D" = variable length "S" = spanned "U" = undefined
MTS_$BLOCK_LENGTH_A	int	2048	block length, in bytes
MTS_$RECORD_LENGTH_A	int	2048	maximum record length, in bytes

Table MTS-1. Magnetic Tape Volume and File Attributes (Continued)

Mnemonic	*Type*	*Default*	*Definition*
MTS_$ASCII_NL_A	t/f	true	true = ASCII newline handling. Strip newlines on write, supply them on read false = no newline handling
MTS_$FILE_SECTION_A	int	1	file section number (labeled volumes)
MTS_$FILE_ID_A	char	" "	file identifier (labeled volumes)
MTS_$FILE_SET_ID_A	char	" "	file set identifier (labeled volumes)
MTS_$GENERATION_A	int	1	generation of file (labeled volumes)
MTS_$GENERATION_VERSION_A	int	1	generation version of file (labeled volumes)
MTS_$CREATE_DATE_A	date	-auto	creation date of file (labeled volumes)
MTS_$EXP_DATE_A	date	-auto	expiration date of file (labeled volumes)
MTS_$FILE_ACCESS_A	char	" "	file accessibility (labeled volumes)
MTS_$SYSTEM_CODE_A	char	" "	system code (labeled volumes)
MTS_$SYSTEM_USE_A	char	" "	system use (labeled volumes)
MTS_$BUFFER_OFFSET_A	int	0	buffer offset (labeled volumes) Must be zero.

MTS_$CLOSE_DESC

Closes a magtape descriptor file.

FORMAT

```
MTS_$CLOSE_DESC (handle, update, status)
```

INPUT PARAMETERS

handle
Pointer to the open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer. Specify the handle returned by MTS_$OPEN_DESC, MTS_$COPY_DESC, or MTS_$CREATE_DESC.

update
Boolean value that determines whether or not the magtape descriptor file is to be modified to reflect the attribute changes specified by calls to MTS_$SET_ATTR. If TRUE, the changes are made.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

Programs must close magtape descriptor files before calling stream manager routines; an open magtape descriptor file cannot be used by the stream manager.

Closing a magtape descriptor file invalidates its handle.

MTS_$COPY_DESC

Copies a magtape descriptor file and opens the destination file.

FORMAT

```
handle = MTS_$COPY_DESC (src-pathname, src-namelen, dst-pathname,
                         dst-namelen, status)
```

RETURN VALUE

handle

Pointer to the open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer.

INPUT PARAMETERS

src-pathname

The pathname of the magtape descriptor file to be copied, in NAME_$PNAME_T format. This is an array of up to 256 characters.

src-namelen

Length of the source pathname, in bytes. This is a 2-byte integer.

dst-pathname

The pathname to which the file is to be copied, in NAME_$PNAME_T format. This is an array of up to 256 characters.

The destination file must not exist before this function is called. To replace a destination file, call the routine NAME_$DELETE_FILE before calling MTS_$COPY_DESC.

dst-namelen

Length of the destination pathname, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

This routine copies the specified magtape descriptor file, opens the destination file and returns a pointer to it.

This routine does not affect the source file.

MTS_$CREATE_DEFAULT_DESC

Creates a magtape descriptor file with the default volume and file attributes.

FORMAT

```
handle = MTS_$CREATE_DEFAULT_DESC (pathname, namelen, status)
```

RETURN VALUE

handle
A pointer to the open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer.

INPUT PARAMETERS

pathname
The pathname of the descriptor file to be created, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelen
Length of the name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

This routine opens a magtape descriptor file with the default volume and file attribute values and returns a pointer to it. The file must not exist before this routine is called. See the Table in the MTS Data Types section for a list of volume and file attributes and their defaults.

MTS_$GET_ATTR

Retrieves a given attribute from a magtape descriptor file.

FORMAT

```
MTS_$GET_ATTR (handle, attribute, value, status)
```

INPUT PARAMETERS

handle

A pointer to the open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer. Specify a handle returned by MTS_$OPEN_DESC, MTS_$COPY_DESC, or MTS_$CREATE_DESC.

attribute

The attribute to be retrieved, in MTS_$ATTR_T format. This is a 2-byte integer. Specify only one of the following predefined values:

```
mts_$unit_a            mts_$labeled_a              mts_$reopen_vol_a
mts_$close_vol_a       mts_$save_vol_pos_a         mts_$vol_device_a
mts_$vol_id_a          mts_$vol_access_a           mts_$owner_id_a
mts_$file_sequence_a   mts_$record_format_a        mts_$block_length_a
mts_$record_length_a   mts_$ascii_nl_a             mts_$file_resv1_a
mts_$file_section_a    mts_$file_id_a              mts_$file_set_id_a
mts_$generation_a      mts_$generation_version_a   mts_$create_date_a
mts_$exp_date_a        mts_$file_access_a          mts_$system_code_a
mts_$system_use_a      mts_$buffer_offset_a
```

See the Table in the MTS Data Types section for a description of volume and file attributes and their defaults.

OUTPUT PARAMETERS

value

The current value of the specified attribute, in MTS_$ATTR_VALUE_T format. Possible values are a 4-byte integer, a Boolean value, or a string, depending upon the attribute requested. See the Table in the MTS Data Types section for a list of volume and file attributes and their corresponding values.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

Programs must call this routine once for each attribute they wish to get.

You can change the attributes within a magtape descriptor file using the MTS_$SET_ATTR system call.

MTS_$LABEL

Labels the magtape volume described by the given magtape descriptor file.

FORMAT

```
MTS_$LABEL (pathname, namelen, status)
```

INPUT PARAMETERS

pathname
The pathname of the magtape descriptor file, in NAME_$PNAME_T format. This is an array of up to 256 characters.

The descriptor file must describe a labeled volume.

namelen
Length of the name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

MTS_$LABEL causes the volume described by the descriptor file to be labeled according to ANSI x3.27-1978.

The tape volume must not be open (by previous calls to the stream manager).

MTS_$OPEN_DESC

Opens the specified magtape descriptor file and returns a pointer to it.

FORMAT

```
handle = MTS_$OPEN_DESC (pathname, namelen, read-write, status)
```

RETURN VALUE

handle
A pointer to the open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer.

INPUT PARAMETERS

pathname
The pathname of the magtape descriptor file, in NAME_$PNAME_T format. This is an array of up to 256 characters.

namelen
Length of the name, in bytes. This is a 2-byte integer.

read-write
Read or write status, in MTS_$RW_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MTS_$READ	Open for reading.
MTS_$WRITE	Open for writing.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

MTS_$OPEN_DESC opens the specified magtape descriptor file for reading or writing and returns a pointer to it.

MTS_$SET_ATTR

Sets an attribute within the specified magtape descriptor file.

FORMAT

```
MTS_$SET_ATTR (handle, attribute, value, status)
```

INPUT PARAMETERS

handle

A pointer to an open magtape descriptor file, in MTS_$HANDLE_T format. This is a 4-byte integer. Specify a handle returned by MTS_$OPEN_DESC, MTS_$COPY_DESC, or MTS_$CREATE_DESC.

attribute

The volume or file attribute to be set, in MTS_$ATTR_T format. This is a 2-byte integer. Specify only one of the following predefined values:

```
mts_$unit_a             mts_$labeled_a              mts_$reopen_vol_a
mts_$close_vol_a        mts_$save_vol_pos_a         mts_$vol_device_a
mts_$vol_id_a           mts_$vol_access_a           mts_$owner_id_a
mts_$file_sequence_a    mts_$record_format_a        mts_$block_length_a
mts_$record_length_a    mts_$ascii_nl_a             mts_$file_resvl_a
mts_$file_section_a     mts_$file_id_a              mts_$file_set_id_a
mts_$generation_a       mts_$generation_version_a   mts_$create_date_a
mts_$exp_date_a         mts_$file_access_a          mts_$system_code_a
mts_$system_use_a       mts_$buffer_offset_a
```

See the Table in the MTS Data Types section for a description of volume and file attributes and their defaults.

value

The value to assign to the attribute, in MTS_$ATTR_VALUE_T format. Possible values are a 4-byte integer, a Boolean value, or a string, depending upon the attribute to be changed. See the Table in the MTS Data Types section for a list of volume and file attributes and their corresponding values.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the MTS Data Types section for more information.

USAGE

Programs must call this routine once for each attribute to be set.

You can change the attributes within a magtape descriptor file using the Shell command EDMTDESC. See the *DOMAIN System Command Reference* for details.

ERRORS

STATUS_$OK
Successful completion.

MTS_$BAD_BLOCK_LENGTH
Bad block length.

MTS_$BAD_BUFFER_OFFSET
Bad buffer offset.

MTS_$BAD_DATA
Descriptor contains bad data.

MTS_$BAD_FILE_SECTION
Bad file section number.

MTS_$BAD_FILE_SEQUENCE
Bad file sequence number.

MTS_$BAD_GENERATION
Bad generation number.

MTS_$BAD_GENERATION_VERSION
Bad generation version number.

MTS_$BAD_RECORD_FORMAT
Bad record format attribute.

MTS_$BAD_RECORD_LENGTH
Bad record length.

MTS_$BAD_UNIT
Bad tape unit number.

MTS_$INVALID_ATTR
Invalid attribute to GET_ATTR/SET_ATTR.

MTS_$INVALID_DATE
Invalid date text string.

MTS_$NOT_LABELED
Attempt to label unlabeled volume.

MTS_$READ_ONLY
SET_ATTR on read-only file.

MTS_$VOL_IN_USE
Volume in use.

MTS_$WRONG_TYPE
Object is not type MT_$UID.

MUTEX

This section describes the data types and the call syntax for the MUTEX programming calls. The MUTEX calls do not produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

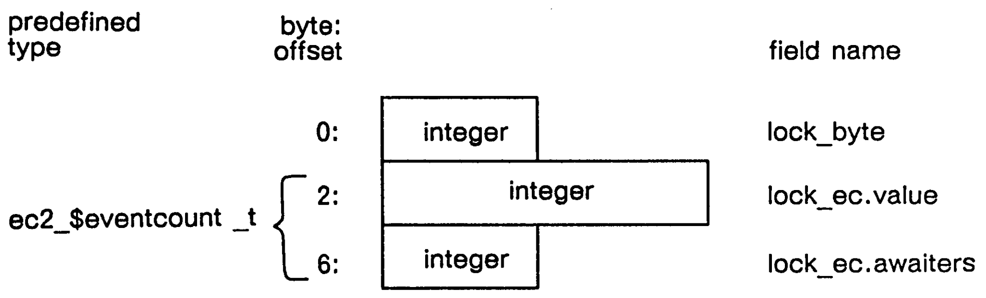
CONSTANTS

MUTEX_$WAIT_FOREVER	integer32(-1) A value that tells MUTEX_$LOCK to wait forever without timing out.

DATA TYPES

MUTEX_$LOCK_REC_T

A mutual exclusion lock record. The diagram below illustrates the MUTEX_$LOCK_REC_T data type:



Field Description:

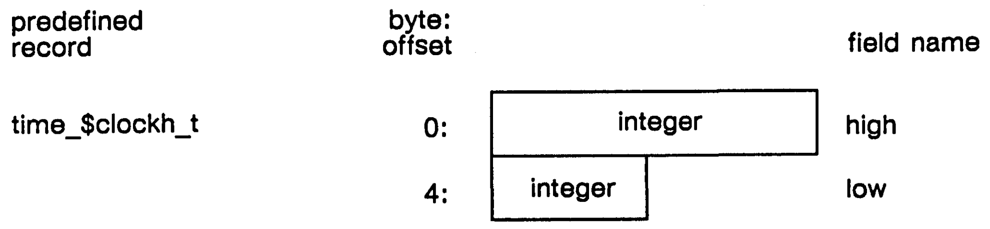
lock_byte
A Boolean value that indicates whether any programs currently hold a MUTEX lock.

lock_ec
An eventcount for programs waiting for the MUTEX lock. The lock_ec field is in EC2_$EVENTCOUNT format and has two subfields:

lock_ec.value	The value of the eventcount.
lock_ec.awaiters	Used internally by the EC2 manager.

TIME_$CLOCK_T

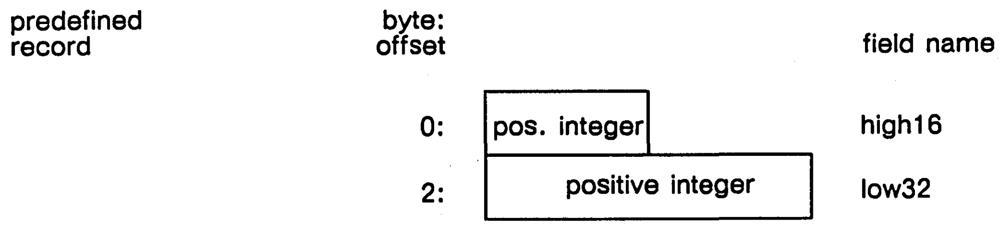
Internal representation of time. The diagram below illustrates the TIME_$CLOCK_T data type:



Field Description:

high
High 32 bits of the clock.

low
Low 16 bits of the clock.



Field Description:

high16
High 16 bits of the clock.

low32
Low 32 bits of the clock.

MUTEX_$INIT

Initializes a mutual exclusion lock record.

FORMAT

```
MUTEX_$INIT (lock-record)
```

OUTPUT PARAMETERS

lock-record

Lock record, in MUTEX_$LOCK_REC_T format. This data type is 8 bytes long. See the MUTEX Data Types section for more information.

USAGE

Use this call to initialize a mutual exclusion (MUTEX) lock record. This lock record allows a program to obtain a MUTEX lock on a file. A MUTEX lock allows a program to get exclusive access to a shared resource.

Initialize a MUTEX lock record within a file. First, map the file with a concurrency mode of MS_$COWRITERS and an access type of MS_$WR. Then use MUTEX_$INIT to initialize the MUTEX lock record.

MUTEX_$LOCK

Obtains a mutual exclusion lock on a file.

FORMAT

```
lock-status = MUTEX_$LOCK (lock-record, wait-time)
```

RETURN VALUE

lock-status

A Boolean value that indicates whether you obtained the lock. TRUE means that you got the lock; FALSE means that the call timed out before obtaining the lock.

INPUT/OUTPUT PARAMETERS

lock-record

Lock record, in MUTEX_$LOCK_REC_T format. This data type is 8 bytes long. See the MUTEX Data Types section for more information.

INPUT PARAMETERS

wait-time

The amount of time to wait for the lock, in TIME_$CLOCK_T format. This data type is 6 bytes long. See the MUTEX Data Types section for more information.

If MUTEX_$LOCK cannot obtain the lock within the time you specify, the call will time out and return control to your program. Specify the waiting time as a relative time. Use the CAL routines to convert time values to TIME_$CLOCK_T format.

If you specify the waiting time using the constant MUTEX_$WAIT_FOREVER, the MUTEX_$LOCK call wait indefinitely to obtain the lock.

USAGE

Use MUTEX_$LOCK to obtain a mutual exclusion (MUTEX) lock on a file. A MUTEX lock lets you have exclusive access to a shared resource.

MUTEX_$LOCK uses the information in a lock record to determine whether you can obtain the lock. (Use MUTEX_$INIT to initialize a lock record.) If another program already has the lock, MUTEX_$LOCK waits for the amount of time you specify. When MUTEX_$LOCK returns, it indicates whether you obtained the lock.

Before calling MUTEX_$LOCK, you must map the file containing the lock record. Map the file with a concurrency mode of MS_$COWRITERS and an access mode of MS_$WR. All programs that map the same MUTEX lock record must be on the same node.

Note that a MUTEX lock is a convention that cooperating programs use to control access to a resource. If a program does not use MUTEX_$LOCK and accesses the resource directly, you cannot guarantee mutual exclusion.

MUTEX_$UNLOCK

Terminates a program's mutual exclusion lock on a file.

FORMAT

```
MUTEX_$UNLOCK (lock-record)
```

INPUT/OUTPUT PARAMETERS

lock-record

Lock record, in MUTEX_$LOCK_REC_T format. This data type is 8 bytes long. See the MUTEX Data Types section for more information.

USAGE

MUTEX_$UNLOCK terminates a program's mutual exclusion lock on a file. A waiting program can then obtain the lock.

NAME

This section describes the data types, the call syntax, and the error codes for the NAME programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

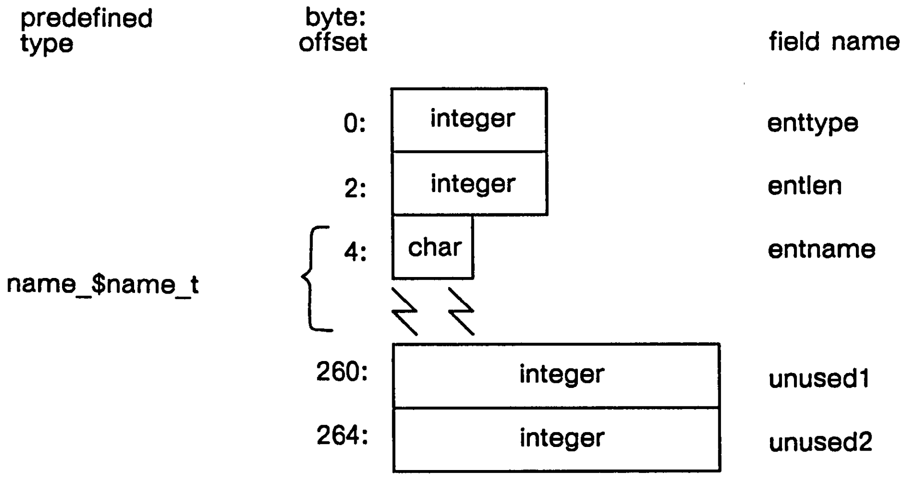
CONSTANTS

NAME_$COMPLEN_MAX	32	Maximum length of an entry name.
NAME_$FILE	1	The file type value for the enttype field of the DIR_ENTRY_T record.
NAME_$LINK	3	The link type value for the enttype field of the DIR_ENTRY_T record.
NAME_$PNAMLEN_MAX	256	Maximum length of a pathname.

DATA TYPES

NAME_$DIR_ENTRY_T

The directory entry returned by NAME_$READ_DIR. The diagram below illustrates the NAME_$DIR_ENTRY_T data type:



Field Description:

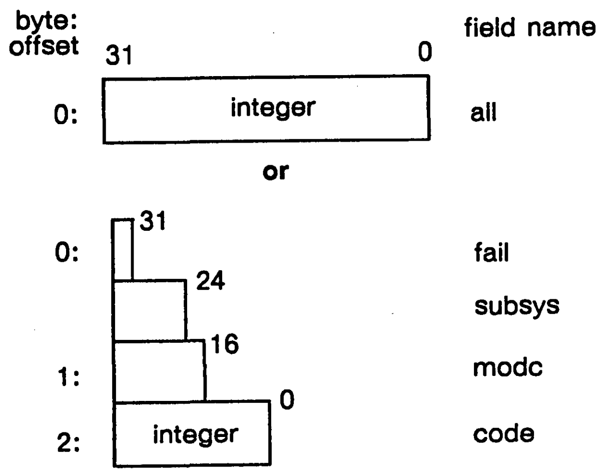
enttype
Type of the directory entry. Either NAME_$FILE or NAME_$LINK.

entlen
Length of the directory entry name.

entname
Name of the directory entry.

unusedn
Reserved for future use by Apollo.

NAME_$DIR_LIST_T	A 1300-element array of NAME_$DIR_ENTRY_T record structures. The diagram below illustrates a single element:
NAME_$NAME_T	An array of up to NAME_$COMPLEN_MAX (32) characters.
NAME_$PNAME_T	An array of up to NAME_$PNAMLEN_MAX (256) characters.
STATUS_$T	A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

NAME_$ADD_LINK

Creates a link.

FORMAT

```
NAME_$ADD_LINK (linkname, name-length, link-text, text-length, status)
```

INPUT PARAMETERS

linkname
Name of the link, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the object is placed in the entry directory of the local node.

name-length
Length of the linkname, in bytes. This is a 2-byte integer.

link-text
Pathname to which the link refers, in NAME_$PNAME_T format. This is an array of up to 256 characters.

The link text replaces the linkname when the linkname is used as part of a pathname. For example, suppose a link named YEATS had a link text //MAN/IN/MASK. Using the object name YEATS is exactly equivalent to using the pathname //MAN/IN/MASK directly.

The link text must be a valid filename or pathname. It does not, however, have to refer to an existing object.

text-length
Length of the link-text pathname, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

A link is an object within a directory that points to another object. That is, the link is associated with a pathname that refers to another object. The associated pathname is refered to as the link text. When the link is referenced, the naming server acts as if the link text were given in place of the link name.

To delete a link, you must use the naming server call NAME_$DROP_LINK, or the Shell command DELETE_LINK (DLL).

This system call corresponds to the CRL Shell command.

NAME_$CNAME

Changes the last element of a pathname.

FORMAT

```
NAME_$CNAME (old-pathname, old-length, new-leaf, leaf-length, status)
```

INPUT PARAMETERS

old-pathname
The current pathname, in NAME_$PNAME_T format. This is an array of up to 256 characters.

old-length
The length of the current pathname, in bytes. This is a 2-byte integer.

new-leaf
The name that replaces the right-most element of the current pathname, in NAME_$NAME_T format. This is an array of up to 256 characters.

leaf-length
The length of the new-leaf name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$CNAME changes the right-most element of the old-pathname to the string specified by the new-leaf argument.

NAME_$CREATE_DIRECTORY

Creates a directory.

FORMAT

```
NAME_$CREATE_DIRECTORY (directory-name, name-length, status)
```

INPUT PARAMETERS

directory-name

Name of the directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the object is placed in the entry directory of the local node.

name-length

Length of directory name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$CREATE_DIRECTORY creates a directory using the specified pathname and name length.

This system call corresponds to the CRD Shell command.

NAME_$CREATE_FILE

Creates a permanent file.

FORMAT

```
NAME_$CREATE_FILE (filename, name-length, status)
```

INPUT PARAMETERS

filename
Name of the file, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name-length
Length of the filename, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

The filename given is treated in the same way as any pathname given to the naming server. For example, a filename beginning with a slash (/) is placed in the entry directory of the local node.

NAME_$DELETE_DIRECTORY

Deletes a directory.

FORMAT

```
NAME_$DELETE_DIRECTORY (directory-name, name-length, status)
```

INPUT PARAMETERS

directory-name

Name of the directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the object is placed in the entry directory of the local node.

name-length

Length of the name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$DELETE_DIRECTORY deletes the specified directory. The directory must be empty for a deletion to succeed.

NAME_$DELETE_FILE

Deletes a file.

FORMAT

```
NAME_$DELETE_FILE (filename, name-length, status)
```

INPUT PARAMETERS

filename

Name of the file, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the object is placed in the entry directory of the local node.

name-length

Length of the filename, in bytes. this is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$DELETE_FILE deletes the specified file.

This system call corresponds to the DLF Shell command.

NAME_$DROP_LINK

Deletes a link.

FORMAT

```
NAME_$DROP_LINK (linkname, name-length, status)
```

INPUT PARAMETERS

linkname
Name of the link, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the object is placed in the entry directory of the local node.

name-length
Length of the link name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$DROP_LINK deletes the specified link to an associated object.

This system call corresponds to the DLL Shell command.

NAME_$EXTRACT_DATA

Extracts data from a directory entry read by NAME_$READ_DIR. (Intended primarily for use in FORTRAN programs.)

FORMAT

```
NAME_$EXTRACT_DATA (dir-entry, entry-type, entry-length, entry-name)
```

INPUT PARAMETERS

dir-entry

The directory entry for which you wish to extract data, in NAME_$ENTRY_T format. This data type is 44 bytes long. See the NAME Data Types section for more information.

In FORTRAN programs, NAME_$READ_DIR returns the directory entries in a (22,n) INTEGER*2 array, where n is the maximum number of directory entries your program is prepared to accept. Each column in this array corresponds to an entry in the specified directory and contains information about that entry.

Specify the first element of the column that corresponds to the entry for which you wish to extract data.

OUTPUT PARAMETERS

entry-type

Object type of the entry. This is a 2-byte integer with one of the following predefined values:

1 - NAME_$FILE
the object is a file. A "file" can be either a streams file or a directory.

2 - NAME_$LINK
the object is a link.

entry-length

Length of the object's name, in bytes. This is a 2-byte integer.

entry-name

The entry name, in NAME_$NAME_T format. This is an array of up to 32 characters.

USAGE

This call extracts the description of a single directory entry from the directory entry array (the dir-list parameter) returned by NAME_$READ_DIR. It is intended primarily for use in FORTRAN programs.

In FORTRAN programs, NAME_$READ_DIR returns the directory entries in a (22,n) INTEGER*2 array, where n is the maximum number of directory entries your program is prepared to accept. Each column in this array corresponds to an entry in the specified directory and contains information about that entry.

The dir-entry parameter for NAME_$EXTRACT_DATA should be one of the array columns. To reference a single column, give the first element of that column.

NAME_$GET_NDIR

Returns the full pathname of the naming directory.

FORMAT

```
NAME_$GET_NDIR (name, name-length, status)
```

OUTPUT PARAMETERS

name
Pathname of the naming directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name-length
Length of the name, in bytes. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

The naming directory is set through the NAME_$SET_NDIR call or the "ND directory-name" Shell command. This system call corresponds to the ND Shell command.

NAME_$GET_PATH

Converts a partial pathname into a full pathname.

FORMAT

```
NAME_$GET_PATH (in-name, in-len, out-name, out-len, status)
```

INPUT PARAMETERS

in-name
The relative pathname of an object, in NAME_$PNAME_T format. This is an array of up to 256 characters.

in-len
Length of the relative pathname, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

out-name
The full (absolute) pathname of the object, in NAME_$PNAME_T format. This is an array of up to 256 characters.

out-len
Length of the relative pathname, in bytes. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$GET_PATH converts a partial pathname into a full pathname. For example, if you have been using file FOO, you can call NAME_$GET_PATH to find out that the full pathname of FOO is //FLYNN/PHL/FOO.

NAME_$GET_WDIR

Returns the full pathname of the working directory.

FORMAT

```
NAME_$GET_WDIR (name, name-length, status)
```

OUTPUT PARAMETERS

name
Pathname of the working directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name-length
Length of the name, in bytes. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

The working directory is set through the NAME_$SET_WDIR call or the "WD directory-name" Shell command.

NAME_$READ_DIR

Reads a directory.

FORMAT

```
NAME_$READ_DIR (dir-name, name-length, dir-list, index, max-count,
               read-count, status)
```

INPUT PARAMETERS

dir-name

Name of the directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the entry directory of the local node is searched for the directory.

Specifying a null character ('') defaults to the current working directory.

name-length

Length of the name, in bytes. This is a 2-byte integer.

If you specify a null character for the directory name, specify zero as the length.

OUTPUT PARAMETERS

dir-list

A list of directory entries, in NAME_$DIR_LIST_T format. This is an array of NAME_$DIR_ENTRY_T data types. See the NAME Data Types section for more information.

The number of NAME_$DIR_ENTRY_T data types in the array must equal or exceed max-count.

INPUT/OUTPUT PARAMETERS

index

Key indicating the directory entry at which to begin reading. This is a 4-byte integer.

On input	This number indicates the entry at which to begin reading.
On output	This number is adjusted by NAME_$READ_DIR to a number suitable for a subsequent call to NAME_$READ_DIR.

To read from the start of the directory, initialize the index to 1 on your first call to NAME_$READ_DIR.

Because NAME_$READ_DIR adjusts the index parameter to a suitable value for a subsequent call, you do not need to change the value yourself.

If after the first call:

Entries remain to be read
: the value of max-count is added to the index parameter, so that the next entry will be read on a subsequent call.

End of directory is encountered
: the index parameter is set to 0. A subsequent call returns a status of NAME_$NO_MORE_ENTRIES.

If max-count is identical to the number of directory entries remaining, the call to NAME_$READ_DIR does not reach the end of the directory and does not set the index parameter to 0. A subsequent call to NAME_$READ_DIR returns a status of NAME_$NO_MORE_ENTRIES.

INPUT PARAMETERS

max-count
Maximum number of directory entries to read. This is a 2-byte integer.

OUTPUT PARAMETERS

read-count
Number of directory entries actually read. This is a 2-byte integer.

If READ_DIR reaches the end of the directory before finding the requested number of entries, it returns a read-count smaller than your requested max-count

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

Possible values are:

STATUS_$OK Completed successfully.

NAME_$NO_MORE_ENTRIES
: All entries in the directory have been read.

USAGE

NAME_$READ_DIR reads a directory and stores entry names, the length of each entry name, and the type of each entry. Pascal and C programs can access this information directly through the directory entry record structure. FORTRAN programs use the NAME_$EXTRACT_DATA system call to access this information.

The index argument pemits a program to make several calls to NAME_$READ_DIR to ensure reading all entries. However, to get an accurate snapshot of a directory, make only one call to NAME_$READ_DIR, using a sufficiently large max-count, because the contents of a directory can change between calls to NAME_$READ_DIR if the directory is not locked.

NAME_$READ_LINK

Returns the link text associated with a link name.

FORMAT

```
NAME_$READ_LINK (linkname, name-length, link-text, text-length, status)
```

INPUT PARAMETERS

linkname
Name of the link, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. If a pathname is specified beginning with a slash (/), the entry directory of the local node is searched for the link.

name-length
Length of the linkname, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

link-text
Text associated with the linkname, in NAME_$PNAME_T format. This is an array of up to 256 characters.

text-length
Length of the text, in bytes. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

When you use a linkname, the naming server replaces the link name with the associated link text. NAME_$READ_LINK returns the text associated with a specified link name.

NAME_$SET_NDIR

Sets the naming directory.

FORMAT

```
NAME_$SET_NDIR (name, name-length, status)
```

INPUT PARAMETERS

name
Pathname of the desired naming directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. A directory name beginning with a period (.) indicates a directory within the working directory. You may also specify a period by itself, which sets the naming directory equal to the working directory.

name-length
Length of the pathname, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$SET_NDIR sets the naming directory to the specified directory. See the *DOMAIN System Command Reference* for a description of naming directories.

This system call corresponds to the "ND directory-name" Shell command.

NAME_$SET_WDIR

Sets the working directory.

FORMAT

```
NAME_$SET_WDIR (name, name-length, status)
```

INPUT PARAMETERS

name

Pathname of the desired working directory, in NAME_$PNAME_T format. This is an array of up to 256 characters.

Specify either an absolute or relative pathname. If a relative pathname is specified, the rest of the pathname defaults to the current working directory. A directory name beginning with a period (.) indicates a directory within the working directory.

name-length

Length of the name, in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the NAME Data Types section for more information.

USAGE

NAME_$SET_NDIR sets the naming directory to the specified directory. See the DOMAIN System Command Reference Manual for a description of working directories.

This system call corresponds to the "WD directory-name" shell command.

ERRORS

STATUS_$OK
Successful completion.

NAME_$ALREADY_EXISTS
Name already exists.

NAME_$BAD_DIRECTORY
Bad directory.

NAME_$BAD_LEAF
Invalid leaf.

NAME_$BAD_LINK
Invalid link.

NAME_$BAD_PATHNAME
Invalid pathname.

NAME_$DIRECTORY_FULL
Directory is full.

NAME_$DIRECTORY_NOT_EMPTY
Directory is not empty.

NAME_$FILE_NOT_DIRECTORY
Branch is not a directory.

NAME_$ILL_LINK_OP
Invalid link operation.

NAME_$INSUFFICIENT_RIGHTS
Insufficient rights.

NAME_$IS_SYSBOOT
Unable to delete system bootstrap (sysboot).

NAME_$NO_RIGHTS
No rights.

NAME_$NODE_UNAVAILABLE
Node is unavailable.

NAME_$NOT_FILE
Name is not a file.

NAME_$NOT_FOUND
Name not found.

NAME_$NOT_LINK
Name is not a link.

PAD

This section describes the data types, the call syntax, and the error codes for the PAD programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

MNEMONIC	Value	Explanation
PAD_$BS	8	Moves the cursor one character position to the left if there is any room in the window.
PAD_$CPR_ALL	2	Cursor position report: Reports on each raw keystroke.
PAD_$CPR_CHANGE	1	Cursor position report: Reports only the changed position since the last output call or position report.
PAD_$CPR_DRAW	4	Cursor position report: Reports on all touchpad data.
PAD_$CPR_FLAG	16#FF	Cursor position report: Indicates that the next 5 bytes is a report.
PAD_$CPR_NONE	0	Cursor position report: Does not report any cursor positions.
PAD_$CPR_PICK	3	Cursor position report: Reports after cursor is settled when it has been moved by the touchpad.
PAD_$CR	13	Returns cursor to the left edge of the pad at the same line it was on.
PAD_$ESCAPE	27	For control characters: Tells Display Manager not to interpret the next character as a control character. This precedes ANSI escape sequences.
PAD_$FF	12	Makes output start at the top of the window or window pane.
PAD_$LEFT_WINDOW	16#FD	Cursor position report: Indicates that the cursor accompanying the report is outside the window.
PAD_$MAX_TABSTOPS	100	Defines the maximum number of tabstops allowed to be set.
PAD_$NEWLINE	10	Marks end of an input or output line, makes next text start on a new line.
PAD_$NO_KEY	16#FE	Cursor position report: Indicates that no keystroke accompanies the report.
PAD_$TAB	9	Moves cursor to next tab stop.

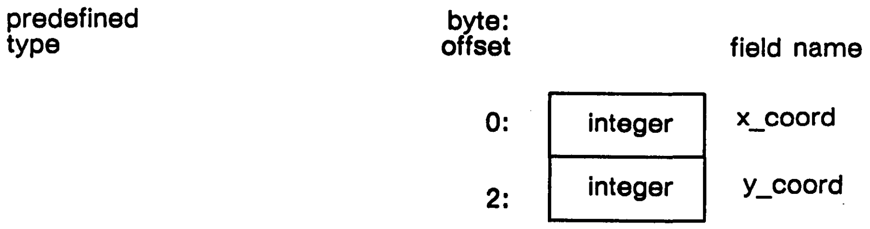
DATA TYPES

PAD_$COORDINATE_T	2-byte integer for x and y bitmap coordinates.

PAD_$CRE_OPT_T	A 2-byte integer. Options of a pane. Any combination of the following pre-defined values: PAD_$ABS_SIZE Size parameter is absolute, rather than relative to the size of the existing pad. PAD_$INIT_RAW Input pad is initially raw, rather than normal (cooked) processing mode.
PAD_$DISPLAY_TYPE_T	A 2-byte integer. Type of display associated with the specified stream id. This is a 2-byte integer. One of the following pre-defined values: PAD_$BW_15P Black and white portrait display. PAD_$BW_19L Black and white landscape display. PAD_$COLOR_DISPLAY Color display (1024 x 1024 pixels). PAD_$800_COLOR Color display (1024 x 800 pixels). PAD_$NONE No display.
PAD_$KEY_DEF_T	An array of up to 256 characters. Display Manager command to be defined on a program- function key using PAD_$DEF_PFK.
PAD_$KEY_NAME_T	An array of up to 4 characters. Name of the program-function key to be defined using PAD_$DEF_PFK.

PAD_$POSITION_T

X and y coordinates of a point on the display. The diagram below illustrates the PAD_$POSITION_T data type:



Field Description:

y_coord
The y coordinate of the point on the display.

x_coord
The x coordinate of the point on the display.

PAD_$REL_ABS_T

A 2-byte integer. Indicates whether cursor movement is relative to the last location, or absolute. X and y are scaled. One of the following pre-defined values:

PAD_$ABSOLUTE
X and y are absolute values. Within a frame, movement is relative to the top left corner of the frame. Outside a frame, x is relative to the left end of the current line, and y is undefined.

PAD_$RELATIVE
Cursor movement is relative to the last location. X and y denote positive or negative offsets to the current cursor position.

PAD_$SIDE_T	A 2-byte integer. Side of a transcript pad that a new pane occupies. One of the following pre-defined values: PAD_$BOTTOM Bottom of transcript pad. PAD_$LEFT Left side of transcript pad. PAD_$RIGHT Right side of transcript pad. PAD_$TOP Top of transcript pad.
PAD_$STRING_T	An array of up to 256 characters. String argument to some functions.

PAD_$TABSTOP_BUF_T

A 100-element array of 2-byte integers. Columns for tab stop settings. Each element contains a column number at which a tab stop will be set. Column numbers are scaled.

PAD_$TYPE_T

A 2-byte integer. A type of pad. One of the following pre-defined values:

PAD_$EDIT
An edit pad.

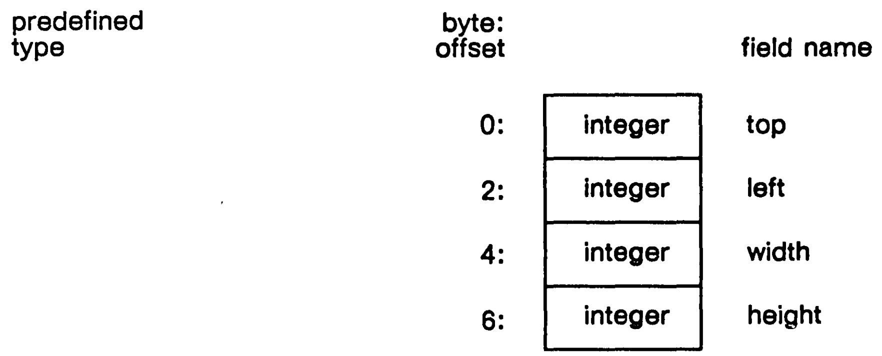
PAD_$INPUT
An input pad.

PAD_$TRANSCRIPT
A transcript pad.

PAD_$READ_EDIT
A read/edit pad.

PAD_$WINDOW_DESC_T

Position of window on display screen. The diagram below illustrates the PAD_$WINDOW_DESC_T data type:



Field Description:

top
The x coordinate of the top left corner of the window, in raster units.

left
The y coordinate of the top left corner of the window, in raster units.

width
The width of the window, divided by the current x scale factor.

height
The height of the window, divided by the current y scale factor.

PAD_$WINDOW_LIST_T

A 10-element array of PAD_$WINDOW_DESC_T record structures. The diagram below illustrates a single element:

predefined type

byte: offset	15 0	field name
0:	integer	top
2:	integer	left
4:	integer	width
6:	integer	height

Field Description:

top
The x coordinate of the top left corner of the window, in raster units.

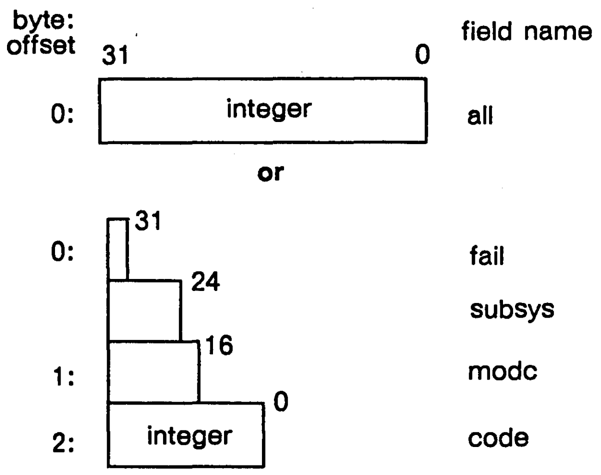
left
The y coordinate of the top left corner of the window, in raster units.

width
The width of the window, divided by the current x scale factor.

height
The height of the window, divided by the current y scale factor.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits are in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

PAD_$CLEAR_FRAME

Clears the current frame, leaving it active.

FORMAT

```
PAD_$CLEAR_FRAME (stream-id, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

seek-key
Unique value identifying the record where clearing begins, in STREAM_$ID_T format. This is a three element array of 4-byte integers. See the STREAM Data Types section for more information.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4-bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to clear information from the frame that you created with the call, PAD_$CREATE_FRAME. Programs that use frames often overwrite text at random points within the frame. You should periodically call PAD_$CLEAR_FRAME to remove this discarded data. By doing so, you prevent data from accumulating in the transcript pad file. You also prevent the Display Manager from invoking the time-consuming frame-rewrite operation.

Clearing begins at the record indicated by the seek-key and continues to the end of the frame. If the first four bytes of the seek-key are 0, the entire frame is cleared. The seek-key is returned by STREAM_$PUT_REC and STREAM_$PUT_CHR. See the STREAM_$ Calls section for more information.

PAD_$CLOSE_FRAME

Closes a frame, leaving its contents in the pad, and returns to line-oriented processing on the input pad.

FORMAT

```
PAD_$CLOSE_FRAME (stream-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4-bytes long. See the PAD Data Types section for more information.

USAGE

After the frame is closed, you can view the frame by scrolling the transcript window backwards. Once the frame is closed, all frame operations except PAD_$CREATE_FRAME are invalid.

PAD_$COOKED

Disables raw mode input or output to a pad.

FORMAT

```
PAD_$COOKED (stream-id, status)
```

INPUT PARAMETERS

stream_id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This call returns the pad to normal (cooked) processing if it is currently in raw mode due to a call to the PAD_$RAW procedure. PAD_$COOKED has no effect if called when the pad is not currently in raw mode. After you execute this procedure, the input window reappears and is empty.

PAD_$CPR_ENABLE

Enables reporting of the keyboard cursor position for an input pad in raw mode. (You can only get keyboard cursor position reports on pads in raw mode).

FORMAT

```
PAD_$CPR_ENABLE (stream-id, report-cpr-type, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the input pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

report-cpr-type

Type of cursor position report. This is a 2-byte integer. Specify one of the following predefined values:

PAD_$CPR_NONE
: Requests no cursor position reports (the default).

PAD_$CPR_CHANGE
: Requests cursor position reports only when the cursor has moved through keystrokes since the last output call or the last position report.

PAD_$CPR_ALL
: Requests a cursor position report with every character.

PAD_$CPR_PICK
: Requests a cursor position report after the cursor has settled after being moved by the touchpad, bitpad, or mouse.

PAD_$CPR_DRAW
: Requests a cursor position report for all cursor positions during cursor movement from the touchpad, bitpad, or mouse.

PAD_$CPR_PICK and PAD_$CPR_DRAW also report new cursor positions resulting from Display Manager commands; for example, arrow keys, tabs, TR, TL, TB.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

You get cursor position reports in response to a STREAM_$GET_REC call, intermixed with raw character data. The Display Manager uses a single byte to represent a raw keystroke. It uses a 6-byte sequence to give you a cursor position report. The sequence looks like this:

```
1 byte --  PAD_$CPR_FLAG, indicating that the next 5
           bytes are a cursor position report.

2 bytes -- The x coordinate of the cursor position.

2 bytes -- The y coordinate of the cursor position.

1 byte --  The raw keystroke or PAD_$NO_KEY if there
           is no keystroke accompanying this cursor
           position report, or PAD_$LEFT_WINDOW if
           the cursor moved outside the window.
```

The x and y coordinates are scaled according to the scaling factors in effect at the time of the PAD_$CPR_ENABLE call (see PAD_$SET_SCALE). The x and y coordinates are relative to the upper left corner of the frame. (If the cursor is not inside a frame, the x coordinate is relative to the start of the current line, and the y coordinate is meaningless.)

In raw mode, the Display Manager does not automatically echo typed keystrokes nor move the cursor. If your program requests PAD_$CPR_ALL but does not act to move the cursor (typically by displaying typed keystrokes), each keystroke produces a cursor position report, usually describing the same cursor position. If you don't intend to echo keyboard input, request PAD_$CPR_CHANGE instead to avoid redundant cursor position reports.

PAD_$CPR_CHANGE compares the present keyboard cursor with the last *output* cursor position. In raw mode, the position of the output cursor is under program control. Therefore, if your program does not move the output cursor to follow the input cursor (which you can move) you may receive a stream of cursor position reports, all showing the same position, as long as the keyboard cursor is not in the same position as the output cursor.

PAD_$CREATE

Creates a new pad and a window pane to view it.

FORMAT

```
PAD_$CREATE (pathname, name-length, pane-type, related-stream-id, side,
            pane_options, pane-size, pane-stream-id, status)
```

INPUT PARAMETERS

pathname

Pathname to a file to display in the window pane, in NAME_$PNAME_T format. This is an array of up to 256 characters.

If the specified pathname refers to an existing file, the Display Manager positions the new window pane at the beginning of the file, and displays any existing data. If the given pathname does not refer to an existing file, a permanent file with that name is created. You usually use a null pathname when creating a transcript pad. You must specify a null pathname when creating an input pad.

name-length

Length of the pathname in bytes. This is a 2-byte integer. A value of 0 creates a temporary file for the pad. You must specify 0 when creating an input pad.

pane-type

The window pane type in PAD_$TYPE_T format. This is a 2-byte integer. Specify one of the following predefined values:

PAD_$EDIT	Creates a pad in which you can view and modify the associated file.
PAD_$INPUT	Creates an input pad.
PAD_$READ_EDIT	Creates a pad in which you can view but not modify the associated file.
PAD_$TRANSCRIPT	Creates a transcript pad.

related-stream-id

The stream ID of a transcript pad, in STREAM_$ID_T format. This is a 2-byte integer. The related-stream-id for an input window pane (PAD_$INPUT) must refer to an open transcript window pane that has no other input window pane associated with it.

side

The side of the transcript pad that the new window occupies, in PAD_$SIDE_T format. This is a 2-byte integer. Specify one of the following predefined values:

PAD_$LEFT

PAD_$RIGHT

PAD_$TOP

PAD_$BOTTOM

You must specify PAD_$BOTTOM when creating an input window pane for a transcript window pane.

pane-options

Attributes of the pane. This is a 2-byte integer. In Pascal, specify any combination of the following set of predefined values:

PAD_$ABS_SIZE
: Specifies an absolute pane-size. If not given, the pane-size parameter is a relative value.

PAD_$INIT_RAW
: Indicates that a new input pad is initially in raw rather than cooked mode. This is for input pads only, it is invalid for any other pad types.

In FORTRAN, specify either 0, to indicate that the pane-size is relative, or give the sum of the desired options.

pane-size

Size of the pane. This is a 2-byte integer. A window pane always takes up one full side of the related window. The size refers only to the depth of the window.

You can express the pane size either as a percentage relative to the existing transcript window, or as an absolute value in terms of the current scale factor.

If you specify the pane-size as an absolute size, the Display Manager attempts to keep the window pane at that size. However, the window pane can never be larger than the related window, so that, if the related window shrinks below the size requested, the window pane also shrinks.

In addition, if you specify the pane size as an absolute size, the value given is multiplied by the current scale factors to yield raster units. The default scale factors are the current font size so that, unless you change the scale, you should express the pane-size in terms of lines or characters.

An input window pane will normally be one line deep, but can grow and shrink depending on how many lines of input are waiting for action. You should specify a size that accommodates this because the size parameter determines the *maximum* number of lines that the input window pane can occupy. The size of an input window pane can never be less than 1. (A common relative size is 20.)

OUTPUT PARAMETERS

pane-stream-id
Number of the stream on which the new window pane is open, in STREAM_$ID_T format.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types for more information.

USAGE

Use this call to create a new pad and window pane on a related stream. The related stream can be either the stream ID of a transcript pad that you previously created with a call to PAD_$CREATE or PAD_$CREATE_WINDOW. For transcript pads, the stream ID can be a standard output stream such as STREAM_$STDOUT, or STREAM_$ERROUT.

You can create any number of window panes on top of the original transcript pad up to the maximum of 40 pads and 60 windows.

You must use PAD_$CREATE to create an input pad for an existing transcript pad.

PAD_$CREATE_FRAME

Creates a frame in a pad.

FORMAT

```
PAD_$CREATE_FRAME (stream-id, width, height, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the input pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

width
Width of the new frame in pixels. This is a 2-byte integer. Value can be up to 32767 raster units. Width is scaled according to the current scale factors.

height
Height of the new frame in pixels. This is a 2-byte integer. Value can be up to 32767 raster units. Height is scaled according to the current scale factors.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to create a frame on an existing transcript pad. Because you can move the cursor anywhere within the frame, create frames when you want to have more control over the cursor position in a given area of the screen.

Your program must either close the frame with PAD_$CLOSE_FRAME or delete the frame with PAD_$DELETE_FRAME before exiting. (Note that you can review a closed frame by scrolling the transcript window backwards, but a deleted frame no longer exists.)

PAD_$CREATE_ICON

Creates a new pad and associated window in icon format.

FORMAT

```
PAD_$CREATE_ICON (pathname, name-length, type, unit, icon-pos, icon-char,
                  window, stream-id, status)
```

INPUT PARAMETERS

pathname

Pathname to a file to display in the pad, in NAME_$PNAME_T format. This is an array of up to 256 characters.

If the specified pathname refers to an existing file, the Display Manager positions the new window pane at the beginning of the file, and displays any existing data. If the given pathname does not refer to an existing file, a permanent file with that name is created. You usually create a null pathname when creating a transcript pad.

name-length

Length of the pathname string. This is a 2-byte integer. A value of 0 creates a temporary file for the pad.

type

Pad type in PAD_$TYPE_T format. This is a 2-byte integer. Specify one of the following predefined values:

PAD_$TRANSCRIPT	Creates a transcript pad.
PAD_$EDIT	Creates a pad in which you can view and modify the associated file.
PAD_$READ_EDIT	Creates a pad in which you can view but not modify the associated file.

unit

Display unit number associated with the stream-ID. This is a 2-byte integer. Usually, there is only one display per node so this value is often 1.

icon-pos

X- and y-coordinates of the upper left corner of the icon window, in PAD_$POSITION_T format. This data type is four bytes long. See the PAD_$ Data Types section for more information.

icon-char

Icon font character to be displayed in the icon window. This character must reside in the current icon font file. A null character value ('') causes the Display Manager to select the default icon character for this pad type.

window

Window descriptor giving the position on the screen that the new window will occupy when expanded to full size (the icon window size is fixed by the the font character selected), in PAD_$WINDOW_DESC_T format. This data type is 8 bytes long. See the PAD Data Types section for more information.

The window specified is the usable part of the displayed window. The displayed window is larger by the size of the border and the legend. If you specify either the width or the height as zero, the window is created using the same rules as for Display Manager commands (see the *DOMAIN System Command Reference*).

OUTPUT PARAMETERS

stream-id

Number of the stream on which the new window is open, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to create a new pad and window in icon format. To change this window from icon format to a full-sized window, use PAD_$SELECT_WINDOW. To change an existing window into icon format, use PAD_$MAKE_ICON.

PAD_$CREATE_WINDOW

Creates a new pad and a window to view it.

FORMAT

```
PAD_$CREATE_WINDOW (pathname, name-length, pad-type, unit, window,
                    stream-id, status)
```

INPUT PARAMETERS

pathname

Pathname to a file to display in the pad, in NAME_$PNAME_T format. This is an array of up to 256 characters. When creating an edit or read/edit pad, this is the pathname of the permanent file for use as the pad. If a file with this name exists, the Display Manager positions the new window at the top of the pad. If such a file doesn't exist, a new file with that name is created. You usually use a null pathname when creating a transcript pad.

name-length

Length of the pathname string. This is a 2-byte integer. When creating an edit or read/edit pad, a value of 0 creates a temporary file as the pad.

pad-type

Pad type in PAD_$TYPE_T format. This is a 2-byte integer. Specify one of the following predefined values:

PAD_$TRANSCRIPT	Creates a transcript pad.
PAD_$EDIT	Creates a pad in which you can view and modify the associated file.
PAD_$READ_EDIT	Creates a pad in which you can view but not modify the associated file.

unit

Display unit number to use. This is a 2-byte integer. Usually there is only one node per display so this value is often 1.

window

Window descriptor giving the position on the screen that the new window will occupy, in PAD_$WINDOW_DESC_T format. This data type is 8 bytes long. See the PAD Data Types section for more information.

The window specified is the usable part of the displayed window. The displayed window is larger by the size of the border and the legend. If you specify either the width or the height as zero, the window is created using the same rules as for Display Manager commands (see the *DOMAIN System Command Reference*).

OUTPUT PARAMETERS

stream-id

Number of the stream on which the new window is open, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to create a new pad and window to view it. Use PAD_$CREATE to create a new pad and window pane on an existing transcript pad. To create an input pad, you must use PAD_$CREATE.

PAD_$DEF_PFK

Defines a program function key for use by a program.

FORMAT

```
PAD_$DEF_PFK (stream-id, key-name, definition, def-len, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

key-name
Name of the key to be defined. This is a 4-byte character array. Use the key name exactly as it appears in the *DOMAIN System Command Reference.* Use uppercase letters (for example, F1) except when you are redefining a lowercase letter key (such as x). Do not use quotes in this character array (except to redefine the quote key).

definition
Display Manager command you want executed whenever the specified key is pressed. This is an array of up to 128 characters.

def-len
Length of the definition in bytes. This is a 2-byte integer. A value of 0 (zero) returns the key to its original definition.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

PAD_$DEF_PFK defines a program function key for use by a program. When you press a defined key, the definition string is entered as a Display Manager command.

Program function keys defined by PAD_$DEF_PFK behave like keys defined through the Display Manager, except that the definition is only effective within windows viewing the associated pad.

Definitions remain in effect after the program finishes executing, but only within windows viewing the pad associated with the program.

The Display Manager command string you specify as the key definition is often the ES command, which contains a text string and lets the program function key simulate the typing of that text. You may specify the ER command, which introduces a two-digit hexadecimal number and feeds that value directly to the program when the user presses the key. The ER command essentially enables raw-mode input of the specified value, with no echoing or other processing by the Display Manager. The *DOMAIN System Command Reference* contains more details on these commands.

The rules for naming keys in PAD_$DEF_PFK differ from the rules for naming keys in the KD (key definition) Display Manager command. That command implicitly converts letters to uppercase and allows the use of single quotes.

PAD_$DELETE_FRAME

Deletes and clears the current frame.

FORMAT

```
PAD_$DELETE_FRAME (stream-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

PAD_$DELETE_FRAME removes the current frame from the pad. After executing this procedure, the pad returns to line-oriented processing. You cannot perform further frame operations until you create another frame with a call to PAD_$CREATE_FRAME.

PAD_$DM_CMD

Executes a Display Manager command.

FORMAT

```
PAD_$DM_CMD (stream-id, command, command-length, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which a pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

command
Display Manager command, in PAD_$STRING_T format. This is an array of up to 256 characters.

command-length
Length of the command string in bytes. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this procedure with caution since it performs actions that you normally perform with the keyboard. Because of this, PAD_$DM_CMD may produce unexpected results.

You can find a list of Display Manager commands in the *DOMAIN System Command Reference.*

PAD_$EDIT_WAIT

Suspends program execution until you close an edit window pane, then converts the stream so that the program can access the new input.

FORMAT

```
PAD_$EDIT_WAIT (pane-stream-id, status)
```

INPUT PARAMETERS

pane-stream-id
Number of the stream on which the edit window is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Your program suspends execution until you close the edit window pane with a CTRL/Y (PW; WC -Q command) or a CTRL/N (WC -Q command).

If you close the edit window pane with a CTRL/N, and the file did not exist before the edit window pane was created, PAD_$EDIT_WAIT returns an error, usually indicating that the file was deleted while open.

You must use this procedure before reading a file edited through an edit window pane.

PAD_$ICON_WAIT

Waits until a window is expanded from an icon format to a full-window size or until the icon window moves.

FORMAT

```
PAD_$ICON_WAIT (stream-id, window-no, icon-moved, icon-pos, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

icon-moved
A Boolean value indicating icon-window movement. It returns a value of TRUE if the icon window has moved.

icon-pos
New position of the moved icon window in PAD_$POSITION_T format. This data type is 4 bytes long. See the PAD_$ Data Types section for more information.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This call may be used on any type of pad.

If the window is not currently in icon format, this call returns immediately.

PAD_$INQ_DISP_TYPE

Returns the type of display associated with the given stream ID.

FORMAT

```
PAD_$INQ_DISP_TYPE (stream-id, display-type, unit, status)
```

INPUT PARAMETERS

stream-id
Number of the stream associated with an input or transcript pad, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

display-type
Type of display associated with the specified stream ID, in PAD_$DISPLAY_TYPE_T format. This is a 2-byte integer. Returns one of the following predefined values:

PAD_$NONE	No display
PAD_$BW_15P	Black and white portrait
PAD_$BW_19L	Black and white landscape
PAD_$COLOR_DISPLAY	Color display (1024 x 1024)
PAD_$800_COLOR	Color display with fewer pixels (1024 x 800)
PAD_$COLOR2_DISPLAY	Color display (1280x1024x8)
PAD_$COLOR3_DISPLAY	Color display (1024x800x8)
PAD_$COLOR4_DISPLAY	Color display (1024x800x4)

unit
Display unit number. This is a 2-byte integer. This parameter is reserved for future use, it will always have the value of 1.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

PAD_$INQ_DISP_TYPE returns the display type and unit number associated with the stream ID. The display unit number can be used as an argument to PAD_$CREATE_WINDOW.

PAD _ $INQ _ FONT

Returns information about the current font.

FORMAT

```
PAD_$INQ_FONT (stream-id, font-width, font-height,
               font-name, font-size, font-len, status)
```

INPUT PARAMETERS

stream-id
The number of the stream on which the pad is open, in STREAM _ $ID _ T format. This is a 2-byte integer.

font-size
The number of bytes available in the "font-name" string buffer. This is a 2-byte integer. PAD _ $INQ _ FONT fills the "font-name" output parameter with this many characters of information. If you do not want to know the pathname, you can specify 0 (zero) as the value of "font-size."

OUTPUT PARAMETERS

font-width
Width of the font in raster units. This is a 2-byte integer. For fonts in which different characters have different widths, "font-width" describes the width of the space character.

font-height
Height of the font in raster units. This is a 2-byte integer. The height includes any interline spacing specified in the font file.

font-name
Full pathname of the font, up to the node entry directory (/), in PAD _ $STRING _ T format. This is an array of up to 256 characters. The pathname is returned with the correct character case (i.e., upper-case characters in the pathname are returned as upper-case; lower-case as lower-case).

font-len
Length of the "font-file" pathname. This is a 2-byte integer. If this value is greater than the input parameter "font-size," the Display Manager truncates the returned pathname.

status
Completion status, in STATUS _ $T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to determine which font your program is currently using. Your program can use up to 100 different character fonts as long as you initially load all the fonts that you intend to use with PAD _ $LOAD _ FONT. When you want your program to use a specific font, call PAD _ $USE _ FONT to invoke a previously loaded font. Each time you want to change a loaded font, use PAD _ $USE _ FONT.

PAD_$INQ_FULL_WINDOW

Returns information about the entire window specified, including the border and legend.

FORMAT

```
PAD_$INQ_FULL_WINDOW (stream-id, window-no, window, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Window number of the window open on the pad. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

window
Window descriptor giving the position on the screen that the window occupies, including the border and legend, in PAD_$WINDOW_DESC_T format. This data type is 8 bytes long. See the PAD Data Types section for more information.

The window gives the position of the top left corner, width and height of the window. The values appear in the following order: top, left, width, height. Top and left are expressed in raster units. Width and height are divided by the current scale factors.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to determine exactly how much screen space your window uses, including the border and legend. A call to PAD_$INQ_WINDOWS returns similar information about the usable part of the display windows (not including the border and legend).

Note that if the specified stream-id and window-no refer to a window pane, the information returned is for the outermost containing window.

PAD_$INQ_ICON

Returns information about a window in icon format.

FORMAT

```
PAD_$INQ_ICON (stream-id, window-no, icon-pos, icon-char, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

icon-pos
Position of the icon, in PAD_$POSITION_T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

icon-char
Character currently displayed in the icon window.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

If the window is not currently in icon format, the information returned describes its previous icon status, if any, and its future icon status, should the Display Manager command ICON or the PAD_$MAKE_ICON call be issued with the default setting for icon-pos and icon-char.

PAD_$INQ_ICON_FONT

Returns information about the current icon font.

FORMAT

```
PAD_$INQ_ICON_FONT (stream-id, window-no,font-name
                font-buf-size, font-len, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Window for which information is wanted. Window-no is an index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

font-buf-size
Number of bytes available in the font-name buffer string. This is a 2-byte integer. PAD_$INQ_FONT fills the output parameter, font-name, with this many characters of information.

OUTPUT PARAMETERS

font-name
Pathname the font from the node entry directory (/), in NAME_$PNAME_T format. This is an array of up to 256 characters. The pathname is returned with the correct character case (i.e., upper-case characters in the pathname are returned as upper-case; lower-case as lower-case).

font-len
Length of the font file pathname. This is a 2-byte integer. If this value is greater than the input parameter font-size, the Display Manager truncates the returned pathname to fit in the smaller number of characters.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to get the pathname of the icon font in use. You can change the icon font in use with the call, PAD_$SET_ICON_FONT.

The default icon font file is /SYS/DM/FONTS/ICONS. You can create a new icon font file to contain your own icons by using the font editor EDFONT. See the *DOMAIN System Command Reference* for a complete description of EDFONT.

PAD_$INQ_KBD

Returns information about the keyboard currently in use.

FORMAT

```
PAD_$INQ_KBD (stream-id, buffer-size, kbd-suffix, length, status)
```

INPUT PARAMETERS

stream-id
Number of the stream associated with an input or transcript pad, in STREAM_$ID_T format. This is a 2-byte integer.

buffer-size
Number of bytes available in the "kbd-suffix" string buffer. This is a 2-byte integer.

OUTPUT PARAMETERS

kbd-suffix
Suffix to be appended to Display Manager pathnames to locate a key definition file, in PAD_$STRING_T format. This is an array of up to 256 characters. Suffixes used by standard DOMAIN software are:

Null string	Corresponds to the 880 keyboard.
Value of "2"	Corresponds to the low-profile keyboard.
Value of "3"	Corresponds to the low-profile keyboard with numeric keypad.

(Display Manager pathnames for key definitions are /SYS/DM/STD_KEYS and USER_DATA/KEY_DEFS.)

length
Actual length of the string. This is a 2-byte integer. If the length parameter is greater than "kbd-suffix," it truncates "kbd-suffix."

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to determine which keyboard is in use. For example, you might want to set up program definition keys according to the type of keyboard in use.

PAD_$INQ_POSITION

Returns the position of the output cursor.

FORMAT

```
PAD_$INQ_POSITION (stream-id, x, y, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

x
X position of the output cursor. This is a 2-byte integer.

y
Y position of the output cursor. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

X and y are divided by the current scale factors.

If this procedure is executed when the cursor is inside a frame, x and y are relative to the upper left corner of the frame. If the cursor is not in a frame, x represents the position on the line and y is undefined.

PAD_$INQ_VIEW

Returns information about the position of a window relative to a pad.

FORMAT

```
PAD_$INQ_VIEW (stream-id, window-number, line, eof-linenum, x-offset,
                y-offset, status)
```

INPUT PARAMETERS

stream-id
Number of the stream associated with an input or transcript pad, in STREAM_$ID_T format. This is a 2-byte integer.

window-number
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

line
Number of the line being viewed. This is a 4-byte integer.

eof-linenum
Last line or frame on the pad. This is a 4-byte integer.

x-offset
Distance the pad is horizontally scrolled. This is a 2-byte integer.

y-offset
Distance the pad is vertically scrolled. This is a 2-byte integer. Only frames can be vertically scrolled.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this routine in conjunction with PAD_$SET_VIEW to control the display of graphic images that are larger than the window. PAD_$INQ_VIEW describes the pad element currently being viewed through the given window, usually a transcript pad element.

If the element currently in view is a frame, x-offset and y-offset describe how the window is positioned in relation to the frame. If you are viewing the current frame and not some previous part of the pad, the value of eof-linenum will be equal to the line parameter.

If the element currently in view is not a frame, the line parameter is the number of the top line in the window.

PAD_$INQ_WINDOWS

Returns information about windows viewing the current pad.

FORMAT

```
PAD_$INQ_WINDOWS (stream-id, windowlist, window-list-size,
                  window-no, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-list-size

Maximum number of windows on which information is desired. This is a 2-byte integer.

OUTPUT PARAMETERS

windowlist

Information describing a window, in PAD_$WINDOW_LIST_T format. This data type is an array of up to 10 elements, each of which is in PAD_$WINDOW_DESC_T format (four 2-byte integers). See the PAD Data Types section for more information.

Windowlist indicates the top left corner and the width and height of each window open on the pad, up to wlistsize. The values appear in the following order: top, left, width, height. Top and left are expressed in raster units, but width and height are divided by the current scale factors.

window-no

The number of windows open on the pad. This is a 2-byte integer. Window number one always refers to the first window created to view the pad. Use this parameter in calls that require a window number.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

PAD _$LOAD _FONT

Loads a character font.

FORMAT

```
PAD_$LOAD_FONT (stream-id, font-pathname, name-length, font-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM _$ID _T format. This is a 2-byte integer.

font-pathname
Pathname of the file containing the character font, in PAD _$STRING _T format. This is an array of up to 256 characters.

name-length
Length of the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

font-id
Font identifier, to be used in later calls to PAD _$USE _FONT. This is a 2-byte integer.

status
Completion status, in STATUS _$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Your program can use up to 100 different character fonts as long as you initially load all the fonts that you intend to use with PAD _$LOAD _FONT. When you want your program to use a specific font, call PAD _$USE _FONT to invoke a previously loaded font. Each time you want to change a loaded font, use PAD _$USE _FONT. To determine which font your program is currently using, call PAD _$INQ _FONT.

The Display Manager first attempts to find the font file using the pathname directly, with the normal defaults. If it fails to find the file, it searches in /SYS/DM/FONTS.

PAD _$LOAD _FONT does not switch fonts. It merely loads the font into the invisible portion of display memory and returns a font ID. After loading the font, your program can call PAD _$USE _FONT to use it.

You can load up to 100 fonts in a given pad.

PAD_$LOCATE

Returns the position of the keyboard cursor in response to a keystroke.

FORMAT

```
PAD_$LOCATE (stream-id, x, y, character, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

x
X position of the input cursor. This is a 2-byte integer.

y
Y position of the input cursor. This is a 2-byte integer.

character
Value of the key pressed.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This procedure returns the cursor position only when a raw character is entered. If the pad is in raw mode, any keystroke will do. In cooked mode, the ER command must be used. This command is usually entered through a function key definition.

The keyboard cursor position must be within the transcript pad.

X and y are divided by the current scale factors.

PAD _ $MAKE _ ICON

Changes an existing window into icon format.

FORMAT

```
PAD_$MAKE_ICON (stream-id, window-no, icon-char, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM _ $ID _ T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD _ $INQ _ WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

icon-char
Icon font character to be displayed in the icon window. This character must reside in the current icon font. A 0 (zero) causes the Display Manager to select the default icon character for this pad type.

OUTPUT PARAMETERS

status
Completion status, in STATUS _ $T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This call changes an existing full-size window into icon format. (If the window is invisible at the time of the call, it first becomes visible and then becomes an icon.) To create a completely new pad and window in icon format, use PAD _ $CREATE _ ICON.

If the window is already an icon, this call has no effect.

Specify the display position for the new icon using the PAD _ $SET _ ICON _ POS routine before executing this call. If you do not do this, the Display Manager assigns a default icon position descriptor and font character.

The size of the icon window is not user-definable. It is determined automatically by the size of the font character specified.

PAD_$MAKE_INVISIBLE

Makes a visible window invisible.

FORMAT

```
PAD_$MAKE_INVISIBLE (stream-id, window-no, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

The effect of this call is the same as if the window were completely obscured by other windows on the screen, except that no amount of pushing, popping, moving, or growing can make it reappear. Only a subsequent call to PAD_$SELECT_WINDOW will restore it to visibility in its full-size format.

If the window is currently invisible, this call has no effect.

If the window is currently in icon format, it will first be made into a full-size window and then turned invisible.

PAD_$MOVE

Moves the output cursor.

FORMAT

```
PAD_$MOVE (stream-id, rel-abs, x, y, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

rel-abs
Indicates whether cursor movement is to be relative or absolute. This is a 2-byte integer in PAD_$REL_ABS_T format. Specify one of the following predefined values:

PAD_$RELATIVE
Movement is relative to the last cursor position. X and y denote positive or negative offsets to the current cursor position, scaled according to the current scale factors.

PAD_$ABSOLUTE
X and y are absolute, within the frame. X and y must be positive. Within a frame, movement is relative to the upper left corner of the frame. Outside a frame, x is relative to the left end of the current line and y is not used. In both cases, x and y are scaled according to the current scale factors.

x
Change to the x-coordinate of the cursor position. This is a 2-byte signed integer.

y
Change to the y-coordinate of the cursor position. This is a 2-byte signed integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

PAD_$MOVE changes the position of the output cursor, which marks the place where the next program output will appear.

The cursor can move vertically only within a frame, not on a line. The Display Manager uses the y value only when a frame is active, and ignores it otherwise.

PAD_$POP_PUSH_WINDOW

Pops or pushes a window.

FORMAT

```
PAD_$POP_PUSH_WINDOW(stream-id, window-no, flag, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
The index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

flag
Indicates if the window is to be pushed or popped. This is a Boolean variable. A value of TRUE pops the specified window to the top of the screen, ensuring that no portion of the window is hidden by another window. A value of FALSE pushes the specified window to the bottom of the screen, allowing other windows to cover it wherever possible.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

PAD_$RAW

Places an input or transcript pad in raw mode.

FORMAT

```
PAD_$RAW (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer. The stream-id given should refer to an input stream, usually standard input (STREAM_$STDIN). PAD_$RAW has no effect on output.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

PAD_$RAW puts the pad into raw mode, and has no effect if it is already in raw mode. PAD_$COOKED returns the pad to normal processing.

In raw mode, the Display Manager sends keyboard input directly to the program without echoing or processing it in any way. ASCII control characters are also sent to the program, but the Display Manager still handles its function keys.

The Display Manager immediately displays every character it receives, unless the window is in HOLD mode. If the window is in HOLD mode, new characters do not appear until the keyboard user scrolls the window or releases HOLD.

When it executes this procedure, the Display Manager clears the input pad and shrinks its window size to zero. The keyboard cursor moves to the current output cursor position in the transcript pad. While the pad is in raw mode, the keyboard and output cursors usually move together.

NOTE: A program using PAD_$RAW *must* execute PAD_$COOKED before termination. Most system utilities, including the Shell, will not work correctly in raw mode.

PAD_$SELECT_WINDOW

Makes an invisible window visible and/or changes an icon-format window into a full-sized window.

FORMAT

```
PAD_$SELECT_WINDOW (stream-id, window-no, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use PAD_$MAKE_INVISIBLE to make windows invisible.

If this call is used to expand an icon to full-size format, the position and dimensions of the large window are the same as those it had when it was last full size. If it was never full-size, its position and dimensions are those specified (or defaulted) when the icon was created (either by PAD_$CREATE_ICON, or by the Display Manager commands CP, CV, CE, or CPB with the -I option specified).

PAD _ $SET _ AUTO _ CLOSE

Sets a window to close automatically when its pad closes.

FORMAT

```
PAD_$SET_AUTO_CLOSE (stream-id, window-no, auto-close, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM _ $ID _ T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD _ $INQ _ WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

auto-close
Indicates whether the window is to close automatically. This is a Boolean value. If TRUE, the window disappears when the pad onto which it opens is closed.

OUTPUT PARAMETERS

status
Completion status, in STATUS _ $T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

When a program first makes this call and then does a STREAM _ $CLOSE, the window specified is closed and deleted from the screen. This is equivalent to specifying the Display Manager command WC -A for a window.

PAD_$SET_BORDER

Adds a border to, or removes the border from, a full window.

FORMAT

```
PAD_$SET_BORDER(stream-id, window-number, flag, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-number
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

flag
Indicates whether to add or remove a border. This is a Boolean variable. If TRUE, the window appears with a border around its edges and a legend at the top. If FALSE, any border and legend are removed from the window, making the window's usable area equal to the amount of space the window occupies on the screen.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this procedure to remove or add a border to a full window that has no other panes associated with it. To get a full window without any panes, you can either create a transcript pad and never make a PAD_$CREATE call to add panes, or create a transcript pane that covers the entire window. Another way to get a full window is to make an input pane invisible by using the PAD_$RAW call.

PAD_$SET_FULL_WINDOW

Moves a window or sets a window position for future use.

FORMAT

```
PAD_$SET_FULL_WINDOW (stream-id, window-no, window, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

window
Window descriptor giving the position on the screen that the new window will occupy when expanded to a full-sized window, in PAD_$WINDOW_DESC_T format. This data type is 8 bytes long. See the PAD_$ Data Types section for more information.

The window specified is the entire window, including the border, legend, and usable part of the window. The call, PAD_$INQ_FULL_WINDOW returns information about the entire window.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

If the window specified is currently in icon format or is invisible, this call establishes a full-size window position for future use (i.e., when your program calls PAD_$SELECT_WINDOW to expand the icon into a full-size window, or you issue the Display Manager commands ICON or WI).

If the window specified is currently full-size, then the window is repositioned to the location given by window.

PAD_$SET_ICON_FONT

Sets the current icon font to a specified font name.

FORMAT

```
PAD_$SET_ICON_FONT (stream-id, window-no, font-name,
                    font-len, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Window whose icon font you want to change. Window-no is an index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the first window created to view the pad.

font-name
Full pathname of the font, up to the node entry directory (/), in NAME_$PNAME_T format. This is an array of up to 256 characters.

font-len
Length of the font file pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to change the icon font in use. This call changes the font of the specified window only.

When a window is created either as a full window or an icon, the Display Manager assigns it an icon from "active icon font." The default active icon font is in /SYS/DM/FONTS/ICONS. You can specify another font to be the active icon font by using the FL command with the -I option.

You can create a new icon font file to contain your own icons by using the font editor EDFONT. See the *DOMAIN System Command Reference* for a complete description of EDFONT.

If the window is in icon format at the time of this call, the icon in the display changes to the new font immediately.

PAD_$SET_ICON_POS

Moves an icon or sets an icon position for future use.

FORMAT

```
PAD_$SET_ICON_POS (stream-id, window-no, icon-pos, icon-char, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer. Window number one always refers to the original transcript pad.

icon-pos
New position (x and y coordinates) of the icon, in PAD_$POSITION_T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

icon-char
Character to be displayed in the icon window.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

If the window specified is currently in full-size format, this call establishes an icon position for future use (i.e., when your program calls PAD_$MAKE_ICON to turn the window into icon format, or you use the Display Manager command ICON).

If the window specified is already in icon format, then the icon is repositioned to the location given by icon-pos, and the specified icon-char replaces the current one.

Compare this call to PAD_$SET_FULL_WINDOW, which performs the same operations for full-size windows.

The size of the icon window is not user-definable. It is determined automatically by the size of the font character specified.

PAD_$SET_SCALE

Sets a scale factor for cursor operations.

FORMAT

```
PAD_$SET_SCALE (stream-id, x-factor, y-factor, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

x-factor
Scale factor for the x-coordinate. This is a 2-byte integer.

y-factor
Scale factor for the y-coordinate. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Specify a scale factor of zero to use the scale of the current character font. This is the default.

Specify a nonzero scale factor to use that number as a multiplier for raster units. One raster unit is equal to one bit in the display.

The scale factor is used to convert between raster units and numbers supplied in routines such as PAD_$MOVE. When using the scale of the current font, you express dimensions in terms of characters and lines. In any case, the numbers you enter are multiplied by the scale factor to yield raster units, and raster units are divided by the scale factor before being returned.

The scale factor is used to process input or output for PAD_$CPR_ENABLE, PAD_$CREATE_FRAME, PAD_$INQ_POSITION, PAD_$LOCATE, PAD_$MOVE, and PAD_$INQ_WINDOWS. In PAD_$INQ_WINDOWS, height and width are scaled, but top and left are not. PAD_$INQ_FONT always returns dimensions in terms of raster units.

The scale factors set with this call apply to the specified stream until specifically reset, even after the calling program ends. Your program should not depend on the scale factors being correctly set, but should call PAD_$SET_SCALE to explicitly set the scale factors as desired.

PAD_$SET_TABS

Sets tab stops within a pad.

FORMAT

```
PAD_$SET_TABS (stream-id, tab-stop-array, no-of-tabs, status)
```

INPUT PARAMETERS

stream-id

Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

tab-stop-array

Columns for tab stops. This is an array of up to 100 2-byte integers. Each element in the array contains a column number at which a tab stop will be set. Column numbers are scaled according to the PAD_$SET_SCALE procedure.

For example, assume that the current vertical and horizontal scale factors are both equal to one. A three-element array containing the integers 100, 300, and 500 would specify tab stops at bit positions 100, 300, and 500 on the screen. Because the display contains approximately 100 bits per inch, these tab stops would be set about 1, 3, and 5 inches (2.54, 7.62, and 12.70 cm) from the left edge of the screen.

no-of-tabs

Number of tab stops set. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This procedure sets tabs only for the pad open on the specified stream. Tab stops for all other pads are unchanged.

The default tab setting has tabs every 4 columns.

PAD_$SET_VIEW

Positions a window to establish a given view.

FORMAT

```
PAD_$SET_VIEW (stream-id, window-no, line, x-offset,
                y-offset, status)
```

INPUT PARAMETERS

stream-id
Number of the stream ssociated with a transcript pad, in STREAM_$ID_T format. This is a 2-byte integer.

window-no
Index into the window list returned by PAD_$INQ_WINDOWS. This is a 2-byte integer.Window number one always refers to the first window created to view the pad.

line
Line number to view. This is a 4-byte integer.

x-offset
Distance to scroll the pad horizontally. This is a 2-byte integer.

y-offset
Distance to scroll the pad vertically (for frames only). This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

This routine repositions a window to establish a particular view of a transcript pad. Programs can call this routine after a call to PAD_$INQ_VIEW and in conjunction with calls to PAD_$INQ_WINDOWS to control the display of graphic images that are larger than the window.

PAD_$USE_FONT

Invokes a loaded font.

FORMAT

```
PAD_$USE_FONT (stream-id, font-id, status)
```

INPUT PARAMETERS

stream-id
Number of the stream on which the pad is open, in STREAM_$ID_T format. This is a 2-byte integer.

font-id
Font identifier returned by PAD_$LOAD_FONT. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD Data Types section for more information.

USAGE

Use this call to change your program's current character font.

Your program can use up to 100 different character fonts as long as you initially load all the fonts that you intend to use with PAD_$LOAD_FONT. When you want your program to use a specific font, call PAD_$USE_FONT to invoke a previously loaded font. Each time you want to change a loaded font, use PAD_$USE_FONT. To determine which font your program is currently using, call PAD_$INQ_FONT.

NOTE: Use PAD_$USE_FONT only to *change* the current font in use. You will get erroneous results if the call specifies the font that is already currently in use.

ERRORS

STATUS_$OK
Successful completion.

PAD_$2MNY_CLIENTS
Operation illegal with more than one client process.

PAD_$2MNY_INPUT_PADS
Only one input pad per transcript.

PAD_$BAD_KEY_NAME
Key name not found.

PAD_$EDIT_QUIT
User quit (WC -Q) out of edit pane.

PAD_$FONT_FILE_ERR
Could not access font file.

PAD_$ID_OOR
Stream id out of range.

PAD_$ILL_PARAM_COMB
Conflict in PAD_$CREATE call.

PAD_$ILL_PTYPE
Cannot do operation on this type of pad.

PAD_$NO_SUCH_WINDOW
Bad window number in INQ/SET_VIEW.

PAD_$NO_WINDOW
Window no longer exists.

PAD_$NOT_ASCII
Existing pad in PAD_$CREATE is not ASCII.

PAD_$NOT_INPUT
Operation valid on input pads only.

PAD_$NOT_RAW
Operation requires pad be in raw mode.

PAD_$NOT_TRANSCRIPT
Operation valid on transcript pads only.

PAD_$STREAM_NOT_OPEN
No stream open on this SID.

PAD_$STREAM_NOT_PAD
Preferred stream is not a pad.

PAD_$TOO_MANY_FONTS
Too many fonts loaded in this pad.

PAD_$VOOR
Value out of range.

PBUFS

This section describes the error messages and the call syntax for the PBUFS programming calls. The PBUFS calls do not use unique data types. Refer to the Introduction at the beginning of this manual for a description of call syntax format.

PBUFS_$CREATE

Creates a paste buffer.

FORMAT

```
PBUFS_$CREATE (buffer-name, type, stream-id, status)
```

INPUT PARAMETERS

buffer-name

Name of the paste buffer you want to create (not a pathname), in NAME_$NAME_T format. This is an array of up to 32 characters. This array must be a full 32 bytes, padded with blanks. See the NAME_$ Data Types section for more information.

type

Indicates whether the paste buffer is to hold text or pictures. This is a Boolean value. TRUE designates a text buffer. FALSE designates a GMF buffer that can hold images.

OUTPUT PARAMETERS

stream-id

Number of a stream with which to refer to the new paste buffer, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD_$ Data Types section for more information.

USAGE

This call creates a paste buffer of the specified name and type in the directory 'node_data/paste_buffers. An error occurs if the named paste buffer already exists in /sys/node_data/paste_buffers.

The file has the temporary attribute, STREAM_$IRM_TEMPORARY. The system will delete this file when you close the stream, unless you call STREAM_$REDEFINE to change the file's attributes first.

Calling PBUFS_$CREATE opens the stream for overwrite access (STREAM_$OVERWRITE).

You can call STREAM_$CREATE, specifying a pathname in /sys/node_data/paste_buffers to achieve the same effect.

PBUFS_$OPEN

Opens a pre-existing paste buffer.

FORMAT

```
PBUFS_$OPEN (buffer-name, type, stream-id, status)
```

INPUT PARAMETERS

buffer-name

Name of the paste buffer you want to open (not a pathname), in NAME_$NAME_T format. This is an array of up to 32 characters. This array must be a full 32 bytes, padded with blanks. See the NAME_$ Data Types section for more information.

type

Indicates whether the paste buffer is to hold text or pictures. This is a Boolean value. TRUE designates a text buffer. FALSE designates a GMF buffer that can hold images. The value you specify must match the value used when creating the paste buffer, or the paste buffer manager returns the completion status PBUFS_$WRONG_TYPE.

OUTPUT PARAMETERS

stream-id

Number of the stream with which to refer to the paste buffer, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PAD_$ Data Types section for more information.

USAGE

This call open a pre-existing paste buffer of the specified name and type in the directory ‘node_data/paste_buffers.

An error occurs if the named paste buffer does not already exist in /sys/node_data/paste_buffers. Use PBUFS_$CREATE to create a buffer.

You can call STREAM_$OPEN, on a file in /sys/node_data/paste_buffers, to achieve the same effect.

ERRORS

PBUFS_$WRONG_TYPE

The actual buffer type differs from the type specified.

PFM

This section describes the data types, the call syntax, and the error codes for the PFM programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

PFM_$ALL_FAULTS	0	Specified when establishing a handler to catch all faults.

DATA TYPES

PFM_$CLEANUP_REC

Cleanup routine information. The diagram below illustrates the PFM_$CLEANUP_REC data type:

predefined type	byte: offset		field name
	0:	integer	magicp
	4:	integer	checkp

Field Description:

magicp
A pointer used by the fault manager.

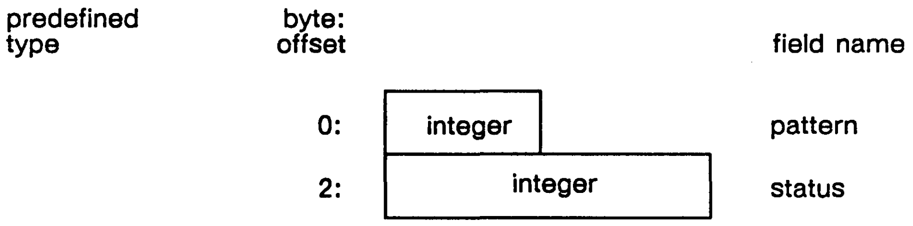
checkp
A pointer used by the fault manager.

PFM_$FAULT_FUNC_P_T

A 4-byte integer. A pointer to a fault handler function.

PFM_$FAULT_REC_T

Parameter to fault handler function. The diagram below illustrates the PFM_$FAULT_REC_T data type:



Field Description:

pattern
Reserved for PFM use.

status
The returned status in STAUTS_$T format.

PFM_$FH_FUNC_VAL_T

A 2-byte integer. Specifies action to be taken when handler completes. One of the following pre-defined values:

PFM_$CONTINUE_FAULT_HANDLING
Specifies that the fault be passed to next handler.

PFM_$RETURN_TO_FAULTING_CODE
Specifies that control be returned to the program.

PFM_$FH_HANDLE_T

A 4-byte integer. Pointer to a fault handler.

PFM_$FH_OPT_SET_T

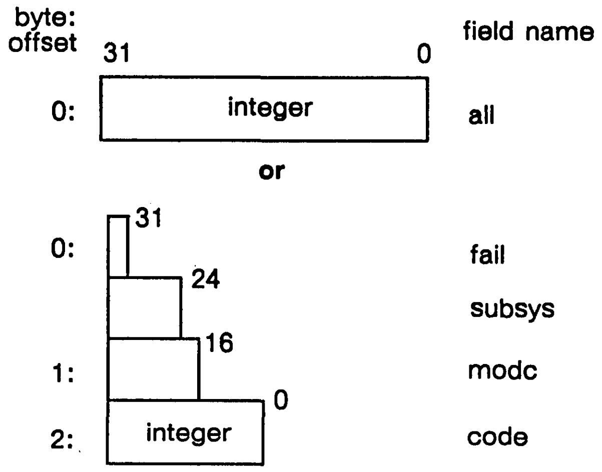
A 2-byte integer. Options for type of handler to establish. Any combination of the following pre-defined values:

PFM_$FH_BACKSTOP
specifies that the handler should be called after all other handlers.

PFM_$FH_MULTI_LEVEL
Specifies that handler applies to faults on its program level, and all subordinate levels.

STATUS _ $T

A status code. The diagram below illustrates the STATUS _ $T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

PFM_$CLEANUP

Establishes a clean-up handler for faults.

FORMAT

```
status = PFM_$CLEANUP (clean-up-record)
```

RETURN VALUE

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

When initially called to establish a clean-up handler, PFM_$CLEANUP returns the status PFM_$CLEANUP_SET. After a fault occurs, PFM_$CLEANUP returns the status of the fault, or the status signaled by PFM_$SIGNAL or PFM_$ERROR_TRAP.

OUTPUT PARAMETERS

clean-up-record

A record uniquely identifying the clean-up handler, in PFM_$CLEANUP_REC format. This data type is 8 bytes long. See the PFM Data Types section for more information.

This parameter is passed as input to the PFM_$RLS_CLEANUP and PFM_$RESET_CLEANUP procedures in order to specify a particular handler. Your program cannot modify or copy this value.

USAGE

PFM_$CLEANUP establishes a clean-up handler that is executed when a fault occurs. Clean-up handlers let the program "clean up" a task, possibly notifying you of the error condition and leaving any open files in a known and stable state.

You may establish more than one clean-up handler. Multiple cleanup handlers are executed consecutively, starting with the most recently established handler and continuing backward in time (LIFO). A built-in clean-up handler is always established when you invoke your program. This built-in handler is always called last. It closes any files that are still open and returns control to the invoking Shell.

The initial call to PFM_$CLEANUP establishes the clean-up handler and returns a status value of PFM_$CLEANUP_SET. When a fault occurs, execution returns to the most recent PFM_$CLEANUP call. The clean-up handler The associated with that call is then removed from the stack and executed.

PFM_$ENABLE

Enables asynchronous faults.

FORMAT

```
PFM_$ENABLE
```

USAGE

PFM_$ENABLE enables asynchronous faults after they have been inhibited by a call to PFM_$INHIBIT. PFM_$ENABLE causes the operating system to pass asynchronous faults on to the program.

While faults are inhibited, the operating system holds at most one asynchronous fault. So, as soon as a PFM_$ENABLE executes, your program receives one asynchronous fault. If more than one fault occurred while faults were inhibited, the program receives the *first* asynchronous fault.

Since a user cannot terminate a program while PFM_$INHIBIT is in effect, it is good programming practice to inhibit asynchronous faults only during critical intervals, or enable faults occasionally to allow users to exit.

PFM_$ERROR_TRAP

Simulates a fault with a given status code, storing traceback information.

FORMAT

```
PFM_$ERROR_TRAP (status)
```

INPUT PARAMETERS

status

Error code, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

USAGE

Use this procedure to force an error exit with the specified status code, or in a fatal error situation where no status code otherwise returns. One possible use is in defining your own error condition.

This procedure differs from PFM_$SIGNAL in that traceback information is stored, so that it is possible to determine where the fault occurred.

PFM_$ESTABLISH_FAULT_HANDLER

Establishes a fault handler.

FORMAT

```
handler-id = PFM_$ESTABLISH_FAULT_HANDLER (target-status, options,
                                            function-pointer, status)
```

RETURN VALUE

handler-id

A value uniquely identifying the established handler, in PFM_$FH_HANDLE_T format. This is a 4-byte integer.

You pass this value to the PFM_$RELEASE_FAULT_HANDLER call when you want to release the handler.

INPUT PARAMETERS

target-status

A value specifying the type of fault that COMMENTs this handler takes effect. This is a 4-byte integer.

To establish a fault handler for all faults produced by a certain DOMAIN module, use any error status code returned by that module, with the fault code field set to 0. To establish a fault handler that handles all faults, use the constant PFM_$ALL_FAULTS.

options

A value specifying the type of handler you want to establish, in PFM_$FH_OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following set of predefined values:

PFM_$FH_MULTI_LEVEL
To declare a multilevel fault handler that handles faults for its own program level and all subordinate levels.

PFM_$FH_BACKSTOP
To establish a backstop fault handler that takes effect after all nonbackstop handlers have taken effect.

(In FORTRAN, you can combine these options by adding the constants.)

function-pointer

The address of the fault handler for the specified type(s) of faults, in PFM_$FAULT_FUNC_P_T format. This is a 4-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

This call establishes a fault handler, making it take effect for all the faults of the specified type or types that occur after the time of the call.

The fault handler remains in effect until you release it using PFM_$RELEASE_FAULT_HANDLER or until the program ends.

PFM_$INHIBIT

Inhibits asynchronous faults.

FORMAT

```
PFM_$INHIBIT
```

USAGE

PFM_$INHIBIT prevents asynchronous faults from being passed to the program. Use this call when an interval of your program cannot be interrupted, for example, when performing I/O. Use the complementary PFM_$ENABLE call to re-enable asynchronous faults.

Asynchronous faults are produced from outside your program and are unrelated to anything within your program. They can occur at any point during your program's execution. A common example of an asynchronous fault is the Display Manager quit (DQ) command that occurs when someone types a CTRL/Q to stop a program.

Since a user cannot terminate a program while PFM_$INHIBIT is in effect, it is good programming practice to inhibit asynchronous faults only during critical intervals.

While faults are inhibited, the operating system holds at most one asynchronous fault. So, as soon as a PFM_$ENABLE executes, your program receives one asynchronous fault. If more than one fault occurred while faults were inhibited, the program receives the *first* asynchronous fault.

Inhibiting asynchronous faults has no effect on the processing of synchronous faults such as floating-point overflow errors, access violations, address errors, and so on.

PFM_$RELEASE_FAULT_HANDLER

Releases a fault handler.

FORMAT

```
PFM_$RELEASE_FAULT_HANDLER (handler-id, status)
```

INPUT PARAMETERS

handler-id

A value uniquely identifying the handler, in PFM_$FH_HANDLE_T format. This is a 4-byte integer.

A unique value is returned by PFM_$ESTABLISH_FAULT_HANDLER each time you establish a handler.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

USAGE

This call COMMENTs the specified fault handler ceasing to have effect for faults that occur after the time of the call.

To establish a fault handler, use the PFM_$ESTABLISH_FAULT_HANDLER call.

PFM_$RESET_CLEANUP

Returns a clean-up handler to the top of the handler stack.

FORMAT

```
PFM_$RESET_CLEANUP (clean-up-record, status)
```

INPUT PARAMETERS

clean-up-record

A record uniquely identifying the clean-up handler, in PFM_$CLEANUP_REC format. This data type is 8 bytes long. See the PFM Data Types section for more information.

A unique record is returned by PFM_$CLEANUP each time a cleanup handler is established. The clean-up-record that is input must not have been altered or copied. If it has been, or if for some other reason the record is invalid, the procedure will fail with the status PFM_$INVALID_CLEANUP_REC.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

USAGE

This procedure re-establishes the clean-up handler identified by the clean-up-record at the top of the stack, so that any subsequent errors invoke it first.

This procedure can only be used within a clean-up handler.

PFM_$RLS_CLEANUP

Releases a specified clean-up handler and any other clean-up handlers above it on the stack.

FORMAT

```
PFM_$RLS_CLEANUP (clean-up-record, status)
```

INPUT PARAMETERS

clean-up-record

A record uniquely identifying the clean-up handler, in PFM_$CLEANUP_REC format. This data type is 8 bytes long. See the PFM Data Types section for more information.

A unique record is returned by PFM_$CLEANUP each time a clean-up handler is established. The clean-up-record that is input must not have been altered or copied. If it has been, or if for some other reason the record is invalid, the procedure will fail with the status PFM_$INVALID_CLEANUP_REC.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

Possible status values are:

PFM_$INVALID_$CLEANUP_REC
: The clean-up-record has been altered or copied and is therefore invalid.

PFM_$BAD_RLS_ORDER
: Program attempting to release a clean-up handler before releasing all handlers established after it. This status is only a warning; the handler is successfully released, and all handlers above it on the stack are also released.

USAGE

PFM_$RLS_CLEANUP releases the specified clean-up handler and all other clean-up handlers above it on the stack.

PFM_$SIGNAL

Exits from the current procedure and signals a status for the clean-up handler on the top of the stack.

FORMAT

```
PFM_$SIGNAL (status)
```

INPUT PARAMETERS

status
Status code, in STATUS_$T format. This data type is 4 bytes long. See the PFM Data Types section for more information.

USAGE

PFM_$SIGNAL can be called from within a clean-up handler or from normal code.

If invoked from within a clean-up handler, PFM_$SIGNAL exits from the current clean-up handler and invokes the clean-up handler on the top of the stack, if there is one. If invoked from outside a clean-up handler, this routine invokes the top clean-up handler on the stack, with the status code given in the PFM_$SIGNAL call.

Typically, PFM_$SIGNAL is called at the end of one clean-up handler to invoke the next handler, and the status parameter is normally assigned the error status originally received from PFM_$CLEANUP. When no more clean-up handlers from the current program are on the stack, PFM_$SIGNAL causes the program to exit to the invoking program (which may be the Shell) with the status code set to the value given in the status parameter.

Traceback information (see the *DOMAIN System Command Reference*) is not stored when PFM_$SIGNAL is called. When a fault occurs, however, the operating system automatically stores traceback information.

Unlike most subroutines, PFM_$SIGNAL does not return to the place from which it was called.

ERRORS

STATUS_$OK
Successful completion.

PFM_$BAD_RLS_ORDER
Cleanup handler released out of order.

PFM_$CLEANUP_NOT_FOUND
Static cleanup handler not found.

PFM_$CLEANUP_SET
Cleanup handler established successfully.

PFM_$CLEANUP_SET_SIGNALLED
PFM_$CLEANUP_SET was signalled.

PFM_$FH_NOT_FOUND
Attempt to release non-existent fault handler.

PFM_$FH_WRONG_LEVEL
Attempt to release fault handler at wrong level.

PFM_$INVALID_CLEANUP_REC
Invalid clean-up record.

PFM_$NO_SPACE
No RWS space to create static clean-up handler.

PGM

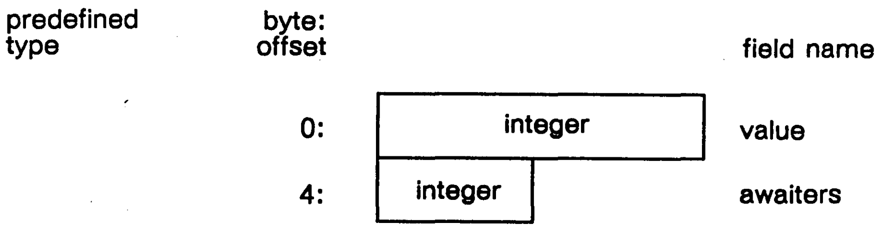
This section describes the data types, the call syntax, and the error codes for the PGM programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

PGM_$ERROR	3	The error severity level.
PGM_$FALSE	1	A test severity level.
PGM_$INTERNAL_FATAL	5	The fatal severity level.
PGM_$MAX_SEVERITY	15	The highest severity level.
PGM_$OK	0	The success severity level.
PGM_$OUTPUT_INVALID	4	A conditional severity level.
PGM_$PROGRAM_FAULTED	6	The program fault severity level.
PGM_$TRUE	0	A test severity level.
PGM_$WARNING	2	The warning severity level.

DATA TYPES

EC_$PTR_T	A 4-byte integer. Pointer to an eventcount.
EC2_$EVENTCOUNT_T	User eventcount. The diagram below illustrates the EC2_$EVENTCOUNT_T data type:



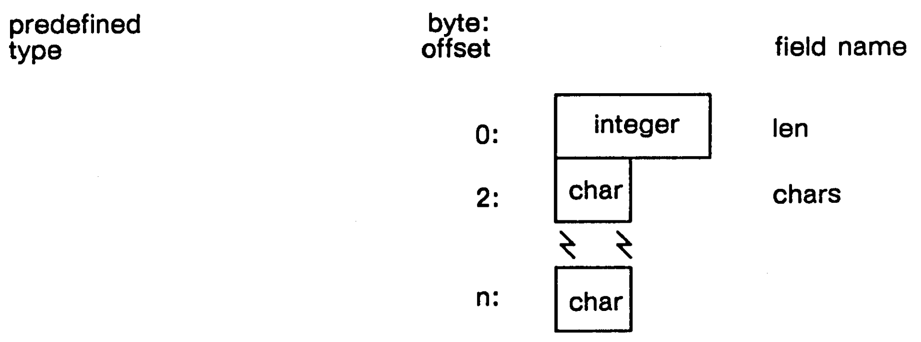
Field Description:

value
Current EC value.

awaiters
First process waiting.

PGM_$ARG

An argument returned by PGM_$GET_ARGS. The diagram below illustrates the PGM_$ARG data type:



Field Description:

len
Length of the argument.

chars
The text of the argument, a character array of up to 128 elements.

PGM_$ARGV	A 128-element array of 4-byte integers. An array of pointers to returned arguments.
PGM_$ARGV_PTR	A 4-byte integer. The address of a returned argument.
PGM_$CONNV	A 128-element array of 2-byte integers. An array of stream IDs.
PGM_$EC_KEY	A 2-byte integer. Key specifying process eventcount. One of the following pre-defined values: PGM_$CHILD_PROC Currently the only valid key.
PGM_$MODE	A 2-byte integer. Specifies the mode in which to invoke a program. Any combination of the following pre-defined values:
PGM_$NAME	An array of up to 128 characters. The text of a retrieved argument.

PGM_$OPTS

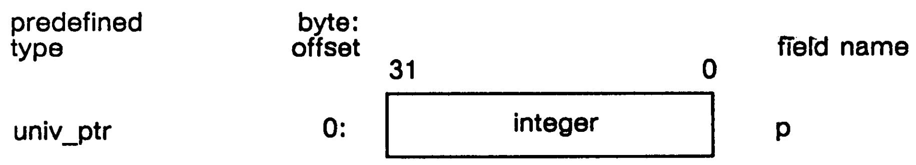
A 2-byte integer. Options for the mode in which to invoke a program. One of the following pre-defined values:

PGM_$WAIT
Specifies synchronous operation of the invoked program.

PGM_$BACKGROUND
Specifies parallel operation of the invoked process.

PGM_$PROC

Process handle record. The diagram below illustrates the PGM_$PROC data type:

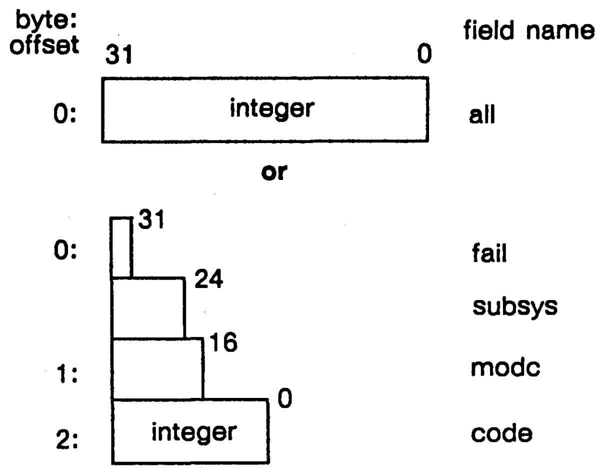


Field Description:

p
The process pointer.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

UID_$T

A type UID. The diagram below illustrates the UID_$T data type:

predefined type	byte: offset		field name
	0:	integer	high
	4:	integer	low

Field Description:

high
The high four bytes of the UID.

high
The low four bytes of the UID.

PGM_$DEL_ARG

Deletes a command line argument.

FORMAT

```
PGM_$DEL_ARG (arg-number)
```

INPUT PARAMETERS

arg-number
Number indicating the argument to delete. This is a 2-byte integer.

USAGE

PGM_$DEL_ARG deletes the specified argument from the argument vector whose address is returned by PGM_GET_ARGS. After execution of PGM_$DEL_ARGS, the previously returned address refers to the newly changed argument vector.

Arguments in the argument vector are numbered 0 through n, where 0 is the program name, and n is the final argument. Because PGM_$DEL_ARGS changes the argument vector, arguments following deleted arguments change in number. For example, say the argument vector contains six arguments (including the program name). After you delete the third argument, arguments 4, 5, and 6 must be referenced as arguments 3, 4, and 5.

PGM_$EXIT

Exits from a program to its caller.

FORMAT

```
PGM_$EXIT
```

USAGE

PGM_$EXIT can be used to exit from a program at any point and return to the program's caller.

PGM_$EXIT differs from a simple exit (for example, via FORTRAN's END statement) in that PGM_$EXIT is valid in a subroutine. Execution in a subroutine terminates the main program. FORTRAN's STOP statement, which can be used in main programs and subprograms, calls PGM_$EXIT.

When PGM_$EXIT is executed, any files left open by the program are closed, any storage acquired is released, and the inhibit count is reset to its value when the program was invoked.

PGM_$EXIT calls PFM_$SIGNAL with a status code equal to the last severity level set by a call to PGM_$SET_SEVERITY. If no PGM_$SET_SEVERITY calls have been made, the status code is PGM_$OK. PFM_$SIGNAL signals this severity to any established clean-up handlers, which normally execute in response to any status code other than PFM_$CLEANUP_SET. Therefore, any established clean-up routines are normally executed after PGM_$EXIT is called.

PGM_$GET_ARG

Returns one argument from the command line.

FORMAT

```
arg-length = PGM_$GET_ARG (arg-number, argument, status, maxlen)
```

RETURN VALUE

arg-length
Length, in bytes, of the returned argument. This is a 2-byte integer.

INPUT PARAMETERS

arg-number
Number of the argument to return. This is a 2-byte integer.

OUTPUT PARAMETERS

argument
String of length arg-length, containing the requested argument, in PGM_$NAME format. This is an array of up to 128 characters.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

INPUT PARAMETERS

maxlen
Maximum length of the argument, in bytes. This is a 2-byte integer.

FORTRAN automatically passes the length of a character string following the string itself. Therefore, to return a character string argument to a FORTRAN program, omit the maxlen parameter. Use the following format for the call:

```
arg-length = PGM_$GET_ARG (arg-number, argument, status)
```

This format applies to character strings only. For an argument of any other type, use the standard call.

If the value of maxlen is less than the returned argument length, the program manager truncates the returned argument to maxlen bytes and returns the status PGM_$ARG_TOO_BIG.

USAGE

PGM_$GET_ARG returns one argument from the program's caller. The argument is in character string format.

Argument numbers on the command line range from 0 to n. Argument 0 is the program name.

PGM_$GET_ARGS

Returns the address of the argument vector.

FORMAT

```
PGM_$GET_ARGS (argument-count, arg-vector-addr)
```

OUTPUT PARAMETERS

argument-count
Number of arguments in the argument vector. This is a 2-byte integer.

arg-vector-addr
Address of the argument vector, in PGM_$ARGV_PTR format. This is a 4-byte integer.

USAGE

PGM_$GET_ARGS returns the address of the argument vector.

The argument vector is an array of addresses pointing to the arguments. This array can be up to 128 elements.

The addresses are in PGM_$ARGV format. This is a 4-byte integer. See the PGM Data Types section for more information.

PGM_$GET_EC

Gets an eventcount to wait for completion of a child process.

FORMAT

```
PGM_$GET_EC (process-handle, process-key, eventcount-pointer, status)
```

INPUT PARAMETERS

process-handle

Process handle of the child process for which to wait, in PGM_$PROC format. This data type is 4 bytes long. See the PGM Data Types section for more information.

The process handle is returned by PGM_$INVOKE when you create a process. Note that the process handle is valid only when you invoke the program in default mode.

process-key

Key specifying which process eventcount the system should return, in PGM_$EC_KEY format. This is a 2-byte integer.

Currently the only allowable value is PGM_$CHILD_PROC.

OUTPUT PARAMETERS

eventcount-pointer

The eventcount address to be obtained, in EC2_$PTR_T format. This is a 4-byte integer.

EC2_$PTR_T is a pointer to an EC2_$EVENTCOUNT_T record. See the EC2 Data Types section for more information.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

USAGE

PGM_$GET_EC returns a pointer to an eventcount that advances when a child process terminates. This eventcount address can be passed to EC2_$WAIT to wait for a specific child process to complete. You identify the child process by passing the process handle as an input parameter.

When a child process is created, the process eventcount value is 0. When a child process terminates, the process eventcount value is 1. To wait on a specific child process, you might use:

```
PGM_$PROC_EC (....gets process event count ....)
EC2_$WAIT    (....waits until eventcount is 1 ....)
```

See the Managing Programs Chapter of the *Programming With General System Calls* manual for more information.

PGM_$GET_PUID

Gets the process UID of a process.

FORMAT

```
PGM_$GET_PUID (process-handle, puid, status)
```

INPUT PARAMETERS

process-handle

Process handle of the child process for which you want a UID, in PGM_$PROC format. This data type is 4 bytes long. See the PGM Data Types section for more information.

The process handle is returned by PGM_$INVOKE when you create a process. Note that the process handle is valid only when you invoke the program in default mode.

OUTPUT PARAMETERS

puid

Process UID, in UID_$T format. This data type is 8 bytes long. See the PGM Data Types section for more information.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

USAGE

PGM_$GET_PUID, which returns the process UID of a child process.

PGM_$GET_PUID, which is used in conjunction with other system calls. These calls are:

- PROC2_$GET_INFO, which returns information about a process given a PUID.
- PROC2_$LIST, which returns a list of the PUIDs of all active user processes.
- PGM_$MAKE_ORPHAN, which returns the PUID of the orphaned process.

PGM_$INVOKE

Invokes a program.

FORMAT

```
PGM_$INVOKE (pathname, namelength, arg-count, arg-vector, stream-count,
            connection-vector, mode, process-handle, status)
```

INPUT PARAMETERS

pathname

Pathname of the program to invoke, in NAME_$PNAME_T format. This is an array of up to 256 characters.

The specified pathname must be an absolute pathname; the Shell's search rules do not apply.

namelength

Length of the pathname, in bytes. This is a 2-byte integer.

arg-count

Number of arguments to pass to the invoked program. This is a 2-byte integer.

This number corresponds to the number of elements in the argument vector.

arg-vector

Array containing the addresses of the arguments to pass to the invoked program, in PGM_$ARGV format. This is an array of 4-byte integers.

A program can pass any number of arguments to a program it is invoking. However, when passing arguments to a Shell, the Shell's syntax limits the number of arguments to 10 (including the program name). Each argument must be preceded by a 2-byte integer indicating the number of bytes in the argument. The first argument must be the name of the program; the simple name, not the absolute pathname (that is, date, *not* //desperado/com/date). Note that if the invoked program calls PGM_$DEL_ARG, the argument vector changes. See the description of PGM_$DEL_ARG for details.

stream-count

Number of streams to pass to the invoked program. This is a 2-byte integer.

You are permitted to pass up to 32 streams. In the invoked program these streams are numbered 0 to 31.

connection-vector

Array containing stream IDs to pass to the invoked program, in PGM_$CONNV format. Each stream ID is a 2-byte integer, in STREAM_$ID_T format. Up to 128 elements are permitted.

By default, every program is invoked with four streams, numbered 0 through 3. Stream 0 is standard input, stream 1 is standard output, Stream 2 is error input, stream 3 is error output.

Stream IDs refer to objects already opened by the calling program, using

STREAM_$CREATE or STREAM_$OPEN. The first element in the connection-vector array becomes stream 0 in the invoked program, the second element becomes stream 1, and so on.

You may leave "holes" in the connection vector by setting a stream ID equal to the predefined constant STREAM_$NO_STREAM.

mode

Mode in which to invoke the program, in PGM_$MODE format. This is a 2-byte integer. Specify a null set, or one of the following predefined values:

PGM_$WAIT	The program executes as a separate program within the same process as the invoking program.
PGM_$BACK_GROUND	The program executes as a separate process that runs to termination independently of the invoking process.

If you pass a null set (default), the program executes as a separate process that communicates its termination status to the invoking program. To specify a null set in C and FORTRAN, declare the variable and initialize it to 0.

OUTPUT PARAMETERS

process-handle

Process handle of the process in which the invoked program runs, in PGM_$PROC format. This data type is 4 bytes long. See the PGM Data Types section for more information.

The process handle is used as an input parameter in the PGM_$GET_EC, PGM_$PROC_WAIT, PGM_$GET_PUID, and PGM_$MAKE_ORPHAN calls to identify an invoked program.

Note that the process handle is valid only after creating a process in default mode. You will get an error (for example, 'reference to illegal address') if you attempt to use the process handle of a process created in background mode. The following calls use the process handle: PGM_$GET_EC, PGM_$GET_PUID, PGM_$MAKE_ORPHAN or PGM_$PROC_WAIT.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

Possible values are:

STATUS_$OK	Success status.
PGM_$BAD_CONNV	Stream vector too large (>32).

Severity level values returned by the program:

PGM_$TRUE	Value of tested condition is true.

PGM_$FALSE	Value of tested condition is false.
PGM_$WARNING	Unusual, but not fatal condition detected.
PGM_$ERROR	Syntactic or semantic errors in input; output is structurally sound.
PGM_$INVALID_OUTPUT	Syntactic or semantic errors in input; output is not structurally sound.
PGM_$INTERNAL_FATAL	Internal fatal error detected.

Any status returned by the invoked program.

Any status returned by modules that PGM_$INVOKE calls.

USAGE

PGM_$INVOKE invokes a program in the specified mode, and passes that program any parameters that it needs. The addresses of arguments are passed to the invoked program by way of the arg-vector, which is an array of those addresses. The invoked program uses the system routines, PGM_$GET_ARGS, PGM_$GET_ARG, and PGM_$DEL_ARG to access the arguments. See the documentation of those routines for details.

You can change standard input for the invoked program by opening the desired input file and passing its stream ID as the first element of the connection vector. The same is true for standard output, standard error input, and standard error output.

When the invoked program finishes executing, files it has opened are closed, storage it has acquired is released, and the inhibit count is the same as it was upon entry.

The behavior of an invoked program differs depending on the mode in which the program is invoked.

Invoking a Program in Wait Mode

When you invoke a program this way, the invoking program executes the program and waits for it to complete before continuing.

A program invoked in wait mode calls PGM_$SET_SEVERITY to indicate its completion status to the invoking program.

A program ends when one of the following takes place:

- A language defined termination statement is executed
- An unhandled fault occurs

- You call PGM_$EXIT

Normal termination returns execution to the calling program. An unhandled error either terminates the program with an error status or invokes a clean-up handler. PGM_$EXIT invokes any established clean-up handlers, then exits to the calling program. Any severity levels set during program execution are returned in the status parameter.

Invoking a Program in Default Mode

When you invoke a program specifying a null set, the invoking program creates a new process in which to run the program. The invoking process may wait for the child process to complete and determine its termination status by calling PGM_$PROC_WAIT.

When a process invokes another process, the invoking process is referred to as the parent process and the invoked process is referred to as the child process. Executing a program in a child process is useful if you wish to perform concurrent processing or if your program requires a large amount of address space.

Waiting for a Child Process

The PGM_$GET_EC call permits you to get a process eventcount that is advanced when a specified process terminates. By using this call in conjunction with the system calls EC2_$READ and EC2_$WAIT, a parent process can wait for the completion of a child process (or a list of event counts).

Getting the Completion Status of a Child Process

Once a child process has completed, examine its completion status. To obtain the completion status of a default mode process, call PGM_$PROC_WAIT in the parent process. PGM_$PROC_WAIT takes the process handle of the invoked program as an input parameter and returns its completion status. If the child process has not completed execution at the time of the PROC_WAIT call, execution of the parent process suspends until a completion status is available.

A certain amount of resources in a parent process are used to keep track of a child process. When a call to PGM_$PROC_WAIT is completed those resources are released. If you invoke a number of child processes without ever calling PROC_WAIT, the parent process may run out of resources. If you are not interested in the completion status of the invoked program, invoke it using background mode.

Invoking a Program in Background Mode

When you invoke a program specifying PGM_$BACK_GROUND, the invoking program creates a new process in which to run the program. Background mode differs from default mode in that a background mode process runs completely independently of the parent. That is, there is no communication of the completion status.

Background mode is useful for performing processing that has no further dependence on the parent process. For example, a parent process may perform interactive data collection, invoke a program in a background process to manipulate the data, and then return to further data collection. This permits you to collect and manipulate the data concurrently.

Because a background mode process has no dependence on the parent, it is referred to as an orphan process. You can change a default child process into an orphan process by calling PGM_$MAKE_ORPHAN.

Note that the process handle is valid only after creating a process in default mode. You will get an error (for example, 'reference to illegal address') if you attempt to use the process handle of a process created in background mode. The following calls use the process handle: PGM_$GET_EC, PGM_$GET_PUID, PGM_$MAKE_ORPHAN or PGM_$PROC_WAIT.

PGM_$MAKE_ORPHAN

Changes a normal child process into an orphan process.

FORMAT

```
PGM_$MAKE_ORPHAN (process-handle, puid, status)
```

INPUT PARAMETERS

process-handle

Process handle of the child process to orphan, in PGM_$PROC format. This data type is 4 bytes long. See the PGM Data Types section for more information.

The process handle is returned by PGM_$INVOKE when you create a process. Note that the process handle is valid only when you invoke the program in default mode.

OUTPUT PARAMETERS

puid

Process UID, in UID_$T format. This data type is 8 bytes long. See the PGM Data Types section for more information.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

USAGE

PGM_$MAKE_ORPHAN changes the specified child process into an orphan process.

An orphan process is one that is run in PGM_$BACKGROUND mode. An orphan process runs independently of the parent process and no termination status is returned to the parent.

PGM_$PROC_WAIT

Waits for a process that has been created with PGM_$INVOKE to terminate and returns a completion status for the process.

FORMAT

```
PGM_$PROC_WAIT (process-handle, status)
```

INPUT PARAMETERS

process-handle
Process handle of the child process for which to wait, in PGM_$PROC format. This data type is 4 bytes long. See the PGM Data Types section for more information.

The process handle is returned by PGM_$INVOKE when you create a process. Note that the process handle is valid only when you invoke the program in default mode.

OUTPUT PARAMETERS

status
The child process completion status, in STATUS_$T format. This data type is 4 bytes long. See the PGM Data Types section for more information.

USAGE

PGM_$PROC_WAIT suspends the execution of a parent process until the completion of a specified child process. This call permits a child process to pass a completion status to the parent upon termination.

Using PGM_INVOKE in default mode (the empty set) and then calling PGM_$PROC_WAIT is equivalent to using PGM_$INVOKE in PGM_$WAIT mode.

PGM_$SET_SEVERITY

Sets the severity level for a program.

FORMAT

```
PGM_$SET_SEVERITY (severity-level)
```

INPUT PARAMETERS

severity-level

The severity level returned to the caller. This is a 2-byte integer. Specify only one of the following predefined values:

PGM_$OK	The program completed successfully and performed the requested action.
PGM_$TRUE	The program completed successfully; its purpose was to test a condition, and the value of that condition was TRUE.
PGM_$FALSE	The program completely successfully; its purpose was to test a condition, and the value of that condition was FALSE.
PGM_$WARNING	The program completed successfully and performed the requested action. However, an unusual (but nonfatal) condition was detected.
PGM_$ERROR	The program could not perform the requested action because of syntactic or semantic errors in the input. The output is structurally sound, however.
PGM_$OUTPUT_INVALID	The program could not perform the requested action because of syntactic or semantic errors in the input, and the output is not structurally sound.
PGM_$INTERNAL_FATAL	The program detected an internal fatal error and ceased processing. The state of the output is neither defined nor guaranteed.
PGM_$PROGRAM_FAULTED	The program detected and handled a fault.

Severity levels are a subset of the general system status codes.

USAGE

Every program returns a severity level to its caller. By default, the severity level is PGM_$OK. Use PGM_$SET_SEVERITY in the invoked program to change the level to another value.

The following are examples of appropriate changes to the severity level:

PGM_$TRUE or PGM_$FALSE would be returned by an "equal" program that compares its two arguments to see if they are equal.

PGM_$WARNING would be returned by DLF (DELETE_FILE) if the file to be deleted did not exist.

PGM_$ERROR would be returned by a compiler if the input program contained an error that prevented a correct translation, but the output object module format was correct.

PGM_$OUTPUT_INVALID would be returned by a compiler if an error in the input program caused the object module format to be invalid.

PGM_$INTERNAL_FATAL would be returned if the program could not proceed because it detected that its data structures were corrupted.

PGM_$PROGRAM_FAULTED would be returned if the program signaled a fault and wishes to inform the invoking program without resignalling the fault.

ERRORS

STATUS_$OK
: Successful completion.

PGM_$ERROR
: The program could not perform the requested action because of syntactic or semantic errors in the input. The output is structurally sound, however.

PGM_$FALSE
: The program completely successfully; its purpose was to test a condition, and the value of that condition was FALSE.

PGM_$INTERNAL_FATAL
: The program detected an internal fatal error and ceased processing. The state of the output is neither defined nor guaranteed.

PGM_$OK
: The program completed successfully and performed the requested action.

PGM_$OUTPUT_INVALID
: The program could not perform the requested action because of syntactic or semantic errors in the input, and the output is not structurally sound.

PGM_$PROGRAM_FAULTE
: The program faulted.

PGM_$TRUE
: The program completed successfully; its purpose was to test a condition, and the value of that condition was TRUE.

PGM_$WARNING
: The program completed successfully and performed the requested action. However, an unusual (but non-fatal) condition was detected.

PM

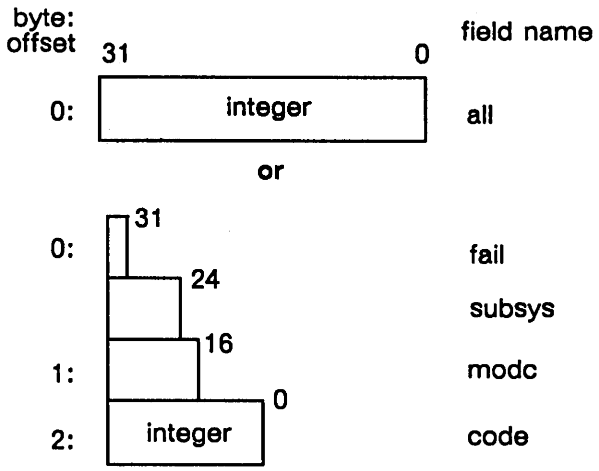
This section describes the data types and the call syntax for the PM programming calls. The PM calls do not use produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

NAME_$PNAMLEN_MAX	256	Maximum length of a pathname.

DATA TYPES

NAME_$PNAME_T	An array of up to NAME_$PNAMLEN_MAX (256) characters.
STATUS_$T	A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

PM_$GET_HOME_TXT

Returns the home directory of the calling process as a string.

FORMAT

```
PM_$GET_HOME_TXT (maxlen, home, len)
```

INPUT PARAMETERS

maxlen
Maximum number of characters to be returned (at most, the size of the buffer you assign for home). This is a 2-byte positive integer. This parameter need not exceed 256.

OUTPUT PARAMETERS

home
Pathname of the home directory for the SID (log-in identifier) of this process. This is an array of up to 256 characters.

len
Number of characters returned in the home parameter. This is a 2-byte positive integer.

USAGE

The home directory is obtained from the network registry when you log in and is inherited by all your processes.

PM_$GET_SID_TXT

Returns the SID (log-in identifier) of the calling process as a string.

FORMAT

```
PM_$GET_SID_TXT (maxlen, sid, len)
```

INPUT PARAMETERS

maxlen

Maximum number of characters to be returned (at most, the size of the buffer you assign for home). This is a 2-byte positive integer. This parameter need not exceed 140.

OUTPUT PARAMETERS

sid

String containing the person, project, organization and node ID of the SID (log-in identifier) of this process, in the form:

```
person.group.project.nodeid
```

This is an array of up to 140 characters.

len

Number of characters returned in the log-in identifier. This is a 2-byte positive integer.

USAGE

Your SID is the full identifier obtained from the network registry when you log in and is inherited by all your processes.

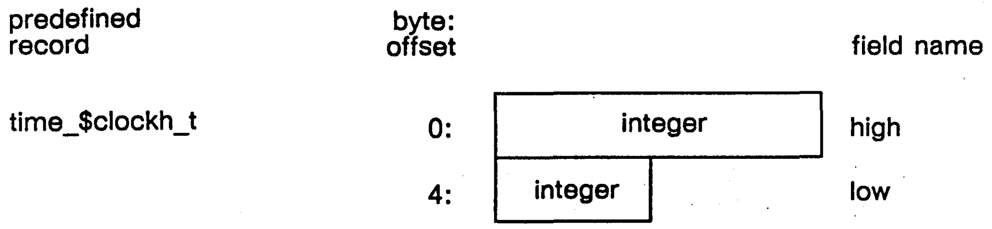
PROC1

This section describes the data types and the call syntax for the PROC1 programming calls. The PROC1 calls do not produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

TIME_$CLOCK_T

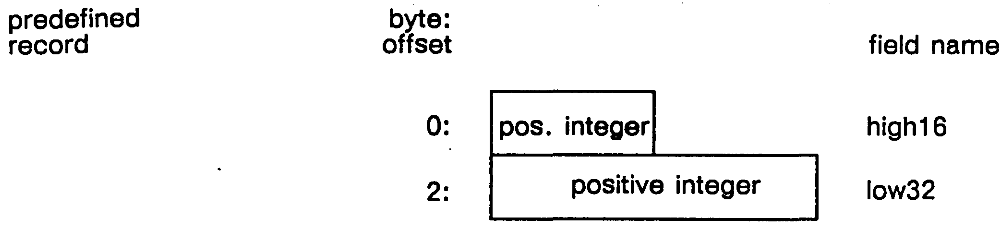
Internal representation of time. The diagram below illustrates the TIME_$CLOCK_T data type:



Field Description:

high
High 32 bits of the clock.

low
Low 16 bits of the clock.



Field Description:

high16
High 16 bits of the clock.

low32
Low 32 bits of the clock.

PROC1_$GET_CPUT

Returns the CPU time used by this process.

FORMAT

```
PROC1_$GET_CPUT (clock)
```

OUTPUT PARAMETERS

clock

The amount of CPU time used by this process since its creation, in TIME_$CLOCK_T format. This data type is 6 bytes long. See the TIME Data Types section for more information.

USAGE

PROC1_$GET_CPUT returns the amount of CPU time that the calling process has used since its creation. The returned clock value has a resolution of 4 microseconds.

CPU time is the time during which the process is running in the CPU. This includes the time that the operating system is performing services for the process, but does not include the time that the process spends waiting for I/O transfers to complete.

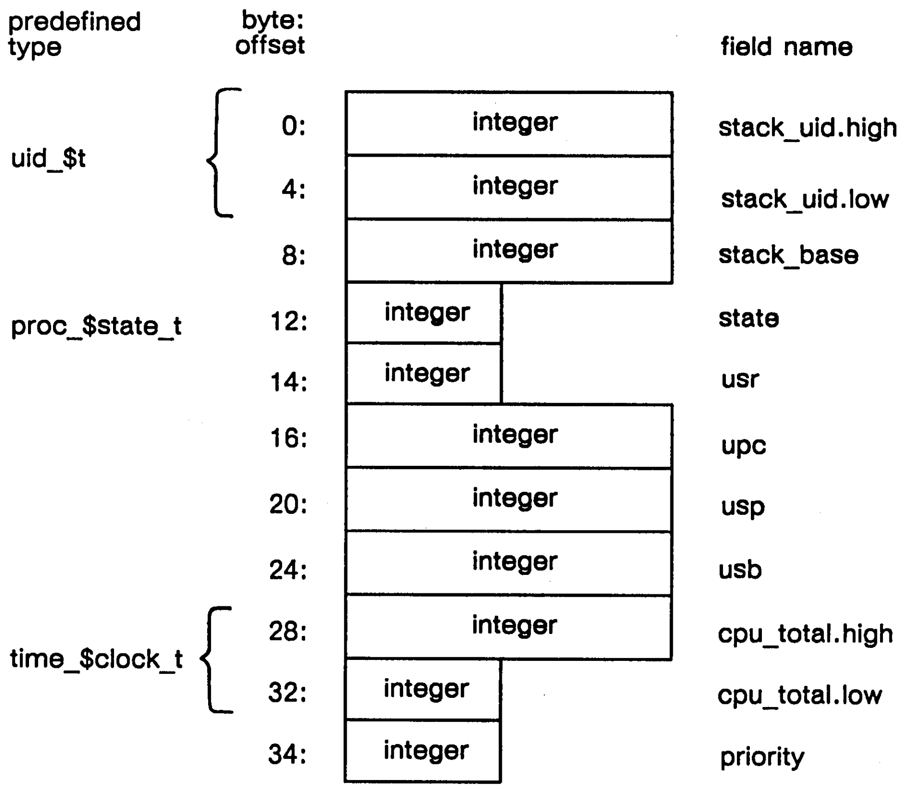
PROC2

This section describes the data types, the call syntax, and the error codes for the PROC2 programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

PROC2_$INFO_T

Process information record. The diagram below illustrates the PROC2_$INFO_T data type:



Field Description:

stack_uid
Uid of user stack.

stack_base
Base address of user stack.

state
Process state - ready, waiting, etc..

usr
User status register.

upc
User program counter.

usp
User stack pointer.

usb
User stack base pointer (A6).

cpu_total
Cumulative cpu time used by process.

priority
Process priority.

PROC2_$UID_LIST_T	An array of UIDs (in UID_$T format) of up to 24 elements.
PROC2_$STATE_T	A 2-byte integer. State of a user process. Any combination of the following pre-defined values:

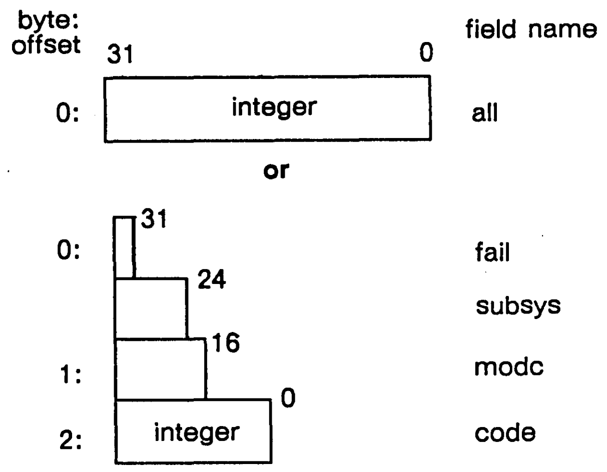
PROC2_$WAITING
Process is waiting.

PROC2_$SUSPENDED
Process is suspended.

PROC2_$SUSP_PENDING
Process suspension is pending.

PROC2_$BOUND
Process is bound.

STATUS_$T — A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

PROC2_$GET_INFO

Returns information about a process.

FORMAT

```
PROC2_$GET_INFO (process-uid, info, info-buf-length, status)
```

INPUT PARAMETERS

process-uid
The UID of the process for which you want information, in UID_$T format. This data type is 8 bytes long. See the PROC2 Data Types section for more information.

You can get process UIDs by calling PROC2_$WHO_AM_I and PROC2_$LIST.

If the process-uid in the call is the caller's own process, the only information returned is the stack UID and virtual address. If you want to find out the amount of CPU time used by the caller's process, use PROC1_$CPU_TIME.

info-buf-length
Length of the information buffer allotted for returned information, in bytes. This is normally 36 bytes.

OUTPUT PARAMETERS

info
Information about the process, in PROC2_$INFO_T format. This data type is 36 bytes long. See the PROC2 Data Types section for more information.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PROC2 Data Types section for more information. Possible values are:

STATUS_$OK	Completed successfully.
PROC2_$IS_CURRENT	Specified calling process UID (success).
PROC2_$UID_NOT_FOUND	Specified UID is not on node.

USAGE

GET_$INFO returns information about a process when supplied with a process UID. The information returned consists of the following:

- The program state (ready, waiting, suspended, SUSP_PENDING, bound).
- The User Status Register (USR).
- The User Program Counter (UPC).

- The user stack pointer (A7).
- The stack base pointer (A6).
- The amount of CPU time used.
- The CPU scheduling priority.

PROC2_$LIST

Returns a list of existing level 2 (user) processes in the caller's node.

FORMAT

```
PROC2_$LIST (uid-list, max-num-uids, number-uids)
```

OUTPUT PARAMETERS

uid-list
The UIDs of the active level 2 processes on the system, in PROC2_$UID_LIST_T format. This is a 24-element array of UIDs. Each UID is a 4-byte integer in UID_$T format.

INPUT PARAMETERS

max-num-uids
Maximum number of process UIDs to be returned. (At most, the size of the buffer you assign for uid-list. This is a 2-byte integer.

OUTPUT PARAMETERS

number-uids
Number of active level 2 processes on the node, even if that number is greater than max-num-uids. This is a 2-byte integer.

USAGE

The UIDs of all level 2 processes (user processes) on the caller's node, up to max-num-uids, are returned.

PROC2_$WHO_AM_I

Returns the UID of the calling prccess.

FORMAT

```
PROC2_$WHO_AM_I (my-uid)
```

OUTPUT PARAMETERS

my-uid
The UID of the calling process, in UID_$T format. This data type is 8 bytes long. See the PROC2 Data Types section for more information.

USAGE

You can use a UID obtained through this call to find out information about your process through the PROC2_$GET_INFO call.

ERRORS

STATUS_$OK
Successful completion.

PROC2_$BAD_STACK_BASE
Bad stack base.

PROC2_$IS_CURRENT
Request is for current process.

PROC2_$NOT_LEVEL_2
Not a level two process.

PROC2_$UID_NOT_FOUND
Process not found.

RWS

This section describes the data types, the call syntax, and the error codes for the RWS programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

RWS_$POOL_T

A 2-byte integer. Types of pools from which to allocate read/write or heap storage. One of the following predefined values:

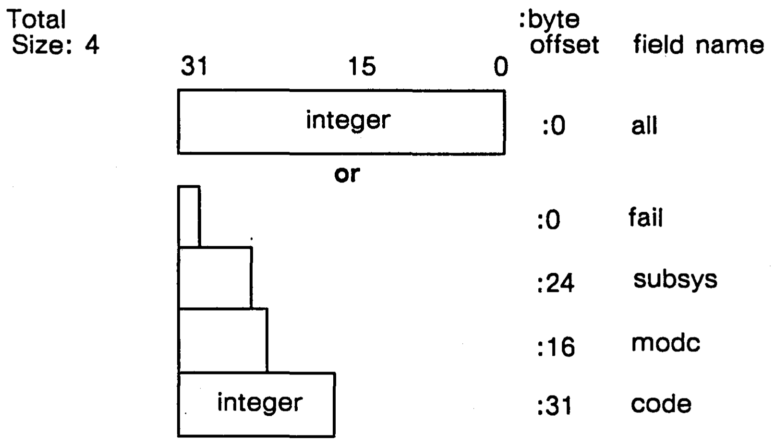
RWS_$STD_POOL
Standard pool makes storage accessible to calling process only.

RWS_$STREAM_TM_POOL
Stream pool makes storage accessible to calling program and to a program invoked with the UNIX EXEC system call.

RWS_$GLOBAL_POOL
Global pool makes storage accessible to all processes.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

	modc The module that encountered the error (bits 16 - 23). code A signed number that identifies the type of error that occurred (bits 0 - 15).
UNIV_PTR	A 4-byte integer. A pointer to allocated storage.

RWS_$ALLOC

Allocates read/write storage for C, FORTRAN or Pascal programs.

FORMAT

```
RWS_$ALLOC (storage_sz, storage_ptr)
```

INPUT PARAMETERS

storage_sz
The number of bytes of storage needed. This is a 4-byte integer.

OUTPUT PARAMETERS

storage_ptr
The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC could not allocate the desired storage.

USAGE

RWS_$ALLOC allocates the specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

This routine is useful for allocating different quantities of dynamic storage, depending on a run-time factor.

FORTRAN programmers: due to FORTRAN calling conventions, this is currently the only RWS call you can use to allocate read/write storage.

Pascal and C programmers can use other RWS calls to allocate read/write or heap (releaseable read/write) storage. See the calls, RWS_$ALLOC_RW_POOL and RWS_$ALLOC_HEAP_POOL for details.

C programmers might want to use the C library routine MALLOC to allocate storage.

RWS_$ALLOC_HEAP

Allocates heap (releaseable read/write) storage for Pascal and C programs.

FORMAT

```
storage_ptr = RWS_$ALLOC_HEAP (storage_sz)
```

RETURN VALUE

storage_ptr
The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC_HEAP could not allocate the desired storage.

INPUT PARAMETERS

storage_sz
The number of bytes of storage needed. This is a 4-byte integer.

USAGE

Note that RWS_$ALLOC_HEAP_POOL replaces this obsolete call, which we include for maintenance purposes only. For current and future development, use RWS_$ALLOC_HEAP_POOL.

RWS_$ALLOC_HEAP allocates the specified number of bytes of releaseable read/write storage to the calling process and returns the address of the storage area. It allocates storage from the standard RWS pool, which makes the storage accessible to the calling program only. Use RWS_$RELEASE_HEAP to release storage allocated with this call.

FORTRAN programmers: due to FORTRAN calling conventions, RWS_$ALLOC is currently the only RWS call you can use to allocate read/write storage.

C programmers might want to use the C library routine MALLOC to allocate storage.

RWS_$ALLOC_HEAP_POOL

Allocates heap (releasable read/write) storage from a specified pool.

FORMAT

```
storage_ptr  = RWS_$ALLOC_HEAP_POOL(alloc_pool, storage_sz )
```

RETURN VALUE

storage_ptr
The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC_HEAP_POOL could not allocate the desired storage.

INPUT PARAMETERS

alloc_pool
Pool from which the storage will be allocated, in RWS_$POOL_T format. This is a 2-byte integer. Specify one of the following predefined values:

RWS_$GLOBAL_POOL	Global pool makes storage accessible to all processes. Note that pointers are valid in *all* processes because they reserve the *identical* portion of address space.
RWS_$STD_POOL	Standard pool makes storage accessible to the calling program only. Most programs use this type.
RWS_$STREAM_TM_POOL	Stream pool makes storage accessible to the calling program and to a program invoked with a UNIX EXEC system call. Use this type when your program needs to pass information across a UNIX EXEC system call.

storage_sz
Number of bytes of storage needed. This is a 4-byte integer.

USAGE

RWS_$ALLOC_HEAP_POOL allocates a specified number of bytes of heap storage to the calling process and returns the address of the storage area.

When you no longer need the storage, call RWS_$RELEASE_HEAP_POOL to return the storage to the pool from which it was allocated.

Whether you allocate heap (releaseable read/write) storage with this call or read/write storage with RWS_$ALLOC_RW_POOL depends on how long you want to keep the storage. Once you allocate read/write storage, the storage exists until the program terminates. However, you can explicitly release heap storage once you have finished using it. The heap requires more system overhead initially to keep track of allocated storage. Read/write storage does not require any system overhead.

(Currently, the overhead for RWS_$ALLOC_HEAP_POOL is between 4 to 16 bytes -- The exact amount of call overhead is subject to change.)

Typically, you allocate heap storage if your program requires a substantial amount of storage for a limited time, or if you want to keep your working set as small as possible. You allocate read/write storage if you do not need to release storage before terminating a program, or if the amount of overhead for a heap is unacceptable.

When allocating heap or read/write storage, you control how your program accesses storage by specifying the type of storage pool to use:

- The standard pool (RWS_$STD_POOL) permits access to the calling process only.
- The global pool (RWS_$GLOBAL_POOL) permits access to all processes.
- The stream pool (RWS_$STREAM_TM_POOL) permits access to the calling program and a program invoked with a UNIX EXEC system call.

The global pool allows you to share information among processes. For example, you might want to create a global queue to pass messages between processes. Note that pointers are valid in *all* processes because all processes reserve an *identical* portion of address space.

The stream pool allows you to make storage accessible between a calling process and an overlay process. For example, the IOS manager uses a stream pool to pass an open stream to a program invoked with an EXEC call. It stores information about that stream in a stream pool.

The following table summarizes the aspects of each type of storage allocation.

Summary of Types of Storage Allocation

	Read/Write Storage	**Heap Storage**
Standard Pool	Storage kept until program exits or until it invokes a program with a UNIX EXEC system call.	Storage kept until you release it with RWS_$RELEASE_HEAP, the program exits, or the program invokes a program with a UNIX EXEC call.
	No system overhead.	About 16 bytes of system overhead.
	Storage available to local process only.	
Global Pool	Storage kept until reboot.	Storage kept until you release it with RWS_$RELEASE_HEAP or reboot.
	About 4 bytes of system overhead.	About 4 bytes of system overhead.
	Storage available to all processes.	

Summary of Types of Storage Allocation, Cont.

	Read/Write Storage	Heap Storage
Stream Pool	Storage kept until program exits.	Storage kept until you release it with RWS_$RELEASE_HEAP.
	No system overhead.	About 16 bytes of system overhead.
	Storage available to the local process or to a program invoked with a UNIX EXEC system call.	

NOTE: Do not depend on the exact amount of system overhead used in RWS system calls. The amount of overhead is subject to change.

Note that this call replaces the obsolete RWS_$ALLOC_HEAP call, which we include for maintenance purposes only. For current and future development, use RWS_$ALLOC_HEAP_POOL.

FORTRAN programmers: due to FORTRAN calling conventions, RWS_$ALLOC is currently the only RWS call you can use to allocate read/write storage. C programmers might want to use the C library routine MALLOC to allocate storage.

RWS_$ALLOC_RW

Allocates read/write storage for Pascal and C programs.

FORMAT

```
storage_ptr = RWS_$ALLOC_RW (storage_sz)
```

RETURN VALUE

storage_ptr
The address of the new storage space, in UNIV_PTR format This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC_RW could not allocate the desired storage.

INPUT PARAMETERS

storage_sz
The number of bytes of storage needed. This is a 4-byte integer.

USAGE

Note that RWS_$ALLOC_RW_POOL replaces this obsolete call, which we include for maintenance purposes only. For current and future development, use RWS_$ALLOC_RW_POOL.

RWS_$ALLOC_RW allocates the specified number of bytes of read/write storage to the calling process and returns the address of the storage area. It allocates storage from the standard RWS pool, which makes the storage accessible to the calling program only. This call does not require any system overhead.

FORTRAN programmers: due to FORTRAN calling conventions, RWS_$ALLOC is currently the only RWS call you can use to allocate read/write storage.

C programmers might want to use the C library routine MALLOC to allocate storage.

RWS_$ALLOC_RW_POOL

Allocates read/write storage from a specified pool.

FORMAT

```
storage_ptr  = RWS_$ALLOC_RW_POOL(alloc_pool, storage_sz )
```

RETURN VALUE

storage_ptr
The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC_RW_POOL could not allocate the desired storage.

INPUT PARAMETERS

alloc_pool
Pool from which storage will be allocated, in RWS_$POOL_T format. This is a 2-byte integer. Specify one of the following following predefined values:

RWS_$GLOBAL_POOL	Global pool makes storage accessible to all processes. Note that pointers are valid in *all* processes because they reserve the *identical* portion of address space.
RWS_$STD_POOL	Standard pool makes storage accessible to the calling program only. Most programs use this type.
RWS_$STREAM_TM_POOL	Stream pool makes storage accessible to the calling program and to a program invoked with a UNIX EXEC system call. Use this type when your program needs to pass information across a UNIX EXEC system call.

storage_sz
Number of bytes of storage needed. This is a 4-byte integer.

USAGE

RWS_$ALLOC_RW_POOL allocates a specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

Whether you allocate read/write storage with this call or heap (releaseable read/write) storage with RWS_$ALLOC_HEAP_POOL depends on how long you want to keep the storage. Once you allocate read/write storage, the storage exists until the program terminates. However, you can explicitly release heap storage once you have finished using it. The heap requires more system overhead initially, to keep track of allocated storage. Read/write storage does not require any system overhead.

Typically, you allocate read/write storage if you do not need to release storage before terminating a program, or if the amount of overhead for a heap is unacceptable. You allocate heap storage if your program requires a substantial amount of storage for a limited time, or if you want to keep your working set as small as possible.

When allocating read/write or heap storage, you control how your program accesses storage by specifying the type of storage pool to use:

- The standard pool (RWS_$STD_POOL) permits access to the calling process only.
- The global pool (RWS_$GLOBAL_POOL) permits access to all processes.
- The stream pool (RWS_$STREAM_TM_POOL) permits access to the calling program and a program invoked with a UNIX EXEC system call.

The global pool allows you to share information among processes. For example, you might want to create a global queue to pass messages between processes. Note that pointers are valid in *all* processes because all processes reserve an *identical* portion of address space.

The stream pool allows you to make storage accessible between a calling process and an overlay process. For example, the IOS manager uses a stream pool to pass an open stream to a program invoked with an EXEC call. It stores information about that stream in a stream pool.

The following table summarizes the aspects of each type of storage allocation.

Summary of Types of Storage Allocation

	Read/Write Storage	**Heap Storage**
Standard Pool	Storage kept until program exits or until it invokes a program with a UNIX EXEC system call.	Storage kept until you release it with RWS_$RELEASE_HEAP, the program exits, or the program invokes a program with a UNIX EXEC call.
	No system overhead.	About 16 bytes of system overhead.
	Storage available to local process only.	
Global Pool	Storage kept until reboot.	Storage kept until you release it with RWS_$RELEASE_HEAP or reboot.
	About 4 bytes of system overhead.	About 4 bytes of system overhead.
	Storage available to all processes.	

Summary of Types of Storage Allocation, Cont.

	Read/Write Storage	Heap Storage
Stream Pool	Storage kept until program exits.	Storage kept until you release it with RWS_$RELEASE_HEAP.
	No system overhead.	About 16 bytes of system overhead.
	Storage available to the local process or to a program invoked with a UNIX EXEC system call.	Storage available to the local process or to a program invoked with a UNIX EXEC system call.

NOTE: Do not depend on the exact amount of system overhead used in RWS system calls. The amount of overhead is subject to change.

Note that this call replaces the obsolete RWS_$ALLOC_RW call, which we include for maintenance purposes only. For current and future development, use RWS_$ALLOC_RW_POOL.

FORTRAN programmers: due to FORTRAN calling conventions, RWS_$ALLOC is currently the only RWS call you can use to allocate read/write storage. C programmers might want to use the C library routine MALLOC to allocate storage.

RWS_$RELEASE_HEAP

Releases storage allocated by the RWS_$ALLOC_HEAP call.

FORMAT

```
RWS_$RELEASE_HEAP (storage_ptr, status)
```

INPUT PARAMETERS

storage_ptr
The address heap storage space, in UNIV_PTR format. This is a 4-byte integer. This must be the pointer returned by a call to RWS_$ALLOC_HEAP.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the RWS Data Types section for more information.

USAGE

Note that RWS_$RELEASE_HEAP_POOL replaces this less efficient call, which we include for maintenance purposes only. For current and future development, use RWS_$RELEASE_HEAP_POOL.

Use this call to release the storage that you previously allocated with RWS_$ALLOC_HEAP.

RWS_$RELEASE_HEAP_POOL

Releases storage to the pool from which it was allocated.

FORMAT

```
RWS_$RELEASE_HEAP_POOL (storage_ptr, alloc_pool, status)
```

INPUT PARAMETERS

storage_ptr
Pointer to the address heap storage space, in UNIV_PTR format. This is a 4-byte integer. This must be the pointer returned by a call to RWS_$ALLOC_HEAP_POOL.

alloc_pool
Pool where storage will be returned to, in RWS_$POOL_T format. This is a 2-byte integer. Specify the same value you specified in the RWS_$ALLOC_HEAP_POOL call, which is one of the following predefined values:

RWS_$GLOBAL_POOL	Global pool makes storage accessible to all processes. Note that pointers are valid in *all* processes because they reserve the *identical* portion of address space.
RWS_$STD_POOL	Standard pool makes storage accessible to the calling program only. Most programs use this type.
RWS_$STREAM_TM_POOL	Stream pool makes storage accessible to the calling program and to a program invoked with a UNIX EXEC system call. Use this type when your program needs to pass information invoked with a UNIX EXEC system call.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the RWS Data Types section for more information.

USAGE

Use RWS_$RELEASE_HEAP_POOL to release storage to the pool from which it was allocated. You allocate storage to a specific pool with the RWS_$ALLOC_HEAP_POOL call.

RWS_$ALLOC_HEAP_POOL dynamically allocates storage from one of the three types of storage pools, and returns a pointer to the new storage. When you no longer need the storage, you release it by passing the "storage_ptr" and "alloc_pool" to RWS_$RELEASE_HEAP_POOL. RWS_$RELEASE_HEAP_POOL returns the storage to the pool from which it was allocated.

Note that this call replaces the less efficient RWS_$RELEASE_HEAP call, which we include for maintenance purposes only. For current and future development, use RWS_$RELEASE_HEAP_POOL.

ERRORS

RWS_$LEVEL_FAILURE
: User program wrote over the storage where the system stored the program level information.

RWS_$NOT_HEAP_ENTRY
: Argument to RWS_$RELEASE_HEAP did not refer to storage allocated with RWS_$ALLOC_HEAP.

RWS_$SCRIBBLED_OVER
: User program wrote over the storage where the system stored the heap's process information.

RWS_$WRONG_LEVEL
: Attempted to release storage that was allocated by a program at a superior (lower) program level. This error can occur when using RWS_$STD_POOL or RWS_$STREAM_TM_POOL.

SIO

This section describes the data types, the call syntax, and the error codes for the SIO programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

SIO_$50	Baud rate.
SIO_$75	Baud rate.
SIO_$110	Baud rate.
SIO_$134	Baud rate.
SIO_$150	Baud rate.
SIO_$300	Baud rate.
SIO_$600	Baud rate.
SIO_$1200	Baud rate.
SIO_$2000	Baud rate.
SIO_$2400	Baud rate.
SIO_$3600	Baud rate.
SIO_$4800	Baud rate.
SIO_$7200	Baud rate.
SIO_$9600	Baud rate.
SIO_$19200	Baud rate.
SIO_$EVEN_PARITY	Possible parity value.
SIO_$MAX_LINE	Maximum number of SIO lines.
SIO_$NO_PARITY	Possible parity value.
SIO_$ODD_PARITY	Possible parity value.
SIO_$STOP_1	Possible stop bit value.
SIO_$STOP_1_POINT_5	Possible stop bit value.
SIO_$STOP_2	Possible stop bit value.
SIO_$5BPC	Bits per character value.
SIO_$6BPC	Bits per character value.
SIO_$7BPC	Bits per character value.
SIO_$8BPC	Bits per character value.

DATA TYPES

SIO_$ERR_ENABLES_T	A 2-byte integer. Determines which errors are enabled. Any combination of the following pre-defined values:
SIO_$LINE_T	A 2-byte integer. SIO line number. Possible values are integers from 0 through SIO_$MAX_LINE (3).
SIO_$OPT_T	A 2-byte integer. An SIO option. One of the following pre-defined values:

SIO_$ERASE
Set erase character.

SIO_$KILL
Set kill character.

SIO_$EOFCHR
Set EOF character.

SIO_$RAW
Transparent input and output.

SIO_$NO_ECHO
Do not echo input.

SIO_$NO_NL
Do not special case newlines.

SIO_$SPEED
Set bit rate.

SIO_$HOST_SYNC
Use xoff/xon to synchronize with host.

SIO_$NLC_DELAY
Constant delay for newlines.

SIO_$QUIT_ENABLE
Enable quits from this line to calling process.

SIO_$INPUT_SYNC
Respond xoff/xon on receive side.

SIO_$LINE
Return line number (inquire only).

SIO_$RTS
Set/clear RTS bit.

SIO_$DTR
Set/clear DTR bit.

SIO_$DCD
Read DTR bit (inquire only).

SIO_$DCD_ENABLE
Enable fault on DCD loss.

SIO_$CTS
Read CTS bit (inquire only).

SIO_$CTS_ENABLE
Enable CTS gating of output.

SIO_$PARITY
Control parity setting/processing.

SIO_$BITS_PER_CHAR
Number of bits per character.

SIO_$STOP_BITS
Number of stop bits.

SIO_$ERR_ENABLE
Enable error reporting.

SIO_$SEND_BREAK
Establish break condition on line.

SIO_$QUITCHR
Set quit character.

SIO_$BP_ENABLE
Enable bit pad processing on line.

SIO_$INT_ENABLE
Enable interrupts in this process.

SIO_$INTCHR
Set interrupt character.

SIO_$SUSP_ENABLE
Enable process suspension character.

SIO_$SUSPCHR
Set process suspension character.

SIO_$RAW_NL
Display NL/CR on NL output in raw mode.

SIO_$UNUSED
Unused.

SIO_$HUP_CLOSE
Set hangup-on-close.

SIO_$RTS_ENABLE
Enable RTS flow control.

SIO_$SPEED_FORCE
Set bit rate, even if disturbs partner channel.

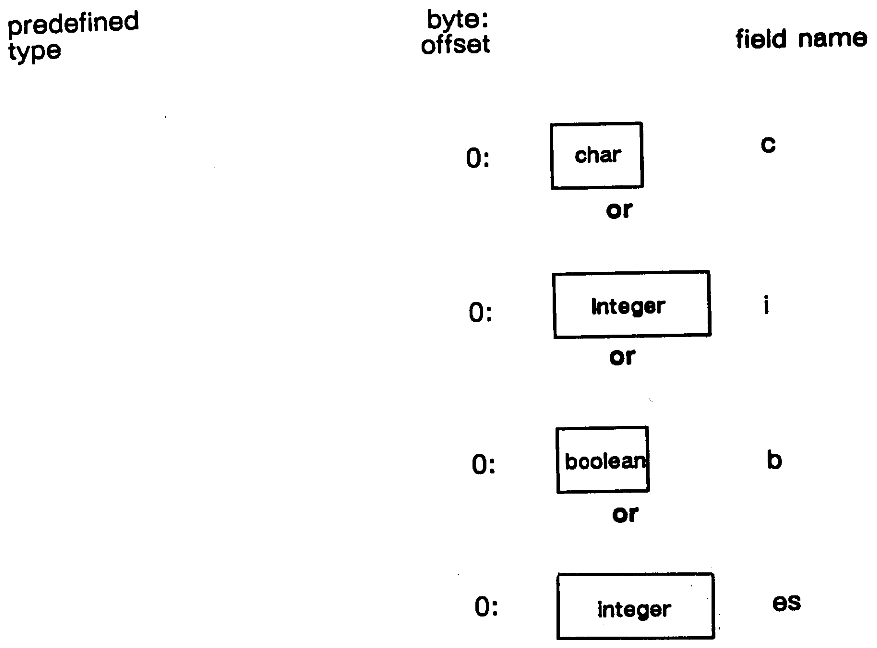
SIO_$FLUSH_IN
Flush input buffer.

SIO_$FLUSH_OUT
Flush output buffer.

SIO_$DRAIN_OUT
Wait for output buffer to drain.

SIO_$VALUE_T

Value corresponding to SIO options. The diagram below illustrates the SIO_$VALUE_T data type:



Field Description:

c
A character value.

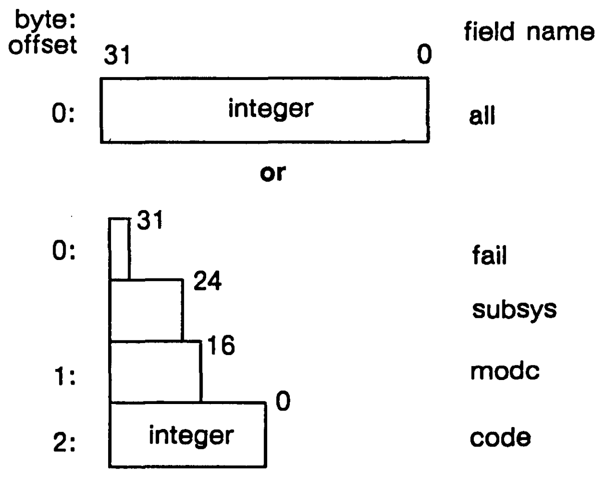
i
A 2-byte integer value.

b
A Boolean value.

es
A set of enabled errors. This is a 2-byte field.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

SIO_$CONTROL

Obtains current settings of serial line options and values.

FORMAT

```
SIO_$CONTROL (stream-id, option, value, status)
```

INPUT PARAMETERS

stream-id

Stream ID of a stream attached to a serial line. This is a 2-byte integer.

The stream specified by stream-id must be attached to a serial line. Any other attachment results in an error.

option

The attribute that is to be set, in SIO_$OPT_T format. This is a 2-byte integer. Specify only one of the following predefined values:

SIO_$ERASE	Sets the erase character, which erases the character immediately before the current cursor position. This option takes a character value. The default is <BACKSPACE>.
SIO_$KILL	Sets the kill character, which deletes characters from the cursor position to the end of the line. This option takes a character value. The default is CTRL/X.
SIO_$EOFCHR	Sets the end-of-file character. This option takes a character value. The default is CTRL/Z.
SIO_$RAW	Sets whether raw mode is on or off. This option takes a Boolean value. The default is FALSE (off). In raw mode, full 8-bit bytes are transmitted in both directions, without any interpretation. Each STREAM_$GET_REC call returns as many bytes as have been received since the last call. When raw mode is turned on or off, any input that your program has received, but has not yet read, is flushed from the input buffer.
SIO_$NO_ECHO	Sets whether NO_ECHO mode is on or off. In NO_ECHO mode, input characters are not automatically echoed as output. This mode may be used to support a half-duplex connection. NO_ECHO mode is off by default.
SIO_$NO_NL	Sets whether NO_NL mode is on or off. This option takes a Boolean value. The default is FALSE (off). Normally, newline characters (decimal 10) are transmitted as a carriage-return, line-feed. In NO_NL mode, the newline character is transmitted as is. This mode makes output transparent without going to raw input.

SIO_$HOST_SYNCH

Sets whether HOST_SYNCH mode is on or off. This option takes a Boolean value. The default is TRUE (on).

In HOST_SYNCH mode, the node sends XOFF (CTRL/S) when its input buffer begins to fill, and XON (CTRL/Q) when its input buffer begins to empty again. This allows for synchronization of high-speed data transfer from computer to computer.

SIO_$INPUT_SYNC

Sets whether the incoming synch mode is on or off. This option takes a Boolean value. The default is FALSE (off).

It is like HOST_SYNCH except it controls processing of incoming XON (CTRL/Q) or XOFF (CTRL/S). It works in raw or cooked mode.

SIO_$SPEED

Sets the baud rate of the line. This option takes a predefined 2-byte integer value. The default is SIO_$9600.

Possible values are:

```
SIO_$50,   SIO_$75,   SIO_$110,  SIO_$134,  SIO_$150,
SIO_$300,  SIO_$600,  SIO_$1200, SIO_$2000, SIO_$2400,
SIO_$3600, SIO_$4800, SIO_$7200, SIO_$9600, SIO_$19200.
```

If you attempt to set a partnered line to an incompatible baud rate, you receive the error status, SIO_$INCOMPATIBLE_SPEED. You may override this error using the SIO_$SPEED_FORCE option. See the USAGE section for details about partnered lines and incompatible speeds.

SIO_$SPEED_FORCE

Sets the baud rate of the line even if the partner line's speed is incompatible. This option takes a predefined 2-byte integer value.

Possible values are the same as for SIO_$SPEED:

```
SIO_$50,   SIO_$75,   SIO_$110,  SIO_$134,  SIO_$150,
SIO_$300,  SIO_$600,  SIO_$1200, SIO_$2000, SIO_$2400,
SIO_$3600, SIO_$4800, SIO_$7200, SIO_$9600, SIO_$19200.
```

When you use SIO_$SPEED FORCE to set the speed of a line, and the new speed is incompatible with the partner, the speed of the partner is changed to 9600 baud, See the USAGE section for details about partnered lines and incompatible speeds.

SIO_$NLC_DELAY

Sets the value of a time delay to be used following transmission of a line feed character, to allow for carriage motion, scrolling time, and so on. This option takes a 2-byte integer value, specifying the number of milliseconds of delay. The default is zero.

SIO_$QUIT_ENABLE

Sets whether THE quit_enable mode is on or off. This option takes a Boolean value. The default is FALSE (off).

In quit_enable mode, the node responds to CTRL/] and to the <BREAK> key, if any. The response is a quit fault in the process using SIO_$QUIT_ENABLE. If SIO_$QUIT_ENABLE is FALSE, then neither the <BREAK> key nor the CTLR/] sequence has any effect. In raw input mode, only the <BREAK> and CTRL/] sequence causes a quit fault.

SIO_$RTS Sets the state of the RTS (Request to Send) line. This option takes a Boolean value. The default is TRUE (on).

The RTS line is an outgoing line.

SIO_$RTS_ENABLE

Enables/disables the RTS (Return to Send) Line. This is a Boolean value. The default is FALSE (off).

If TRUE, the operating system handles flow control. For this to work properly the CTS line must also be enabled.

SIO_$DTR Sets the state of the outgoing DTR (Data Terminal Ready) line. This option takes a Boolean value. The default is TRUE (on).

On most modems the DTR line controls whether the modem will answer incoming calls (i.e., when DTR is TRUE.) When it is reset it causes the modem to hang up the phone line.

SIO_$DCD_ENABLE

Sets whether the DCD_ENABLE mode is on or off. This option takes a Boolean value. The default is FALSE (off).

If the connection is broken (i.e., the remote modem hangs up) the DCD line becomes FALSE. If SIO_$DCD_ENABLE is TRUE then a fault with status FAULT_$STOP will occur at the time of the transition of DCD from TRUE to FALSE.

SIO_$CTS_ENABLE

Sets whether the CTS_ENABLE mode is on or off. This option takes a Boolean value. The default is FALSE (off).

Some devices use one of the RS-232 control lines for flow control instead of XON/XOFF. If such a line is wired to the CTS line on the connector and if SIO_$CTS_ENABLE is TRUE, then transmission will be inhibited whenever CTS is FALSE.

SIO_$PARITY Sets the state of parity detection or parity generation. This option takes a predefined 2-byte integer value. The default is SIO_$NO_PARITY. Possible choices are: SIO_$ODD_PARITY, SIO_$EVEN_PARITY, and SIO_$NO_PARITY.

If parity is enabled (whether odd or even) then one bit is added to each character. The parity bit is checked by the hardware on received characters and errors are reported, subject to the SIO_$ERR_ENABLE option. If the number of bits per character is fewer than 8, then the parity bit is delivered with the data in raw mode and is stripped in cooked mode.

SIO_$BITS_PER_CHAR
Sets the number of bits per character. This option takes a predefined 2-byte integer value. The default is SIO_$8BPC, which is 8 bits per character. Possible choices are: SIO_$xBPC, where x may be 5, 6, 7, or 8.

SIO_$STOP_BITS
Sets the number of stop bits. This option takes a predefined 2-byte integer value. The default is SIO_$STOP_1. Possible values are: SIO_$STOP_x where x may be 1, 1_POINT_5, or 2.

SIO_$ERR_ENABLE
Sets which kinds of errors can be reported in calls to STREAM_$GET_REC on this stream. This option takes a set of values, in SIO_$ERR_ENABLES_T format. Specify any combination of the following predefined 2-byte integer values:

SIO_$CHECK_PARITY
Report received parity errors.

SIO_$CHECK_FRAMING
Report received framing errors.

SIO_$CHECK_DCD_CHANGE
Report an "error" when DCD line changes state.

SIO_$CHECK_CTS_CHANGE
Report an "error" when CTS line changes state.

SIO_$CHECK_FRAMING is set by default.

SIO_$SEND_BREAK
Causes a break condition on the line. This option takes a 2-byte integer value, specifying the duration of the break, in milliseconds. The default is ...? A reasonable value is 200.

SIO_$QUITCHR
Sets the quit character This option takes a character value. The default is CTRL/.

SIO_$BP_ENABLE
Enables/disables processing of bit pad input from a graphics tablet. This option takes a Boolean value. The default is FALSE (disabled).

When enabled, data received on the SIO line is not delivered through STREAM_$GET_REC. Instead, the SIO driver interrupt routine accumulates data and passes it a point at a time to the display driver. During this processing, subsequent points within plus or minus two in both x and y dimensions are ignored.

SIO_$INT_ENABLE
: Enables/disables interrupts for the current process. This option takes a Boolean value. The default is FALSE (disabled).

SIO_$INTCHR Sets the process interrupt character. (This option is used primarily by DOMAIN/IX.) This option takes a character value. The default is FALSE (CRTL/C).

SIO_$SUSP_ENABLE
: Enables/disables suspend faults for the current process. This option takes a Boolean value. The default is FALSE (disabled).

SIO_$SUSPCHR Sets the process suspend character. (This option is used primarily by DOMAIN/IX.) This option takes a character value. The default is CRTL/P.

SIO_$RAW_NL
: Sets whether NO_NL mode is on or off in raw mode.(i.e., when SIO_$RAW is TRUE). This option takes a Boolean value. The default is FALSE (off).

Normally, newline characters (decimal 10) are transmitted as a carriage-return, line-feed. In NO_NL mode, the newline character is transmitted as is. This mode makes output transparent without going to raw input.

SIO_$HUP_CLOSE
: Causes the modem to be hung up on the last close (STREAM_$CLOSE) of the SIO line. The hangup is performed by dropping DTR for 3/4 second.

SIO_$FLUSH_IN
: Causes the input buffer of an SIO line to be flushed. This option takes a Boolean value. The default is FALSE (off).

SIO_$FLUSH_OUT
: Causes the output buffer of an SIO line to be flushed. This option takes a Boolean value. The default is FALSE (off).

SIO_$DRAIN_OUT
: Causes the process to wait until all the characters in the output buffer have been transmitted before returning. This option takes a Boolean value. The default is FALSE (off).

value

Each of the SIO_$CONTROL options accepts a corresponding value. For the character options, the value is simply the character. For the mode-setting options, the value is a Boolean (LOGICAL) data item. For most of the remaining options, the value is a 2-byte integer. In one case, you may specify a set of values. The type of value required for each option is described along with the option, above.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the SIO Data Types section for more information.

Possible values are:

STATUS_$OK The operation completed successfully.

STREAM_$NOT_OPEN
No stream is open on the specified stream ID.

SIO_$STREAM_NOT_SIO
The specified stream ID is not connected to a serial line.

SIO_$BAD_OPTION
The call specified an invalid option name.

SIO_$INCOMPATIBLE_SPEED
The specified speed is incompatible with the speed of the line's partner.

USAGE

To poll the serial line for unread input, use STREAM_$GET_CONDITIONAL.

The hardware configuration for some machine types is such that certain SIO lines are "partnered" with each other. Below is a list of machine types and the SIO lines that are partnered on them.

Machine Type	*Partnered Lines*
DN400 DN420 DN600	No Partners
DN300	1,2
DSP80	1,2
DN460 DN660	0,1 2,3
DN550	1,2

A characteristic of partnered lines is that some baud rates are incompatible. That is, certain combinations of baud rates can not be held by partnered lines. Below are two lists of baud rates.

Incompatible Rates A	*Incompatible Rates B*
SIO_$50 SIO_$7200	SIO_$75 SIO_$150 SIO_$2000 SIO_$19200

If one partner is set to a baud rate in the A list, attempting to set the other partner to a baud rate in the B list (using the SIO _ $SPEED option) will result in the error, SIO _ $INCOMPATIBLE _ SPEED. The same is true for the reverse (having a partnered line set to a rate in the B list and attempting to set its partner to a rate in the A list). Speeds other than those in the two lists have no compatibility issues.

You may choose to set an incompatible baud rate by force, using the SIO _ $SPEED _ FORCE option. This will change the specified line to the specified speed; however, it will change the speed of the partnered line to SIO _ $9600 (which is always a compatible speed).

SIO_$INQUIRE

Obtains current settings of serial line options and values.

FORMAT

```
SIO_$INQUIRE (stream-id, option, value, status)
```

INPUT PARAMETERS

stream-id

Stream-id of a stream attached to a serial line. This is a 2-byte integer.

option

The attribute that is to be reported, in SIO_$OPT_T format. This is a 2-byte integer. One of the following predefined values:

SIO_$ERASE	Returns the erase character, which erases the character immediately before the current cursor position. This option takes a character value. The default is <BACKSPACE>.
SIO_$KILL	Returns the kill character, which deletes characters from the cursor position to the end of the line. This option takes a character value. The default is CTRL/X.
SIO_$EOFCHR	Returns the end-of-file character. This option takes a character value. The default is CTRL/Z.
SIO_$RAW	Returns whether raw mode is on or off. This option takes a Boolean value. The default is FALSE (off). In raw mode, full 8-bit bytes are transmitted in both directions, without any interpretation. Each STREAM_$GET_REC call returns as many bytes as have been received since the last call. When raw mode is turned on or off, any input that your program has received, but has not yet read, is flushed from the input buffer.
SIO_$NO_ECHO	Returns whether the NO_ECHO mode is on or off. In NO_ECHO mode, input characters are not automatically echoed as output. This mode may be used to support a half-duplex connection. NO_ECHO mode is off by default.
SIO_$NO_NL	Returns whether the NO_NL mode is on or off. This option takes a Boolean value. The default is FALSE (off). Normally, newline characters (decimal 10) are transmitted as a carriage-return, line-feed. In NO_NL mode, the newline character is transmitted as is. This mode makes output transparent without going to raw input.
SIO_$HOST_SYNCH	Returns whether the HOST_SYNCH mode is on or off. This option takes a Boolean value. The default is TRUE (on).

In HOST_SYNCH mode, the node sends XOFF (CTRL/S) when its input buffer begins to fill, and XON (CTRL/Q) when its input buffer begins to empty again. This allows for synchronization of high-speed data transfer from computer to computer.

SIO_$INPUT_SYNC

Returns whether incoming synch mode is on or off. This option takes a Boolean value. The default is FALSE (off).

It is like HOST_SYNCH except it controls processing of {_incoming_} XON (CTRL/Q) or XOFF (CTRL/S). It works in raw or cooked mode.

SIO_$LINE

Returns the serial line number corresponding to the stream ID. This option returns an integer value from 0 to 3.

SIO_$SPEED

Returns the baud rate of the line. This option takes a predefined 2-byte integer value. The default is SIO_$9600.

Possible values are:

```
SIO_$50,   SIO_$75,   SIO_$110,  SIO_$134,  SIO_$150,
SIO_$300,  SIO_$600,  SIO_$1200, SIO_$2000, SIO_$2400,
SIO_$3600, SIO_$4800, SIO_$7200, SIO_$9600, SIO_$19200.
```

If you attempt to set a partnered line to an incompatible baud rate, you receive the error message, SIO_$INCOMPATIBLE_SPEED. You may override this error using the SIO_$SPEED_FORCE option. See the USAGE section for details about partnered lines and incompatible speeds.

SIO_$NLC_DELAY

Returns the value of a time delay to be used following transmission of a line feed character, to allow for carriage motion, scrolling time, and so on. This option takes a 2-byte integer value, specifying the number of milliseconds of delay. The default is zero.

SIO_$QUIT_ENABLE

Returns whether the QUIT_ENABLE mode is on or off. This option takes a Boolean value. The default is FALSE (off).

In QUIT_ENABLE mode, the node responds to CTRL/] and to the <BREAK> key, if any. The response is a quit fault in the process using SIO_$QUIT_ENABLE. If SIO_$QUIT_ENABLE is FALSE then neither the <BREAK> key nor the CTLR/] sequence has any effect. In raw input mode only the <BREAK>, and CTRL/] sequence causes a quit fault.

SIO_$RTS

Returns the state of the RTS (Request to Send) line. This option takes a Boolean value. The default is TRUE (on).

The RTS line is an outgoing line.

SIO_$RTS_ENABLE
Enables/disables the RTS (Return to Send) Line. This is a Boolean value. The default is FALSE (off).

If TRUE, the operating system handles flow control. For this to work properly the CTS line must also be enabled.

SIO_$DTR
Returns the state of the outgoing DTR (Data Terminal Ready) line. This option takes a Boolean value. The default is TRUE (on).

On most modems the DTR line controls whether the modem answers incoming calls (i.e., when DTR is TRUE.) When it is reset it causes the modem to hang up the phone line.

SIO_$DCD
Reports the state of the DCD (Data Carrier Detect) line. It is an incoming line, which usually means there is an active modem at the other end of the phone line.

SIO_$DCD_ENABLE
Returns whether DCD_ENABLE mode is on or off. This option takes a Boolean value. The default is FALSE (off).

If the connection is broken (i.e., the remote modem hangs up) the DCD line becomes FALSE. If SIO_$DCD_ENABLE is TRUE then a fault with status FAULT_$STOP occurs at the time of the transition of DCD from TRUE to FALSE.

SIO_$CTS
Returns the state of the CTS (Clear to Send) line. This is an incoming line.

SIO_$CTS_ENABLE
Returns whether CTS_ENABLE mode is on or off. This option takes a Boolean value. The default is FALSE (off).

Some devices use one of the RS-232 control lines for flow control instead of XON/XOFF. If such a line is wired to the CTS line on the connector and if SIO_$CTS_ENABLE is TRUE, then transmission will be inhibited whenever CTS is FALSE.

SIO_$PARITY
Returns the state of parity detection or parity generation. This option takes a predefined 2-byte integer value. The default is SIO_$NO_PARITY. Possible choices are: SIO_$ODD_PARITY, and SIO_$EVEN_PARITY, and SIO_$NO_PARITY.

If parity is enabled (whether odd or even) then one bit is added to each character. The parity bit is checked by the hardware on received characters and errors are reported, subject to the SIO_$ERR_ENABLE option. If the number of bits per character is fewer than 8, then the parity bit is delivered with the data in raw mode and is stripped in cooked mode.

SIO_$BITS_PER_CHAR
Returns the number of bits per character. This option takes a predefined 2-byte integer value. The default is SIO_$8BPC, which is 8 bits per character. Possible choices are: SIO_$xBPC, where x may be 5, 6, 7, or 8.

SIO_$STOP_BITS
Returns the number of stop bits. This option takes a predefined 2-byte integer value. The default is SIO_$STOP_1. Possible values are: SIO_$STOP_x where x may be 1, 1_POINT_5, or 2.

SIO_$ERR_ENABLE
Returns which kinds of errors can be reported in calls to STREAM_$GET_REC on this stream. This option takes a set of values, in SIO_$ERR_ENABLES_T format. Specify any combination of the following predefined 2-byte integer values:

SIO_$CHECK_PARITY
Report received parity errors.

SIO_$CHECK_FRAMING
Report received framing errors.

SIO_$CHECK_DCD_CHANGE
Report an "error" when DCD line changes state.

SIO_$CHECK_CTS_CHANGE
Report an "error" when CTS line changes state.

SIO_$CHECK_FRAMING is set by default.

SIO_$QUITCHR
Returns the quit character. This option takes a character value. The default is CTRL/.

SIO_$BP_ENABLE
Enables/disables processing of bit pad input from a graphics tablet. This option takes a Boolean value. The default is FALSE (disabled).

When enabled, data received on the SIO line is not delivered through STREAM_$GET_REC. Instead, the SIO driver interrupt routine accumulates data and passes it a point at a time to the display driver. During this processing, subsequent points within plus or minus two in both x and y dimensions are ignored.

SIO_$INT_ENABLE
Enables/disables interrupts for the current process. This option takes a Boolean value. The default is FALSE (disabled).

SIO_$INTCHR Returns the process interrupt character. (This option is used primarily by DOMAIN/IX.) This option takes a character value. The default is FALSE (CRTL/C).

SIO_$SUSP_ENABLE
Enables/disables suspend faults for the current process. This option takes a Boolean value. The default is FALSE (disabled).

SIO_$SUSPCHR Returns the process suspend character. (This option is used primarily by DOMAIN/IX.) This option takes a character value. The default is CRTL/P.

SIO_$RAW_NL
Returns whether the NO_NL mode is on or off in raw mode.(i.e., when SIO_$RAW is TRUE). This option takes a Boolean value. The default is FALSE (off).

Normally, newline characters (decimal 10) are transmitted as a carriage-return, line-feed. In NO_NL mode, the newline character is transmitted as is. This mode makes output transparent without going to raw input.

SIO_$HUP_CLOSE
Causes the modem to be hung up on the last close (STREAM_$CLOSE) of the SIO line. The hangup is performed by dropping DTR for 3/4 second. This option takes a Boolean value. The default is FALSE (off).

OUTPUT PARAMETERS

value
Each of the SIO_$INQUIRE options returns a corresponding value. For the character options, the value is simply the character. For the mode-setting options, the value is a Boolean (LOGICAL) data item. For most of the remaining options, the value is a 2-byte integer. In one case, a set of values may be returned. The type of value returned for each option is described along with the option, above.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the SIO Data Types section for more information.

Possible values are:

STATUS_$OK The operation completed successfully.

STREAM_$NOT_OPEN
No stream is open on the specified stream ID.

SIO_$STREAM_NOT_SIO
The specified stream ID is not connected to a serial line.

SIO_$BAD_OPTION
The call specified an invalid option name.

USAGE

The stream specified by stream ID must be attached to a serial line. Any other attachment results in an error.

When raw mode is turned on or off, any input that your program has received, but has not yet read, is flushed from the input buffer.

To poll the serial line for unread input, use STREAM_$GET_CONDITIONAL.

ERRORS

STATUS_$OK
: Successful completion.

SIO_$BAD_OPTION
: Bad option parameter.

SIO_$ILLEGAL_STRID
: Illegal stream ID.

SIO_$INCOMPATIBLE_SPEED
: Speed incompatible with partner SIO line.

SIO_$STREAM_NOT_OPEN
: Stream not open.

SIO_$STREAM_NOT_SIO
: Object on this stream is not an SIO line.

SMD

This section describes the call syntax and the error codes for the SMD programming calls. Refer to the Introduction at the beginning of this manual for a description of call syntax format. Refer to the SMD insert files for data-type descriptions.

SMD_$BLT_U

Starts a bit transfer from one area of display memory to another.

FORMAT

```
SMD_$BLT_U (registers, status)
```

INPUT PARAMETERS

registers
Values for the BLT register, in SMD_$BLT_REGS_T format. This is a thirteen-element array of 2-byte integers.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$BLT_U starts the block transfer within display memory.

The BLT register contains values for CS; CD; source and destination Xs, Xe, Ys, and Ye; and the display mode.

In FORTRAN programs, specify the display mode as the sum of selected variables from Table SMD-1.

By default, the display driver waits for the BLT to complete before returning to the calling program. (This is a "busy" wait, meaning the CPU is active while waiting.) If the display mode includes IDONE, however, control returns to the caller immediately, and generates the SMD event SMD_$EVENT_SCROLL_BLT_COMPLETE (see SMD_$EVENT_WAIT_U) when it finishes.

If the display mode includes IDONE, control returns to the calling program immediately, as noted above. However, if display memory is mapped into the process's address space, the program must not reference display memory or call any of the following vector-drawing routines, until the BLT completes:

SMD_$DRAW_ABS_U
SMD_$DRAW_REL_U
SMD_$MOVE_ABS_U
SMD_$MOVE_REL_U

Table SMD-1. Display Mode Values

Mnemonic	*Meaning*
SMD_$CLRMODE	Fill destination with a constant.
SMD_$DECR	Source overlaps destination and x is being decremented, that is, destination is to right of source.
SMD_$IDONE	Start BLT and immediately return control to calling program.
SMD_$NONINTERLACE	Disable hardware interlacing.
SMD_$BLT	Perform bit BLT operation. Required in all calls to SMD_$BLT_U.

SMD_$BORROW_DISPLAY_NC_U

Requests use of the display driver and display memory without clearing the screen (black and white only).

FORMAT

```
SMD_$BORROW_DISPLAY_NC_U (unit, status)
```

INPUT PARAMETERS

unit
Unit number of the display to be used. This is a 2-byte integer, and must be equal to 1.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$BORROW_DISPLAY_NC_U requests the use of the display memory and the display (SMD) driver routines. A program must execute this routine (or SMD_$BORROW_DISPLAY_U) before it can call any other display driver routines.

This procedure gains exclusive use of the display for the calling program. The display manager continues to operate for all other processes and pads, but the screen does not reflect its actions. Control of the screen returns to the display manager when SMD_$RETURN_DISPLAY_U is executed, or when you type CTRL/Q.

To gain access to display memory, call SMD_$MAP_DISPLAY_U.

Currently, the only valid unit number is 1.

SMD_$BORROW_DISPLAY_U

Requests use of the display driver and display memory, and clears the screen.

FORMAT

```
SMD_$BORROW_DISPLAY_U (unit, status)
```

INPUT PARAMETERS

unit
Unit number of the display to be used. This is a 2-byte integer, and must equal 1.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$BORROW_DISPLAY_U requests the use of the display memory and the display driver routines. A program must execute this routine (or SMD_$BORROW_DISPLAY_NC_U) before it can call any other display driver routines.

The display is cleared when this procedure is executed.

SMD_$BORROW_DISPLAY_NC_U performs an identical borrowing operation, but does not clear the screen.

This procedure gains exclusive use of the Display for the calling program. The Display Manager continues to operate for all other processes and pads, but the screen does not reflect its actions. Control of the screen returns to the display manager when SMD_$RETURN_DISPLAY_U is executed, or when you type CTRL/Q.

To gain access to display memory, call SMD_$MAP_DISPLAY_U.

Currently, the only valid unit number is 1.

SMD_$CLEAR_KBD_CURSOR_U

Clears the keyboard cursor from the display.

FORMAT

```
SMD_$CLEAR_KBD_CURSOR_U (status)
```

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$CLEAR_KBD_CURSOR_U disables the keyboard cursor and removes it from the display. To re-enable the cursor, call SMD_$MOVE_KBD_CURSOR_U.

SMD_$CLEAR_WINDOW_U

Clears an area on the screen.

FORMAT

```
SMD_$CLEAR_WINDOW_U (boundaries, status)
```

INPUT PARAMETERS

boundaries
The x and y coordinates of the destination area to be cleared, in SMD_$WINDOW_LIMITS_T format. This data type is 8 bytes long. In FORTRAN, use a four-element array of 2-byte integers.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$CLEAR_WINDOW_U clears the area of the screen within the boundaries.

This procedure returns control to its caller after the area is clear.

SMD_$COLOR

Sets the color of lines drawn on the display.

FORMAT

```
SMD_$COLOR (color)
```

INPUT PARAMETERS

color
Either SMD_$WHITE or SMD_$BLACK. This is a 2-byte integer.

USAGE

The color set with this call is used in all subsequent vector or box drawing calls executed by the program, until another SMD_$COLOR call is executed.

SMD_$WHITE makes subsequent vectors white or green. SMD_$BLACK makes subsequent vectors black.

This call does not change the color of the background.

SMD_$COND_EVENT_WAIT_U

Checks an SMD eventcount, but does not wait.

FORMAT

```
SMD_$COND_EVENT_WAIT_U (event-type, event-data, reserved, status)
```

OUTPUT PARAMETERS

event-type
The type of event that occurred. This is a 2-byte integer. Possible values are SMD_$INPUT, SMD_$SCROLL_BLT_COMPLETE, SMD_$TPAD_DATA, SMD_$TPAD_AND_INPUT, and SMD_$NO_EVENT.

event_data
The data associated with the event, in SMD_$EVENT_DATA_T format. This is a 2-byte integer.

reserved
A 2-byte integer; reserved for future use.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$COND_EVENT_WAIT_U causes no suspension of the calling program. Programs can use this procedure to check for keyboard or touchpad input.

The only difference between SMD_$COND_EVENT_WAIT_U and SMD_$EVENT_WAIT_U is that the first checks the eventcount but does not wait. The second waits for an event.

The SMD_$COND_EVENT_WAIT_U routine returns one type of event (SMD_$NO_EVENT) that SMD_$EVENT_WAIT_U does not. When this type is output, it means nothing happened.

SMD_$COND_INPUT_U

Returns a character if one has been typed.

FORMAT

```
input-flag = SMD_$COND_INPUT_U(char)
```

RETURN VALUE

input-flag
Boolean (LOGICAL) value. Contains TRUE if a character has been typed and FALSE otherwise.

OUTPUT PARAMETERS

char
The character typed at the keyboard. This is a character variable.

USAGE

If a character has been typed at the keyboard, the value of this function is TRUE. The function returns the character and removes it from the keyboard input buffer.

If no characters have been typed, the value of the function is FALSE, and the returned char parameter is undefined.

SMD_$DRAW_ABS_U

Draws a vector given an absolute position.

FORMAT

```
SMD_$DRAW_ABS_U (column, line)
```

INPUT PARAMETERS

column

The number of the column to which the vector will be drawn. This is a 2-byte integer in the range 0 to 799.

line

The number of the line to which the vector will be drawn. This is a 2-byte integer in the range 0 to 1023.

USAGE

SMD_$DRAW_ABS_U draws a vector from the current position to the point specified by (column, line).

Call SMD_$VECTOR_INIT_U once to initialize the vector drawing package before using this procedure.

No error checking is performed on the arguments, for optimal performance. Incorrect program operation occurs if a column or line value is outside the specified range.

The current position is updated to (column, line) upon completion of this procedure. Use SMD_$MOVE_ABS_U or SMD_$MOVE_REL_U to set the position without drawing a vector.

SMD_$DRAW_BOX_U

Draws a box on the screen.

FORMAT

```
SMD_$DRAW_BOX_U (boundaries, status)
```

INPUT PARAMETERS

boundaries
The x and y coordinates of the box on the screen, in SMD_$WINDOW_LIMITS_T format. This data type is 8 bytes long. In FORTRAN this is a four-element array of 2-byte integers.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format.

USAGE

SMD_$DRAW_BOX_U draws lines vertically and horizontally to connect the supplied endpoints.

The supplied values for Xs and Ys must be less than Xe and Ye, respectively.

SMD_$DRAW_REL_U

Draws a vector given a relative position.

FORMAT

```
SMD_$DRAW_REL_U (column, line)
```

INPUT PARAMETERS

column
The column number, relative to the current position, to which the vector will be drawn. This is a two-byte integer.

line
The line number, relative to the current position, to which the vector will be drawn. This is a two-byte integer.

USAGE

SMD_$DRAW_REL_U draws a vector from the current position to the point computed by adding the value of column to the current column and adding the value of line to the current line.

Call SMD_$VECTOR_INIT_U to initialize the vector drawing package before using this procedure.

When the value for column is added to the current position column number, the sum must be between 0 and 799 for portrait displays, or between 0 and 1023 for landscape displays. Similarly, when the value for line is added to the current position line number, the sum must be between 0 and 1023 for portrait displays, or 0 and 799 for landscape displays.

No error checking is performed on the arguments, for optimal performance. Incorrect program operation occurs if a computed column or line value is outside the specified range.

The current position is updated to the computed column and line values by this procedure. Use SMD_$MOVE_REL_U or SMD_$MOVE_ABS_U to set the position without drawing a vector.

SMD_$EVENT_WAIT_U

Suspends the calling process until you type characters at the keyboard or until the current scroll or BLT is complete.

FORMAT

```
SMD_$EVENT_WAIT_U (event-type, event-data, reserved, status)
```

OUTPUT PARAMETERS

event-type
The type of event that occurred. This is a 2-byte integer. Possible values are the following:

```
SMD_$INPUT
SMD_$SCROLL_BLT_COMPLETE
SMD_$TPAD_DATA
SMD_$TPAD_AND_INPUT
```

event-data
The data associated with the event, in SMD_$EVENT_DATA_T format. This is a 2-byte integer.

reserved
A 2-byte integer; reserved for future use.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$EVENT_WAIT_U suspends the calling process until a display driver event occurs. Programs can use this procedure to read keyboard input.

An SMD_$INPUT event occurs when you type a character.

An SMD_$SCROLL_BLT_COMPLETE event occurs when a block transfer (BLT) is complete, or when the last block transfer required as part of a scrolling operation has been started.

The display driver notifies the calling program of SMD_$INPUT, SMD_$SCROLL_BLT_COMPLETE, SMD_$TPAD_DATA, and SMD_$TPAD_AND_INPUT events.

SMD_$GET_EC

Gets the eventcount address of the eventcount that will be advanced upon keyboard input or when a BLT is done.

FORMAT

```
SMD_$GET_EC (smd-key, eventcount-pointer, status)
```

INPUT PARAMETERS

smd-key
This specifies which eventcount to obtain. It is in SMD_$EC_KEY_T format and may have a value of either SMD_$INPUT_EC (for the keyboard) or SMD_$SCROLL_BLT_EC (for a user-initiated BLT.) This is a 2-byte integer.

OUTPUT PARAMETERS

eventcount_pointer
The eventcount address to be obtained, in EC2_$PTR_T format. EC2_$PTR_T is a pointer to an EC2_$EVENTCOUNT_T. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

The SMD eventcounter is advanced whenever anything is entered via the keyboard and also whenever any user-initiated BLT is done.

SMD_$INQ_DISP_TYPE

Returns the type of the display physically attached to the given unit number.

FORMAT

```
display_type = SMD_$INQ_DISP_TYPE (unit)
```

RETURN VALUE

display_type

The display configuration in smd_$display_type_t format. This is a 2-byte integer. It has one of the following predefined values:

`SMD_$NONE`	`No display`
`SMD_$BW_15P`	`Black and white portrait`
`SMD_$BW_19L`	`Black and white landscape`
`SMD_$COLOR_DISPLAY`	`Color display (1024 x 1024)`
`SMD_$800_COLOR`	`Color display with fewer pixels (1024 x 800)`
`SMD_$COLOR2_DISPLAY`	`Color display (1280x1024x8)`
`SMD_$COLOR3_DISPLAY`	`Color display (1024x800x8)`
`SMD_$COLOR4_DISPLAY`	`Color display (1024x800x4)`

INPUT PARAMETERS

unit

This parameter has three possible meanings, as follows:

1. The display unit, if the graphics routines are to operate in a borrowed display. This is a 2-byte integer. Currently, the only valid display unit number for borrow-display mode is 1.

2. The stream identifier for the pad, if the graphics routines are to operate in frame or direct mode. Use STREAM_$ID_T format. This is a 2-byte integer.

3. Any value, such as zero, if the graphics routines do not use the display. This is a 2-byte integer.

USAGE

Use this call to return the type of the display physically attached to the given unit number.

SMD_$LOAD_FONT_FILE_U

Loads a font file.

FORMAT

```
SMD_$LOAD_FONT_FILE_U (pathname, name-length, font-id, status)
```

INPUT PARAMETERS

pathname
Pathname, in NAME_$PNAME_T format, of the file containing the font to be loaded.

name-length
The number of characters in the pathname. This is a 2-byte integer.

OUTPUT PARAMETERS

font-id
The internal identifier assigned to the font. Font-id is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$LOAD_FONT_FILE_U loads the font in the named file into display memory and assigns an identifier to the font. Your program passes the font-id to SMD_$WRITE_STRING to identify the font.

The images of all loaded fonts coexist in the invisible 28-K byte portion of display memory. This area is large enough for about eight small fonts.

If insufficient space is available in either display memory or internal tables to load the font, SMD_$LOAD_FONT_FILE_U returns an error. In this case, your program must unload one or more font files to create space for the new font.

To unload fonts loaded with this routine, use SMD_$UNLOAD_FONT_FILE_U.

Fonts loaded with this routine are no longer usable when the program exits or aborts. They are not, however, unloaded from display memory.

SMD_$LOAD_FONT_U

Loads a font into display memory and returns a font-id.

FORMAT

```
font-id = SMD_$LOAD_FONT_U (table-ptr, status)
```

RETURN VALUE

font-id
The internal identifier assigned to the font. This is a two-byte integer.

INPUT PARAMETERS

table-ptr
Address of the font table. This is a 4-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

This function returns an integer font-id. Use the font-id to identify the font in calls to SMD_$WRITE_STRING_U.

FORTRAN programs can use the IADDR function to get an address. Pascal programs can use the ADDR function.

The MS_$MAP call can be used to map a font file prior to using this call.

The display driver loads the font into any available space in the invisible 28K bytes of display memory. This area is large enough for about eight small fonts.

If insufficient space remains in invisible display memory or for internal tables, an error occurs. Your program must then unload one or more fonts to make room for the new one.

To unload the font, use SMD_$UNLOAD_FONT_U.

SMD_$MAP_DISPLAY_U

Maps display memory into the process' address space.

FORMAT

```
SMD_$MAP_DISPLAY_U (display-address, status)
```

OUTPUT PARAMETERS

display-address
The address of the first byte of the display memory. Display memory is mapped starting at this address, for the next 128-K bytes. This value is in SMD_$DISPLAY_MEMORY_PTR_T format. This is a 4-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$MAP_DISPLAY_U creates an association between the display memory and 128-K bytes of the calling process' address space. Following this call, your program can directly access the display memory by references to the mapped portion of the address space.

Be careful not to access display memory while a bit BLT is underway. Doing so causes the offending program to abort with a hardware bus error fault. To avoid this problem, do not include IDONE in calls to SMD_$BLT_U or use SMD_$EVENT_WAIT_U to wait for BLT completion.

To unmap the display memory, use SMD_$UNMAP_DISPLAY_U. Display memory is automatically unmapped when SMD_$RETURN_DISPLAY_U is executed, or when you type CTRL/Q to exit from the program.

SMD_$MAP_DISPLAY_U returns an error status if the display has not been borrowed, or if the display is already mapped into the calling process' address space.

SMD _ $MOVE _ ABS _ U

Sets the current position for vector drawing.

FORMAT

```
SMD_$MOVE_ABS_U (column, line)
```

INPUT PARAMETERS

column
The number of the column to which the position is set. This is a 2-byte integer in the range 0 through 799.

line
The number of the line to which the position is set. This is a 2-byte integer in the range 0 through 1023.

USAGE

SMD _ $MOVE _ ABS _ U sets the current position, from which the next vector will be drawn.

Call SMD _ $VECTOR _ INIT _ U to initialize the vector drawing package before using this procedure.

No error checking is performed on the arguments, for optimal performance. Incorrect program operation occurs if a column or line value is outside the specified range.

SMD_$MOVE_KBD_CURSOR_U

Moves the keyboard cursor to a specified position.

FORMAT

```
SMD_$MOVE_KBD_CURSOR_U (position, status)
```

INPUT PARAMETERS

position
X and y coordinates, in SMD_$POS_T format, for the cursor position. This data type is 4 bytes long.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. .

USAGE

SMD_$MOVE_KBD_CURSOR_U moves the keyboard cursor to a new position.

If the cursor was previously removed from the display (via SMD_$CLEAR_KBD_CURSOR_U), this call re-enables it.

In the position parameter, valid line values are 0-1023 for portrait displays, or 0 to 799 for landscape displays, and valid column values are 0-799 for portrait displays, or 0 1023 for landscape displays. The values represent the position of the lower left point of the cursor.

The keyboard cursor is 8 bits wide and 13 bits high.

SMD_$MOVE_REL_U

Sets the current position for vector drawing.

FORMAT

```
SMD_$MOVE_REL_U (column, line)
```

INPUT PARAMETERS

column

The column number, relative to the current position, from which the new column position is computed. This is a 2-byte integer.

line

The line number, relative to the current position, from which the new line position is computed. This is a 2-byte integer.

USAGE

SMD_$MOVE_REL_U computes a new position based upon the current value and the supplied arguments.

Call SMD_$VECTOR_INIT_U to initialize the vector drawing package before using this procedure.

When the value for column is added to the current column number, the sum must be between 0 and 799 for portrait displays, or between 0 and 1023 for landscape displays. Similarly, when the value for line is added to the current line number, the sum must be between 0 and 1023 for portrait displays and between 0 and 799 for landscape displays.

No error checking is performed on the arguments for optimal performance. Incorrect program operation occurs if the computed column or line value is outside the specified range.

SMD_$OP_WAIT_U

Waits for the current scroll or BLT operation to complete.

FORMAT

```
SMD_$OP_WAIT_U
```

USAGE

SMD_$OP_WAIT_U waits for completion of the current scroll or BLT operation. When this routine returns, the program can safely reference display memory.

If no scroll or BLT operation is underway, this routine returns immediately.

SMD_$RETURN_DISPLAY_U

Returns control of the display to the Display Manager.

FORMAT

```
SMD_$RETURN_DISPLAY_U (unit, status)
```

INPUT PARAMETERS

unit
Unit number of the display to be returned. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$RETURN_DISPLAY_U returns control of the display to the Display Manager. After executing this procedure, the calling program can no longer use display driver calls.

If the display was mapped into the process' address space, this procedure unmaps it.

After execution of this procedure, the Display Manager updates the display to reflect all input, output, and scrolling operations that occurred for all pads and processes while the screen was under direct program control.

If SMD_$BORROW_DISPLAY_U has not yet been successfully executed, an error status is returned.

Currently, the only valid unit number is 1.

SMD_$SET_QUIT_CHAR

Defines the quit character.

FORMAT

```
SMD_$SET_QUIT_CHAR (character, status)
```

INPUT PARAMETERS

character
A single character that is the new quit character. This is a character variable.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

The default quit character is CTRL/Q.

SMD_$SET_TP_CURSOR

Changes normal cursor to touchpad cursor and moves it to the indicated position.

FORMAT

```
SMD_$SET_TP_CURSOR (unit, position, buttons)
```

INPUT PARAMETERS

unit
A 2-byte integer indicating which display unit to use.

position
The new position of the cursor, in SMD_$POS_T format. This data type is 4 bytes long.

buttons
A 2-byte integer containing device dependent information such as a function button code. If the locator device has no such additional data, then buttons should be zero.

USAGE

This call is for use by programs that process data from a locator device other than the touch pad, such as a tablet. SMD_$SET_TP_CURSOR should not be used with the touch pad.

If the keyboard cursor is currently displayed and the touch pad is enabled (see SMD_$TP_ENABLE and SMD_$MOVE_KBD_CURSOR_U) executing SMD_$SET_TP_CURSOR removes the keyboard cursor, and displays the touchpad cursor at the location denoted by position.

Currently, the only valid unit number is 1.

SMD_$SOFT_SCROLL_U

Starts horizontal or vertical scrolling on the screen, 2 raster lines at a time.

FORMAT

```
SMD_$SOFT_SCROLL_U (boundaries, direction, displacement, status)
```

INPUT PARAMETERS

boundaries
The x and y coordinates for the edges of the area to be scrolled, in SMD_$WINDOW_LIMITS_T format. This data type is 8 bytes long. In FORTRAN this is a four-element array of 2-byte integers.

direction
Direction in which to scroll in SMD_$DIRECTION_T format. This is a 2-byte integer. Possible values are: SMD_$UP, SMD_$DOWN, SMD_$LEFT, and SMD_$RIGHT.

displacement
Number of horizontal or vertical raster lines to scroll. This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$SOFT_SCROLL_U scrolls two raster lines at a time, until the number of lines scrolled is equal to the displacement.

Scrolling takes place only within the specified area. The rest of the display does not change.

This procedure starts the scrolling operation, then returns control to the calling program.

Scrolling replicates the two raster lines on the boundary opposite to the direction of scrolling (for example, if scrolling up, the two raster lines on the bottom of the scrolled area), and does not clear them. Therefore, if you want the scrolling operation to produce blank lines, you must make certain the lines that are replicated are blank. You can clear a section of the display explicitly using SMD_$CLEAR_WINDOW.

Lines scrolled beyond the stated boundaries are lost.

Because scrolling occurs two lines at a time, the display driver can "interlace" other tasks with soft scrolling. Thus, the program can call most other display driver routines while scrolling is underway. The program must not, however, reference display memory, and it must not call any of the following routines:

SMD_$DRAW_ABS_U
SMD_$DRAW_BOX_U
SMD_$DRAW_REL_U
SMD_$MOVE_ABS_U
SMD_$MOVE_REL_U
SMD_$RETURN_DISPLAY_U
SMD_$SOFT_SCROLL_U

The display driver waits for scrolling to complete before executing one of these procedures. Calls to SMD_$BLT_U are executed only if the display mode does not include IDONE. If the display mode value includes IDONE, the driver waits for the current scrolling operation to complete before starting the BLT. Attempting a BLT with any part of its source or destination in the scrolled area is not recommended.

The program must not reference display memory while scrolling is underway. The program can call SMD_$EVENT_WAIT_U to find out when the completion of scrolling is imminent, and can then prepare data for another display operation. After preparing the data, the program can call SMD_$OP_WAIT to wait until references to display memory are safe.

SMD_$STOP_TP_CURSOR

Turns off the touch pad cursor and puts back the blinking cursor, if the blinking cursor would otherwise be displayed.

FORMAT

```
SMD_$STOP_TP_CURSOR (unit)
```

INPUT PARAMETERS

unit

A 2-byte integer indicating which display unit to use.

USAGE

This call is for use only by programs that process data from a locator device other than the touchpad. SMD_$SET_TP_CURSOR should not be used with the touchpad.

Currently, the only valid unit number is 1.

If the touchpad cursor is currently displayed, it is replaced with the blinking keyboard cursor.

SMD_$TP_DIRECT

Controls whether locator device data directly controls the touch pad cursor or is sent to the user program.

FORMAT

```
SMD_$TP_DIRECT (on-off, status)
```

INPUT PARAMETERS

on-off

A Boolean (logical) variable indicating whether to send the locator device directly to the program (TRUE) or to the internal display driver routine that controls the touchpad cursor (FALSE).

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

If the value of on-off is FALSE, (the initial state) locator device (for example, touchpad) data causes the touchpad cursor to move, as long as the touchpad cursor and keyboard cursor are enabled (see SMD_$TP_DISABLE). Locator device data are delivered through SMD_$EVENT_WAIT as SMD_$TPAD_AND_INPUT, but only when keystrokes are also delivered.

If the on-off parameter is TRUE, the touchpad cursor is not displayed in response to locator device data, and data is delivered in a continuous stream through SMD_$EVENT_WAIT as SMD_$TPAD_DATA.

If the on-off parameter is TRUE, locator data is always delivered, regardless of whether or not the touchpad cursor or keyboard cursor is enabled.

SMD_$TP_DISABLE

Prevents locator device data from moving the touchpad cursor.

FORMAT

```
SMD_$TP_DISABLE (status)
```

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

This call is for programs that modify display memory directly while the keyboard cursor is displayed. The touchpad cursor can interfere with display memory modifications because its location is unknown to the user program. SMD_$TP_DISABLE prevents display of the touchpad cursor.

This call also prevents display of the keyboard cursor if the touchpad has moved it and the user program has not yet been given the new position though a SMD_$EVENT_WAIT_U call.

The SMD vector drawing routines modify display memory directly. Therefore, SMD_$TP_DISABLE should be called before calling the vector routines, if the keyboard cursor is displayed at the same time the vectors are drawn.

SMD routines that modify display memory directly, other than the vector drawing routines (SMD_$DRAW_REL_U and SMD_$DRAW_ABS_U), automatically disable the touchpad cursor when they begin executing and re-enable the cursor when they finish.

In many cases the keyboard cursor is cleared (removed from the display using SMD_$CLEAR_KBD_CURSOR_U) before display modifications are made. This call is not needed in such cases.

The touchpad cursor is initially disabled.

SMD_$TP_ENABLE

Allows the touch pad cursor to be displayed and moved around the screen.

FORMAT

```
SMD_$TP_ENABLE (status)
```

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

This call enables display of the touchpad cursor. The touchpad cursor can be disabled by calling SMD_$TP_DISABLE.

The touchpad cursor is initially disabled.

In order for the touchpad or other locator device to move the cursor, three conditions must be satisfied:

1. SMD_$MOVE_KBD_CURSOR must be called to display the cursor in the first place.

2. SMD_$TP_ENABLE must be called to allow the locator device to affect the cursor.

3. On_off, an input parameter of SMD_$TP_DIRECT, must be FALSE. (See SMD_$TP_DIRECT.) Your program must make explicit calls to satisfy 1 and 2. The third condition is satisfied by default.

SMD_$UNLOAD_FONT_FILE_U

Unloads a font file from display memory.

FORMAT

```
SMD_$UNLOAD_FONT_FILE_U (font-id, status)
```

INPUT PARAMETERS

font-id
The internal identifier assigned to the font to be unloaded. This 2-byte value is returned by SMD_$LOAD_FONT_FILE_U.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$UNLOAD_FONT_FILE_U unloads a font that was loaded with SMD_$LOAD_FONT_FILE_U. Following this call, the font is no longer usable in calls to SMD_$WRITE_STRING.

An error is returned if the font is not loaded in display memory, or if the associated font file is not mapped into the process' address space.

SMD_$UNLOAD_FONT_U

Unloads a font from display memory.

FORMAT

```
SMD_$UNLOAD_FONT_U (font-id, status)
```

INPUT PARAMETERS

font-id
The internal identifier assigned to the font to be unloaded. This 2-byte value is returned by SMD_$LOAD_FONT_U.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

This procedure unloads the specified font, making it unavailable for use. The program must reload the font before using it again.

Use SMD_$UNLOAD_FONT_U for fonts loaded with SMD_$LOAD_FONT_U. Use SMD_$UNLOAD_FONT_FILE_U for font files loaded with SMD_$LOAD_FONT_FILE_U.

SMD_$UNMAP_DISPLAY_U

Unmaps display memory from the process' address space.

FORMAT

```
SMD_$UNMAP_DISPLAY_U (status)
```

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

SMD_$UNMAP_DISPLAY_U unmaps the display memory from the calling process' address space. Following this call, the 128K-byte portion of the address space onto which the display memory was mapped is no longer usable.

An error status is returned if the display has not been borrowed or if the display memory is not mapped when this call is made.

SMD_$VECTOR_INIT_U

Initializes the vector-drawing routines.

FORMAT

```
SMD_$VECTOR_INIT_U (display-address)
```

INPUT PARAMETERS

display-address

Starting address of the display memory in the program's address space. This is a 4-byte integer. This value is returned by SMD_$MAP_DISPLAY_U.

USAGE

SMD_$VECTOR_INIT_U initializes the vector-drawing routines supplied with the display driver. These routines are named SMD_$DRAW_ABS_U, SMD_$DRAW_REL_U, SMD_$MOVE_ABS_U, and SMD_$MOVE_REL_U. You must use SMD_$VECTOR_INIT_U once before calling any vector-drawing routines.

The vector-drawing routines operate incorrectly if the value of display-address differs from that returned by SMD_$MAP_DISPLAY_U.

The current position is set to line 0 and column 0. If you call a vector drawing routine before calling a position moving routine, the display driver draws a vector from the current position. All position-moving and vector-drawing routines update the current position.

SMD_$WRITE_STRING_U

Displays a string of text on the screen.

FORMAT

```
SMD_$WRITE_STRING_U (position, font_id, string, length, waitflag, status)
```

INPUT PARAMETERS

position
Row and column positions for the first character of the string, in SMD_$POS_T format. This data type is 4 bytes long.

font_id
Internal font identifier, returned by SMD_$LOAD_FONT_U.

string
String of ASCII text to be displayed. This is an array of up to 120 characters.

length
Length, in bytes, of the string to be displayed. This is a 2-byte integer.

waitflag
Boolean (logical) value indicating whether to wait for scrolling to complete before displaying the string.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long.

USAGE

This procedure displays a character string on the screen.

The ordinal (ASCII) value of each character in the input string is used as an index into the font denoted by font-id. For instance, the character "A" in the input string causes the output of the 60th character in the font. If no 60th character is defined in the font, the character "A" is ignored. No error occurs and no graphic character is displayed. The cursor is moved to the right an amount equal to the size of one space character.

The position parameter defines the base from which the first character is written. For the standard font, valid line values are 0 to 792 and valid column values are 12 to 1019 for a portrait display; for a landscape display, valid line values are 0 to 1016 and valid column values are 12 to 795.

The ordinal (ASCII) values of the characters in the string must be in the range 0 through 127.

Set the value of waitflag to TRUE if the string is to be displayed within an area that is being scrolled (via SMD_$SOFT_SCROLL_U) and FALSE otherwise.

SMD_$WRITE_STRING does not clear characters from the portion of the display where the string is written. Since characters may not fill the entire character box, pieces of previous characters may appear along with the string you wish to write. SMD_$CLEAR_WINDOW can be used to clear a section of the display.

ERRORS

SMD_$ACCESS_DENIED
Display borrow request denied by screen manager.

SMD_$ALREADY_ACQUIRED
Display already acquired.

SMD_$ALREADY_BORROWED
Display already borrowed by this process.

SMD_$ALREADY_MAPPED
Display memory is already mapped.

SMD_$BORROW_ERROR
Error borrowing display from screen manager.

SMD_$CANT_BORROW_BOTH
Cannot borrow both displays simultaneously.

SMD_$DISP_ACQD
Pad/stream operations not allowed while display acquired.

SMD_$DISPLAY_IN_USE
Unable to borrow: display in use.

SMD_$DISPLAY_MAP_ERROR
Error-mapping display memory.

SMD_$FONT_NOT_LOADED
Specified font is not loaded.

SMD_$FONT_NOT_MAPPED
Font associated with specified ID is not mapped.

SMD_$FONT_TABLE_FULL
Internal font table is full.

SMD_$FONT_TOO_LARGE
Font too large.

SMD_$HDM_FULL
Hidden display memory is full.

SMD_$HDMT_UNLOAD_ERR
Error unloading internal (HDMT) table.

SMD_$ILLEGAL_CALLER
Invalid use of display driver procedure.

SMD_$ILLEGAL_DIRECTION
Invalid direction from SM.

SMD_$ILLEGAL_UNIT
Invalid display unit number.

SMD_$INVALID_BLT_COORD
Invalid screen coordinates in BLT request.

SMD_$INVALID_BLT_CTL
Invalid BLT control register.

SMD_$INVALID_BLT_MODE
Invalid BLT mode register.

SMD_$INVALID_BLTD_INT
Invalid BLT-done interrupt.

SMD_$INVALID_BUFFER_SIZE
Invalid buffer size.

SMD_$INVALID_CRSR_NUMBER
Invalid cursor number.

SMD_$INVALID_DIRECTION
Invalid direction argument.

SMD_$INVALID_DISPLACEMENT
Invalid scroll displacement argument.

SMD_$INVALID_IR_STATE
Invalid interrupt routine state.

SMD_$INVALID_KEY
Invalid eventcount key.

SMD_$INVALID_LENGTH
Invalid length argument.

SMD_$INVALID_POS
Invalid position argument.

SMD_$INVALID_WID
Invalid DM window ID.

SMD_$INVALID_WINDOW
Invalid window limits argument.

SMD_$NO_MORE_WIDS
No more direct mode window IDs are available.

SMD_$NOT_BORROWED
Cannot return: display not borrowed.

SMD_$NOT_IMPLEMENTED
Nonconforming and main memory BLTs are not implemented.

SMD_$NOT_MAPPED
Display memory is not mapped.

SMD_$NOT_ON_COLOR
Operation not implemented on color display.

SMD_$PROCESS_NOT_FOUND
Process not found.

SMD_$PROTOCOL_VIOL
Internal protocol violation.

SMD_$QUIT_WHILE_WAITING
Quit while waiting.

SMD_$RETURN_ERROR
Error returning display to screen manager.

SMD_$TOO_MANY_PAGES
Too many pages to be wired.

SMD_$UNEXP_BLT_INUSE
Unexpected BLT in use.

SMD_$UNSUPPORTED_FONT_VE
Unsupported font version number.

SMD_$WAIT_QUIT
Quit while waiting.

SMD_$WINDOW_OBSCURED
Acquire denied because window is obscured.

STATUS_$OK
Successful completion.

STREAM

This section describes the data types, the call syntax, and the error codes for the STREAM programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

STREAM_$MAX	127	Maximum number of possible stream IDs.
STREAM_$NO_STREAM		Place-holder for stream ID when passing streams.
STREAM_$SUBS		Subsys component of status return denoting streams.
STREAM_$DIR_ENTTYPE_FILE		The file-type value for the enttype field of the DIR_ENTRY_T record.
STREAM_$DIR_ENTTYPE_LINK		The link-type value for the enttype field of the DIR_ENTRY_T record.
STREAM_$DIR_ENTRY_SIZE		Size of a directory entry record.

The following are mnemonic definitions used to specify attributes in the inquire and redefine input mask. The attributes followed by an asterisk (*) are attributes to which a stream must be open for information to be returned on an inquire. The attributes followed by a plus sign (+) are attributes which attributes cannot be redefined.

MNEMONIC	Bit	Explanation
STREAM_$STRID	0	Stream ID. +
STREAM_$OBJ_NAME	1	Object name.
STREAM_$OBJ_NAMLEN	1	Object name length.
STREAM_$REC_LGTH	2	Record length.
STREAM_$TEMPORARY	3	Temporary or permanent. *
STREAM_$EXPLICIT_TYPE	4	Explicit record type.
STREAM_$AB_FLAG	5	ASCII or binary file.
STREAM_$EXPLICIT_ML	6	Explicit move mode. *
STREAM_$CC	7	Carriage control.
STREAM_$REC_TYPE	8	Record type.
STREAM_$CONC	9	Object concurrency.
STREAM_$OCONC	10	Concurrency at open. *
STREAM_$OPOS	11	Access type. *
STREAM_$PRE_EXIST	12	Pre-existing object. +
STREAM_$HDR_LGTH	13	Header length. +
STREAM_$FILE_LENGTH	14	File length. +

STREAM_$SEEK_KEY	15	Seek key. * +
STREAM_$CUR_REC_LEN	16	Current record length. * +
STREAM_$CUR_REL_REC_NO	17	Current relative record number. * +
STREAM_$BLKS_USED	18	Number of blocks used. +
STREAM_$DTU	19	Date and time last used. +
STREAM_$DTM	20	Date and time last modified. +
STREAM_$SPARSE	21	Sparsely written file. * +
STREAM_$OTYPE	22	Object type.
STREAM_$CLOSE_ON_EXEC	23	Close stream on DOMAIN/IX Exec call.
STREAM_$NDELAY	24	Forced STREAM_$GET_CONDITIONAL.
STREAM_$APPEND_MODE	25	File in append mode.
STREAM_$FORCED_LOCATE	26	Force locate mode.

DATA TYPES

STREAM_$PARM1_T

A 2-byte integer. Specifies the type of data on which the seek is being performed. One of the following pre-defined values:

STREAM_$KEY
Seek with key returned earlier by stream manager.

STREAM_$REC
Record-oriented seek.

STREAM_$CHR
Character-oriented seek.

STREAM_$EOF
Seek to the end-of-file.

STREAM_$PARM2_T

A 2-byte integer. Specifies the type of seek being performed. One of the following pre-defined values:

STREAM_$RELATIVE
Seek relative to current position.

STREAM_$ABSOLUTE
Seek relative to BOF or EOF.

STREAM_$OPOS_T	A 2-byte integer. Specifies the access type of an object on open/create. One of the following pre-defined values: STREAM_$READ Open/create for read-only access. STREAM_$WRITE Create (new) for write access. STREAM_$OVERWRITE Write access; truncate file to BOF if it already exists. STREAM_$UPDATE Write access; file may already exist; position to start of file on open. STREAM_$APPEND Write access; if file already exists, position to EOF on open. STREAM_$MAKE_BACKUP Create new file: rename existing file to .BAK on close.
STREAM_$OMODE_T	A 2-byte integer. Specifies the concurrency at open of an object. One of the following pre-defined values: STREAM_$NO_CONC_WRITE Allows no concurrent writers to open file while this stream is open. STREAM_$CONTROLLED_SHARING Currently the same as NO_CONC_WRITE. STREAM_$REGULATED Allows unrestricted reading and writing of the file.
STREAM_$FCONC_T	A 2-byte integer. Specifies the object concurrency. One of the following pre-defined values: STREAM_$N_OR_1 Allows N readers or 1 writer in file concurrently. STREAM_$N_AND_1 Allows N readers AND up to 1 writer concurrently.

	STREAM_$N_AND_N Allows any number of writers or readers concurrently. STREAM_$STRICT_N_OR_1 Disallows multiple writers even when they are in a process family.
STREAM_$CC_T	A 2-byte integer. Specifies the type of carriage control employed in the object. One of the following pre-defined values: STREAM_$CC_T ASCII ("Apollo standard") carriage control. STREAM_$F&&_CC Fortran-77 standard (column 1) carriage control.
STREAM_$RTYPE_T	A 2-byte integer. Specifies the record structure of the object. One of the following pre-defined values: STREAM_$V1 Variable length records with count fields. STREAM_$F2 Fixed-length records. STREAM_$UNDEF No record structure in data.
STREAM_$IR_OPT	A 2-byte integer. Specifies method for accessing attribute record. One of the following pre-defined values: STREAM_$USE_STRID Use the stream-id to access the attribute record. STREAM_$NAME_CONDITIONAL Inquire is about the filename; only return information if file is open. STREAM_$NAME_UNCONDITIONAL Use the filename to access the attribute block.
STREAM_$INQUIRE_MASK_T	A 2-byte integer. Attributes to inquire. Specify any combination of the mnemonic constants for object attributes.
STREAM_$REDEF_MASK_T	A 2-byte integer. Attributes to redefine. Specify any combination of the valid mnemonic constants for object attributes.

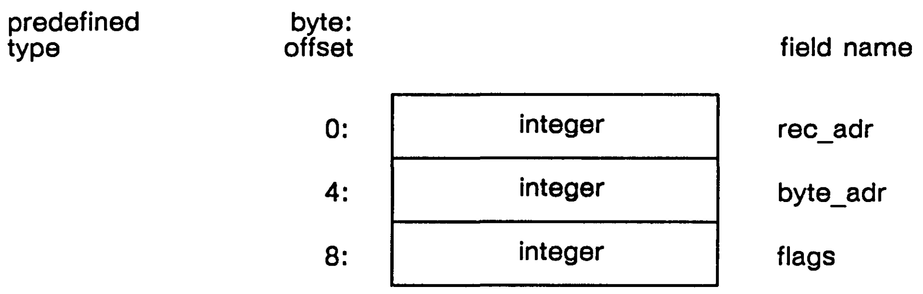
STREAM_$EC_KEY_T	A 2-byte integer. Specifies an eventcount to get. One of the following pre-defined values: STREAM_$GETREC_EC_KEY Stream eventcount key. STREAM_$EDIT_WAIT_EC_KEY Edit pad eventcount key.

{ options avail at open time, through stream_$opt_open }

STREAM_$OPEN_OPTIONS_T	A 2-byte integer. Options available at open time.. One of the following pre-defined values: STREAM_$NO_DELAY Do not wait for I/O (currently applies only to opening pipes).
STREAM_$OPEN_OPTIONS_SET_T	A 2-byte integer. Options available at open time with the STREAM_$OPEN_OPT call. Currently the only option available is: STREAM_$NO_DELAY Do not wait for I/O (currently applies only to opening pipes).
STREAM_$IR_REC_T	Attribute record for INQUIRE and REDEFINE calls. The streams chapter of the *Programming With General System Calls* manual describes how to use the attribute record. The diagram below illustrates the STREAM_$IR_REC_T data type:
STREAM_$ID_T	A 2-byte integer. Open stream identifier.

STREAM_$SK_T

Seek key returned on most stream calls. The diagram below illustrates the STREAM_$SK_T data type:

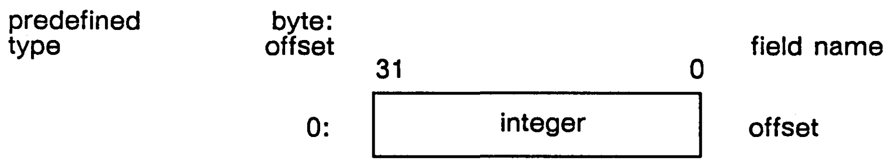


Field Description:

rec_adr
The address of the record sought.

byte_adr
The address of the byte sought.

flags
Flags containing seek information.



Field Description:

offset
The offset of the record or character sought.

predefined type	byte: offset		field name
	0:	integer	strid
	2:	integer	obj_namlen
	4:	integer	rec_lgth
	8:	integer	flags1*
	10:	integer	flags2*
	12:	integer	unused
	14:	integer	hdr_lgth
	16:	integer	file_lgth
stream_$sk_t	20:	integer	seek_key.rec_adr
	24:	integer	seek_key.byte_adr
	28:	integer	seek_key.flags
	32:	integer	cur_rec_len
	36:	integer	cur_rel_rec_no
	40:	integer	blks_used
time_$clockh_t	44:	integer	dtu
time_$clockh_t	48:	integer	dtm
uid_$t	52:	integer	otype.high
	56:	integer	otype.low
	60:	integer	flags3*
	62:	integer	flags4*
name_$name_t	64:	char	obj_name
	n:	char	

*see below for field names

strid
The stream ID of the object.

obj_namlen
The length of the object's name.

rec_lgth
The length of the longest record in the object.

flags1
A bit mask containing predefined or Boolean values indicating object attributes. The following table lists the bit numbers within the mask, the record field names, and a short decription of each attribute:

Bit #	*Field Name*	*Description*
Bit 0	temporary	Temporary or permanent object
Bit 1	explicit_type	Explicit fixed-length records
Bit 2	ab_flag	ASCII or binary file
Bit 3	explicit_ml	Explicit move mode
Bit 4-5	unused1	
Bit 6	cc	Type of carriage control
Bit 7-9	unused2	
Bit 10-11	rec_type	Record type
Bit 12-13	unused3	
Bit 14-15	conc	Object concurrency

flags2
A bit mask containing predefined or Boolean values indicating object attributes. The following table lists the bit numbers within the mask, the record field names, and a short decription of each attribute:

Bit #	*Field Name*	*Description*
Bit 0-1	unused4	
Bit 2-3	oconc	Concurrency at open
Bit 4-5	unused5	
Bit 6-8	opos	Access type
Bit 9	pre_exist	Pre-existing object
Bit 10-15	unused6	

hdr_lgth
The length of the object header.

file_lgth
The length of the file.

seek_key
The current seek-key.

cur_rec_len
The length of the current record.

cur_rel_rec_no
The current record number relative to BOF.

blks_used
The number of blocks occupied by the file.

dtu
The date and time of the last use of the object.

dtm
The date and time of the modification of the object.

otype
Specifies the type of the object.

flags3
A bit mask containing predefined or Boolean values indicating object attributes. The following table lists the bit numbers within the mask, the record field names, and a short decription of each attribute:

Bit #	*Field Name*	*Description*
`Bit 0`	`sparse`	`File may contain "holes."`
`Bit 1-15`	`unused7`	

flags4
A bit mask containing predefined or Boolean values indicating object attributes. The following table lists the bit numbers within the mask, the record field names, and a short decription of each attribute:

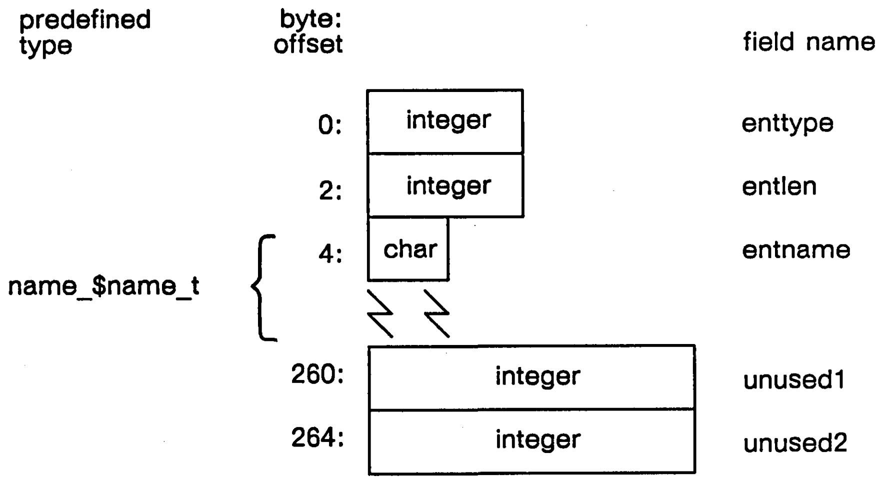
Bit #	*Field Name*	*Description*
`Bit 0-12`	`unused8`	
`Bit 13`	`close_on_exec`	`Close stream on UNIX Exec call`
`Bit 14`	`ndelay`	`Forced`

	STREAM_$GET_CONDITIONAL	
Bit 25	append_mode	File in append mode
Bit 26	forced_locate	Force locate mode

obj_name
The name of the object.

STREAM_$DIR_ENTRY_T

The directory entry returned by STREAM_$GET_REC. The diagram below illustrates the STREAM_$DIR_ENTRY_T data type:



Field Description:

enttype
Type of the directory entry. Either NAME_$FILE or NAME_$LINK.

entlen
Length of the directory entry name.

entname
Name of the directory entry.

unusedn
Reserved for future use by Apollo.

STREAM_$FORCE_WRITE_OPTIONS_T

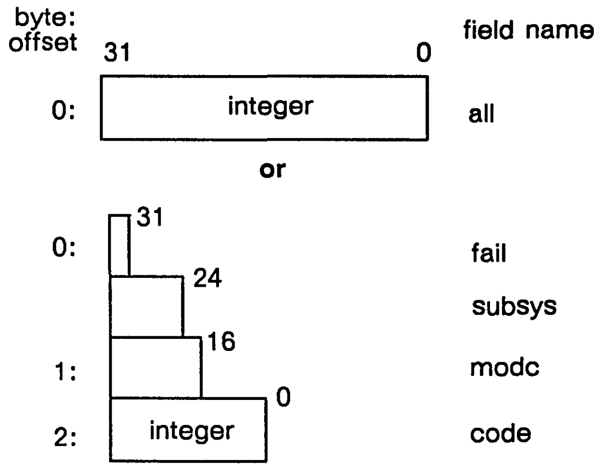
A 2-byte integer. Options available for force writing to disk. Any combination of the following pre-defined values:

STREAM_$FW_FILE
Specifies that a file should be force-written.

STREAM_$FW_DIR
Specifies that the directory of the file should be force-written.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

STREAM_$CLOSE

Closes a stream.

FORMAT

```
STREAM_$CLOSE (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of the stream to be closed, in STREAM_$ID_T format. This is a 2-byte integer.

The number used for stream identification becomes available for reuse. If the object is open on more than one stream, STREAM_$CLOSE closes only the specified stream.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$CLOSE closes the stream, so that you can no longer use the stream-id to operate on the object. Closing a stream to an object releases any locks maintained for the current user and may thus make the object available to other users.

If the stream is a disk file opened for any type of write access (STREAM_$WRITE, STREAM_$OVERWRITE, STREAM_$APPEND, or STREAM_$UPDATE), STREAM_$CLOSE updates its header, reflecting any changes made to the file while it was open, and indicating the date and time of last use and modification.

A program can close only the streams it has opened, and those opened by programs it has invoked (that is, opened at lower levels). Trying to close a stream opened at a higher level produces an error status code.

Closing a temporary object deletes it if no other process is using it.

STREAM_$CREATE

Creates an object (if the object does not already exist) and opens a stream to it.

FORMAT

```
STREAM_$CREATE (pathname, name-length, access, concurrency, stream-id, status)
```

INPUT PARAMETERS

pathname
Name of the object to be created, in NAME_$PNAME_T format. This is a character array of up to 256 elements.

name-length
Length of the pathname, in bytes. This is a 2-byte integer. To create a temporary object, specify a length of 0.

access
Type of access requested, in STREAM_$OPOS_T format. Possible values are:

STREAM_$APPEND	Permits adding data to the end of an object. The stream pointer points to the end of the object (EOF).
STREAM_$MAKE_BACKUP	Creates a temporary file, with the same type and attributes as the file specified in the pathname. This access is used to create a backup file. (See below for a detailed description.)
STREAM_$OVERWRITE	Permits replacing the entire contents of an object. The stream pointer is positioned at the start of the object data and data is truncated.
STREAM_$UPDATE	Permits replacing selected portions of the contents of an object. The stream pointer is positioned at the start of the object data, just past the header if it has one.
STREAM_$WRITE	Permits writing data to a new object. If writing is attempted on an existing object, an error status is returned.

If you specify the access option STREAM_$WRITE, the pathname must refer to a new object; otherwise, an error status is returned.

If you specify the access option STREAM_$MAKE_BACKUP, a new, unnamed temporary file is created by this call, which has the same type and other attributes as the file given by the pathname (if it exists). The new file is created on the same volume (i.e. the same node) as the file given by the pathname. The file given by the pathname is not opened or modified in this case, but is examined to extract its attributes. Even though the existing file is not modified, it is conceptually being replaced, so this operation requires write access to the file.

The application then writes the new file, and when it is closed (by the STREAM_$CLOSE call) the name of the file given at create time is changed to pathname.BAK. The new (formerly unnamed temporary) file gets the old name, and becomes permanent.

If the ".bak" file already exists, it is deleted. (The caller must have either D or P rights to delete the file.) If the ".bak" file is locked at the time STREAM_$CLOSE is called, it is deleted when it is unlocked.

If the pathname mentioned in the create call does not exist, then an ordinary STREAM_$CREATE is done, as though the access option had been STREAM_$WRITE instead of STREAM_$MAKE_BACKUP.

concurrency

Requested concurrency at open, in STREAM_$OMODE_T format. Possible values are:

STREAM_$CONTROLLED_SHARING	No concurrent writing.
STREAM_$NO_CONC_WRITE	No concurrent writing.
STREAM_$UNREGULATED	Unregulated read and write access.

OUTPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

If the pathname specifies an object that does not exist, the stream manager creates a new UASC disk file with that pathname and opens a stream to it.

If the pathname specifies an existing object, the stream manager opens a stream to it for overwrite, update, or append access. If write access is specified for an existing object, an error status is returned.

STREAM_$CREATE can open existing objects of any type, but can create only disk files.

Default attributes for a new disk file are listed below:

Attribute	*Default Value*
Data type	ASCII
Record type	UASC file format
Location	Lowest level in the directory pathname. If no pathname is specified, assume the current working directory.
Concurrency	No default.
Object concurrency	One writer or any number of readers.
Carriage control	DOMAIN standard.

If the object already exists, its attributes remain the same when it is opened. For streams to serial lines, however, "cooked" input mode and NO_WAIT are always in effect when the stream is opened. To change the object's attributes, call STREAM_$REDEFINE (or SIO_$CONTROL) before writing the object.

Both STREAM_$CREATE and STREAM_$OPEN open a stream to an object. However, STREAM_$CREATE creates the object if it does not exist, whereas STREAM_$OPEN returns an error if the object does not exist.

STREAM_$CREATE_BIN

Creates a binary record-structured file (if the file does not already exist) and opens a stream to it.

FORMAT

```
STREAM_$CREATE_BIN (pathname, name-length, access, concurrency, stream-id,
                    status)
```

INPUT PARAMETERS

pathname

Name of the object to be created, in NAME_$PNAME_T format. This is a character array of up to 256 elements.

name-length

Length of the pathname, in bytes. This is a 2-byte integer. To create a temporary object, specify a length of 0.

access

Type of access requested, in STREAM_$OPOS_T format. Possible values are:

STREAM_$APPEND
: Permits adding data to the end of an object. The stream pointer points to the end of the object (EOF).

STREAM_$OVERWRITE
: Permits replacing the entire contents of an object. The stream pointer is positioned at the start of the object data and data is truncated.

STREAM_$UPDATE
: Permits replacing selected portions of the contents of an object. The stream pointer is positioned at the start of the object data, just past the header if it has one.

STREAM_$WRITE
: Permits writing data to a new object. If writing is attempted on an existing object, an error status is returned.

concurrency

Requested concurrency at open, in STREAM_$OMODE_T format. Possible values are:

STREAM_$CONTROLLED_SHARING
: No concurrent writing.

STREAM_$NO_CONC_WRITE
: No concurrent writing.

STREAM_$UNREGULATED
: Unregulated read and write access.

OUTPUT PARAMETERS

stream-id
Number of the stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

If the named object does not exist, the stream manager creates a binary file of fixed length records and opens a stream to it.

If the named object already exists, the file attributes remain the same and the stream manager opens a stream to it for overwrite, update, or append access. If write access is specified for an existing object, an error status is returned.

To change the file's attributes, call STREAM_$REDEFINE (or SIO_$CONTROL) before writing the object.

Both STREAM_$CREATE_BIN and STREAM_$OPEN open a stream to a file. However, STREAM_$CREATE_BIN creates the file if it does not exist, whereas STREAM_$OPEN returns an error if the file does not exist.

STREAM_$CREATE_BIN differs from STREAM_$CREATE in that STREAM_$CREATE creates a UASC file by default.

STREAM_$CREATE_HERE

Creates an object at the specified location and opens a stream to it.

FORMAT

```
STREAM_$CREATE_HERE (pathname, name-length, access, concurrency,
                     loclen, locname, stream-id, status)
```

INPUT PARAMETERS

pathname
Name of the object to be created, in NAME_$PNAME_T format. This is a character array of up to 256 elements.

name-length
Length of the pathname, in bytes. This is a 2-byte integer. To create a temporary object, specify a length of 0.

access
Type of access requested, in STREAM_$OPOS_T format. Possible values are:

STREAM_$APPEND
Permits adding data to the end of an object. The stream pointer points to the end of the object (EOF).

STREAM_$OVERWRITE
Permits replacing the entire contents of an object. The stream pointer is positioned at the start of the object data and data is truncated.

STREAM_$UPDATE
Permits replacing selected portions of the contents of an object. The stream pointer is positioned at the start of the object data, just past the header if it has one.

STREAM_$WRITE
Permits writing data to a new object. If writing is attempted on an existing object, an error status is returned.

concurrency
Requested concurrency at open, in STREAM_$OMODE_T format. Possible values are:

STREAM_$CONTROLLED_SHARING
No concurrent writing.

STREAM_$NO_CONC_WRITE
No concurrent writing.

STREAM_$UNREGULATED
Unregulated read and write access.

loclen
Length of locname, in bytes. This is a 2-byte integer.

locname

Location at which to create the object, in NAME_$PNAME_T format. This is a character array of up to 256 elements.

The location can be a tree name or a leaf name.

OUTPUT PARAMETERS

stream-id

Number of the stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

This call creates an object at a specified location. It is especially useful for creating a temporary file on the same logical volume as an existing object.

If the pathname specifies an object that does not exist, the locname must specify the parent directory for the new object. If the pathname specifies an existing object, the locname and loclen are ignored.

If both the object and the location pathnames are valid, the stream manager opens a stream to the object for overwrite, update, or append access. If write access is specified for an existing object, an error status is returned.

STREAM_$CREATE_HERE can open existing objects of any type, but can create only disk files.

Both STREAM_$CREATE_HERE and STREAM_$OPEN open a stream to an object. STREAM_$CREATE_HERE, like STREAM_$CREATE, creates the object if it does not exist, whereas STREAM_$OPEN returns an error if the object does not exist.

STREAM_$DELETE

Deletes an object and closes the associated stream.

FORMAT

```
STREAM_$DELETE (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

An error occurs if the stream is open for read access only.

An error status is returned if the stream is not open.

USAGE

STREAM_$DELETE deletes the object, then closes the specified stream.

If the object cannot be deleted, an error occurs and all streams associated with the object remain open. Input pads cannot be deleted.

If the object is open on more than one stream, STREAM_$DELETE deletes the object causing "object deleted" errors when other streams try to read or write the object.

Files or pads are deleted immediately, even if several processes have opened the object.

For serial lines and magnetic tape files, this call operates exactly like STREAM_$CLOSE.

STREAM_$FORCE_WRITE_FILE

Forcibly writes a disk file open on the given stream.

FORMAT

```
STREAM_$FORCE_WRITE_FILE (stream-id, options, status)
```

INPUT PARAMETERS

stream-id
The number of the stream on which the disk file is open, in STREAM_$ID_T format. This is a 2-byte integer.

options
The object types to be force-written, in STREAM_$FORCE_WRITE_OPTIONS_T format. Possible values are:

STREAM_$FW_FILE	Forces the file to disk.
STREAM_$FW_DIR	Forces the file's directory to disk.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

Programs can call STREAM_$FORCE_WRITE_FILE to ensure that the file is stored on disk before it is closed.

If a program is handling a critical file, it can call this routine on the open stream to force-write the file's directory, thereby ensuring that the file pointer is saved in the directory.

STREAM_$GET_BUF

Reads data from an object into a specified buffer.

FORMAT

```
STREAM_$GET_BUF (stream-id, bufptr, buflen, retptr, retlen, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the buffer into which the data may be read, in UNIV_PTR format. This is a 4-byte integer.

To obtain a value for bufptr, FORTRAN programs can use the IADDR function. Pascal programs can use the ADDR function. The buffer can be aligned on a byte boundary; therefore, the value of bufptr can be odd.

buflen
Number of bytes of data to be read. This is a 4-byte integer.

OUTPUT PARAMETERS

retptr
Pointer to the data returned, in UNIV_PTR format. This is a 4-byte integer.

Address the returned data only by using retptr. The stream manager may use "locate mode," in which it doesn't copy the desired data to the location indicated by bufptr. FORTRAN programs that call the stream manager in locate mode should use the "pointer variable" FORTRAN extension.

The value of retptr is meaningful only until execution of the next stream call on this stream.

retlen
Number of bytes of data returned. This is a 4-byte integer.

seek-key
Unique key identifying the location of the data read, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

To obtain a seek-key value for the current stream position, call STREAM_$GET_BUF with a buflen of 0.

If the returned status is nonzero, the seek-key may not be useful.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

For UASC files, STREAM_$GET_BUF returns the requested number of bytes, including newline characters. UASC file records are delimited by the line-feed character (16#0A).

For non-UASC files, STREAM_$GET_BUF functions the same as STREAM_$GET_REC. That is, for fixed- or variable-length records STREAM_$GET_BUF returns one record, and for nonrecord-structured files STREAM_$GET_BUF returns the requested number of characters.

FORTRAN programs using this procedure in locate mode should use the pointer variable FORTRAN extension. Otherwise, call STREAM_$REDEFINE to set move mode before using this procedure.

STREAM_$GET_CONDITIONAL

Reads a record if the record is available; otherwise, it sets the returned record length to zero.

FORMAT

```
STREAM_$GET_CONDITIONAL (stream-id, bufptr, buflen, retptr, retlen,
                         seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the buffer into which the data may be read, in UNIV_PTR format. This is a 4-byte integer.

To obtain a value for bufptr, FORTRAN programs can use the IADDR function. Pascal programs can use the ADDR function. The buffer can be aligned on a byte boundary; therefore, the value of bufptr can be odd.

buflen
Number of bytes of data requested. This is a 4-byte integer.

If the number of bytes remaining in the record is less than buflen, STREAM_$GET_CONDITIONAL returns the remainder of the record. If the number of bytes remaining in the record is greater than buflen, the stream manager reads enough data to fill the buffer and returns a negative value in retlen. The absolute value of retlen is the number of bytes remaining in the record.

OUTPUT PARAMETERS

retptr
Pointer to the data returned, in UNIV_PTR format. This is a 4-byte integer.

Address the returned data only by using retptr. The stream manager may use "locate mode," in which it doesn't copy the desired data to the location indicated by bufptr. FORTRAN programs that call the stream manager in locate mode should use the "pointer variable" FORTRAN extension.

Records are aligned on word boundaries. Therefore, if the procedure reads an entire record, the value of retptr will be word-aligned and positive. The value of retptr is meaningful only until execution of the next stream call for this stream.

retlen

Number of bytes of data actually returned. This is a 4-byte integer.

If the call returned any data, retlen has a value equal to the requested number of bytes. It has a value of 0 if the call returned no data.

seek-key

Unique key identifying the location of the data returned, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

If the returned status is nonzero, the seek-key is not useful.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$GET_CONDITIONAL performs read operations on streams such as SIO lines, input pads, and mailboxes, for which data may not yet be available at the time of the call. Under these conditions, STREAM_$GET_REC waits for data. STREAM_$GET_CONDITIONAL never waits. If data is not immediately available, it returns a length of zero.

This call is commonly used in conjunction with STREAM_$GET_EC and EC2_$WAIT.

Since data from ordinary files is always available, this call is equivalent to STREAM_$GET_REC for files.

No error occurs if the stream manager cannot find data at the current stream position, unless the current position is known to be at EOF. In this case, a zero is returned in retlen.

STREAM_$GET_EC

Gets the eventcount address of the eventcount to be advanced upon any activity within the specified stream.

FORMAT

```
STREAM_$GET_EC (stream-id, stream-key, eventcount-pointer, status)
```

INPUT PARAMETERS

stream-id
The stream ID, in STREAM_$ID_T format. This is a 2-byte integer.

stream-key
The key that specifies which eventcount to get, in STREAM_$EC_KEY_T format. The only value allowed is STREAM_$GETREC_EC_KEY.

OUTPUT PARAMETERS

eventcount-pointer
The eventcount address to be obtained, in EC2_$PTR_T format. EC2_$PTR_T is a pointer to an EC2_$EVENTCOUNT_T array.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

The eventcount is advanced whenever data becomes available through the stream. This call is valid for all streams, including those open to files, pads, mailboxes, and COMMENTs.

If the input pad is in raw mode then an event is counted after each single character stroke; if the keyboard is in cooked mode then an event is counted after each carriage return.

See the description of EC2_$WAIT for a description of eventcount data structures. See *Programming With General System Calls* for a discussion of eventcounts.

STREAM_$GET_PRIOR_REC

Reads the previous record.

FORMAT

```
STREAM_$GET_PRIOR_REC (stream-id, bufptr, buflen, retptr, retlen,
                       seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the buffer into which the data may be read, in UNIV_PTR format. This is a 4-byte integer.

To obtain a value for bufptr, FORTRAN programs can use the IADDR function. Pascal programs can use the ADDR function. The buffer can be aligned on a byte boundary; therefore, the value of bufptr can be odd.

buflen
Number of bytes of data requested. This is a 4-byte integer.

If the number of bytes remaining in the record is less than buflen, STREAM_$GET_PRIOR_REC returns the remainder of the record. The value returned in retlen is the number of bytes actually read. If the number of bytes remaining in the record is greater than buflen, the stream manager reads enough data to fill the buffer and returns a negative value in retlen. The absolute value of the returned retlen is the number of bytes remaining in the record.

OUTPUT PARAMETERS

retptr
Pointer to the data returned, in UNIV_PTR format. This is a 4-byte integer.

Address the returned data only by using retptr. The stream manager may use "locate mode," in which it doesn't copy the desired data to the location indicated by bufptr. FORTRAN programs that call the stream manager in locate mode should use the "pointer variable" FORTRAN extension.

Records are aligned on word boundaries. Therefore, if the procedure reads an entire record, the value of retptr will be word-aligned and positive. The value of retptr is meaningful only until execution of the next stream call on this stream.

retlen
Number of bytes of data returned. This is a 4-byte integer.

If the number of bytes remaining in the record is less than buflen, STREAM_$GET_PRIOR_REC returns the remainder of the record. The value of retlen is the number of bytes actually read.

If the number of bytes remaining in the record is greater than buflen, the stream manager reads enough data to fill the buffer and returns a negative value in retlen. The absolute value of the returned retlen is the number of unread bytes remaining in the record.

seek-key

Unique key identifying the location of the data returned, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

The seek-key identifies the beginning of the returned data, as it does for STREAM_$GET_REC. Use it in STREAM_$SEEK calls followed by STREAM_$GET_REC (not STREAM_$GET_PRIOR_REC) calls.

If the returned status is nonzero, the seek-key is not useful.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$GET_PRIOR_REC reads the previous record from the object. The object must be open on the specified stream.

This call operates on file types UASC, PAD, HDR_UNDEF, and REC with record format F2. It will not work on REC files with record format V1, or on nonfile-type streams such as SIO lines, input pads, or mailboxes. STREAM_$GET_PRIOR_REC works as follows:

UASC and REC files

If the seek-key is positioned at the beginning of a record, it is positioned to the previous record. If the seek-key is in the middle of a record, then its position is not changed.

HDR_UNDEF files

The seek-key is repositioned by subtracting the caller's buffer size from the current position.

After these respective actions are taken, an ordinary STREAM_$GET_REC operation is done.

FORTRAN programs using this procedure in locate mode should use the pointer variable FORTRAN extension. Otherwise, call STREAM_$REDEFINE to set move mode before using this procedure.

An error occurs if the stream manager cannot find a record at the current stream position -- for example, if the current position is beyond EOF or at BOF (beginning of file).

STREAM_$GET_REC

Reads the next sequential record from an object.

FORMAT

```
STREAM_$GET_REC (stream-id, bufptr, buflen, retptr, retlen, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the buffer into which the record may be read, in UNIV_PTR format. This is a 4-byte integer.

To obtain a value for bufptr, FORTRAN programs can use the IADDR function. Pascal programs can use the ADDR function. The buffer can be aligned on a byte boundary; therefore, the value of bufptr can be odd.

buflen
Number of bytes of data to be read. This is a 4-byte integer.

If the number of bytes remaining in the record is less than buflen, STREAM_$GET_REC returns the remainder of the record. If the number of bytes remaining in the record is greater than buflen, the stream manager reads enough data to fill the buffer and returns a negative value in retlen. The absolute value of retlen is the number of bytes remaining in the record.

OUTPUT PARAMETERS

retptr
Pointer to the data returned, in UNIV_PTR format. This is a 4-byte integer.

Address the returned data only by using retptr. The stream manager may use "locate mode," in which it doesn't copy the desired data to the location indicated by bufptr. FORTRAN programs that call the stream manager in locate mode should use the "pointer variable" FORTRAN extension.

Records are aligned on word boundaries. Therefore, if the procedure reads an entire record, the value of retptr will be word-aligned and positive. The value of retptr is meaningful only until execution of the next stream call for this stream.

retlen
Number of bytes of data returned. This is a 4-byte integer.

If the number of bytes remaining in the record is less than buflen, STREAM_$GET_PRIOR_REC returns the remainder of the record. The value of retlen is the number of bytes actually read.

If the number of bytes remaining in the record is greater than buflen, the stream manager reads enough data to fill the buffer and returns a negative value in retlen. The absolute value of the returned retlen is the number of unread bytes remaining in the record.

seek-key

Unique key identifying the location of the data read, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

To obtain a seek-key value for the current stream position, call STREAM_$GET_REC with a buflen of 0.

Use the seek-key value in STREAM_$SEEK calls followed by STREAM_$GET_REC calls.

If the returned status is nonzero, the seek-key is not useful.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$GET_REC returns at most the requested number of bytes of the next sequential record in the object.

FORTRAN programs using this procedure in locate mode should use the pointer variable FORTRAN extension. Otherwise, call STREAM_$REDEFINE to set move mode before using this procedure.

An error occurs if the stream manager cannot find a record at the current stream position -- for example, if the current position is at EOF.

STREAM_$INQUIRE

Returns information about an object.

FORMAT

```
STREAM_$INQUIRE (input-mask, inquiry-type, attributes, error-mask, status)
```

INPUT PARAMETERS

input-mask

Integer bit mask indicating the attributes for which information is requested, in STREAM_$INQUIRE_MASK_T format. This is a 4-byte integer.

The following lists the predefined symbols for each bit position in the mask.

Bit 0	STREAM_$STRID	Stream-id
Bit 1	STREAM_$OBJ_NAME	Object name
Bit 1	STREAM_$OBJ_NAMLEN	Object name length
Bit 2	STREAM_$REC_LGTH	Record length
Bit 3	STREAM_$TEMPORARY	Temporary or permanent *
Bit 4	STREAM_$EXPLICIT_TYPE	Explicit record type
Bit 5	STREAM_$AB_FLAG	ASCII or binary file
Bit 6	STREAM_$EXPLICIT_ML	Explicit move mode *
Bit 7	STREAM_$CC	Carriage control
Bit 8	STREAM_$REC_TYPE	Record type
Bit 9	STREAM_$CONC	Object concurrency (not implemented)
Bit 10	STREAM_$OCONC	Concurrency at open *
Bit 11	STREAM_$OPOS	Access type *
Bit 12	STREAM_$PRE_EXIST	Pre-existing object
Bit 13	STREAM_$HDR_LGTH	Header length
Bit 14	STREAM_$FILE_LENGTH	File length
Bit 15	STREAM_$SEEK_KEY	Seek key *
Bit 16	STREAM_$CUR_REC_LEN	Current record length *
Bit 17	STREAM_$CUR_REL_REC_NO	Current relative record number *
Bit 18	STREAM_$BLKS_USED	Number of blocks used
Bit 19	STREAM_$DTU	Date and time last used
Bit 20	STREAM_$DTM	Date and time last modified
Bit 21	STREAM_$SPARSE	Sparsely written file *
Bit 22	STREAM_$OTYPE	Object type
Bit 23	STREAM_$CLOSE_ON_EXEC	Close stream on UNIX Exec call
Bit 24	STREAM_$NDELAY	Forced STREAM_$GET_CONDITIONAL
Bit 25	STREAM_$APPEND_MODE	File in append mode
Bit 26	STREAM_$FORCED_LOCATE	Force locate mode

* Attributes to which a stream must be open for information to be returned.

Pascal and C programs specify these predefined values as members of a set. FORTRAN programs must add the desired values to each other to result in a correct input-mask value.

inquiry-type

Type of inquiry, in STREAM_$IR_OPT format. Possible values are:

STREAM_$USE_STRID
: Specifies an inquiry by stream ID. On input, the attribute record must contain the stream-id to which the request applies. On output, the attribute record contains the requested information, if the stream is open. If the stream is not open, an error is returned.

STREAM_$NAME_CONDITIONAL
: Specifies an inquiry by name to be executed only if a stream is open to the object. On input, the attribute record must contain the object's pathname and name-length. On output, the attribute record contains the requested information if a stream is open. If no stream is open to the object, an error is returned.

STREAM_$NAME_UNCONDITIONAL
: Specifies an unconditional inquiry by name. On input, the attribute record must contain the object's pathname and name-length. On output, the attribute record contains the requested information whether or not a stream is open.

INPUT/OUTPUT PARAMETERS

attributes

Record containing attribute information, in STREAM_$IR_REC_T format. On input, this record contains a pathname or stream-id that identifies the object. On output, this record contains the returned information.

For serial lines, STREAM_$INQUIRE returns the default values.

The attribute parameter is able to convey a large amount of information by passing it in many fields of one record (including a number of bit fields). These record fields are listed below along with their size and a brief explanation of the information they transmit.

Stream-id	STREAM_$ID_T. A 2-byte integer. Specified on input in conjunction with the STREAM_$USE_STRID inquiry-type.
Name-length	a 2-byte integer. Name-length of the object. Specified on input in conjunction with the STREAM_$NAME_CONDITIONAL and STREAM_$NAME_UNCONDITIONAL inquiry-type.
Record length	A longword. If the object has variable-length records, the length of the longest record is returned; if fixed-length records, the fixed record length is returned.
Flag1	A field of 16 bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number	*Attribute*	*Data Type*	*Explanation*
Bit 0	Temporary	Boolean	TRUE if the object is temporary.
Bit 1	Explicit type	Boolean	TRUE if record type is explicit.
Bit 2	ASCII/Binary	Boolean	TRUE if data is ASCII, otherwise it is binary.
Bit 3	Force move mode	Boolean	TRUE if move mode is used (only applies to open streams).
Bits 4,5	Unused 1		
Bit 6	Carriage control	STREAM_$CC_T	Either STREAM_$F77_CC (FORTRAN) or STREAM_$APOLLO_CC (DOMAIN).
Bits 7-9	Unused 2		
Bits 10,11	Record Type	STREAM_$RTYPE_T	Either STREAM_$F2 (fixed length), STREAM_$V1 (variable-length), or STREAM_$UNDEF (undefined).
Bits 12,13	Unused 3		
Bits 14,15	Object Concurrency	STREAM_$FCONC_T	Not implemented. Always is STREAM_$N_AND_N.

Flag2	A word. This is a field of bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number Attribute	*Data Type*	*Explanation*
Bits 16,17 Unused 4		
Bits 18,19 Concurrency at open	STREAM_OMODE_T	Either STREAM_$UNREGULATED, STREAM_$NO_CONC_WRITE, or STREAM_$CONTROLLED_SHARING. Only returned from opened streams.
Bits 20,21 Unused 5		
Bits 2-24 Access Type	STREAM_$OPOS_T	Either STREAM_$READ, STREAM_$WRITE, STREAM_$UPDATE, STREAM_$APPEND, or STREAM_$OVERWRITE. Only returned from opened streams.
Bit 25 Pre-existing	Boolean	TRUE if object already exists.
Bits 26-31 Unused 5		

Unused 6

Header length	A 2-byte integer. Length of streams header.
File length	A 4-byte integer. Total file length, in bytes (including header).
Seek key	STREAM_$SK_T. A three-element INTEGER*4 array. Current stream position. This attribute is only returned from opened streams.
Current record length	A 4-byte integer. Size of current record. This attribute is only returned from opened streams.
Current relative record number	A 4-byte integer. Only applies to files with fixed-length records. This attribute is only returned from opened streams.
Blocks used	A 4-byte integer. Number of disk blocks currently used for the file. Only applies to pads and disk files.
Date/Time Used	TIME_$CLOCKH_T. A 4-byte integer. Date and time of last use. Only applies to pads and disk files.
Date/Time Modified	TIME_$CLOCKH_T. A 4-byte integer. Date and time of last modification. Only applies to pads and disk files.

Object type	UID_$T. A two-element INTEGER*4 array. Type UID of the object. The following table lists valid UID types.

UASC_$UID	UASC file
RECORDS_$UID	Record-structured file
HDR_$UNDEF_$UID	Nonrecord-structured file
OBJECT_FILE_$UID	Object module file
PAD_$UID	Saved transcript pad
INPUT_PAD_$UID	Input pad
SIO_$UID	Serial line descriptor file
MBX_$UID	Mailbox object
MT_$UID	Magnetic tape descriptor file

Sparse flag	Boolean. When TRUE, file allocation may have "holes."
Flag3	A word. This is a field of bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number	*Attribute*	*Data Type*	*Explanation*
Bits 16-27	Unused 7		
Bit 29	Close on exec	Boolean	TRUE causes stream to be closed upon an AUX exec call.
Bit 30	No delay mode	Boolean	When TRUE, any system call that reads data from a stream will act like STREAM_$GET_CONDITIONAL (returns if data not available).
Bit 31	Append mode	Boolean	When TRUE, any call to STREAM_$PUT_REC or STREAM_$PUT_CHR does a seek to EOF before writing any data.
Bit 28	Force locate mode	Boolean	Normally, if the force move mode bit (bit 6) is not set, streams may use either move mode or locate mode. If this bit is set, streams will only use locate mode, the caller need not supply a buffer. This option can only be set for file-type streams (UASC, REC, HDR_UNDEF, and CASE_HM).

Unused 8	A 4-byte integer.
Name	NAME_$PNAME_T. A character array of up to 256 elements. The name of the object. Specified on input with STREAM_$NAME_CONDITIONAL and STREAM_$NAME_UNCONDITIONAL inquiry-types.

The array specified as the attribute parameter need not be large enough for every field, but just sufficient to span the required fields. For example, to inquire on explicit move mode, only a six-element INTEGER*2 array is required (for a FORTRAN program), because the necessary flag is in FLAG1.

Accessing attribute record bit fields is discussed in detail in the *Programming With General System Calls.*

OUTPUT PARAMETERS

error-mask

Integer bit-mask indicating the requested fields that could not be returned, in STREAM_$INQUIRE_MASK_T format. This is a 4-byte integer.

This procedure may complete with partial success if it can return some, but not all, of the requested attributes. If an attribute is unavailable, STREAM_$INQUIRE sets the corresponding bit in the error mask and continues to inquire about other attributes. In cases of partial success, the returned status code is nonzero. The program must check the error mask to find out where the error occurred.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$INQUIRE returns the attributes of the object specified in the mask. To receive the information you must specify either the stream ID of the object or the object name and name-length. However, some attributes such as access type apply only to objects to which a stream is open. These parameters are marked with asterisks in the inquiry mask parameter description above.

The stream position does not change as a result of this call.

STREAM_$OPEN

Opens a stream to an existing object.

FORMAT

```
STREAM_$OPEN (pathname, name-length, access, concurrency, stream-id, status)
```

INPUT PARAMETERS

pathname

Name of the object to be opened, in NAME_$PNAME_T format. This is a character array of up to 256 elements.

name-length

Length of the pathname. This is a 2-byte integer.

access

Type of access requested, in STREAM_$OPOS_T format. Possible values are:

STREAM_$APPEND
: Permits adding data to the end of an object. The stream pointer points to the end of the object (EOF).

STREAM_$OVERWRITE
: Permits replacing the entire contents of an object. The stream pointer is positioned at the start of the object data and data is truncated.

STREAM_$READ
: Permits reading data from an existing object.

STREAM_$UPDATE
: Permits replacing selected portions of the contents of an object. The stream pointer is positioned at the start of the object data, just past the header if it has one.

STREAM_$WRITE
: Permits writing data to a new object. If writing is attempted on an existing object, an error status is returned.

If you specify the access option STREAM_$WRITE, the pathname must refer to a new object, otherwise an error status is returned.

concurrency

Requested concurrency at open, in STREAM_$OMODE_T format. Possible values are:

STREAM_$CONTROLLED_SHARING
: Does not allow concurrent read and write access.

STREAM_$NO_CONC_WRITE
: Does not allow concurrent read and write access.

STREAM_$UNREGULATED
: Allows concurrent read and write access.

OUTPUT PARAMETERS

stream-id
Number of the stream on which the object was opened, in STREAM_$ID_T format. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

This routine opens a stream to the named object and assigns access and concurrency types. It returns the stream ID to be used in subsequent stream activity with the object.

An error occurs if the object does not exist.

STREAM_$OPEN does not return information about the object's characteristics. Use STREAM_$INQUIRE to obtain that information.

If the object already exists, its attributes remain the same when it is opened. For streams to SIO lines, however, "cooked" input mode and NO_WAIT are always in effect when the stream is opened. To change the object's attributes, call STREAM_$REDEFINE (or SIO_$CONTROL) before writing to the object.

STREAM_$PUT_CHR

Writes data to an object without terminating the current record, if one exists.

FORMAT

```
STREAM_$PUT_CHR (stream-id, bufptr, buflen, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which an object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the data to be written, in UNIV_PTR format. This is a 4-byte integer.

FORTRAN programs can use IADDR to obtain the buffer address for the bufptr parameter. Pascal programs can use ADDR. Alternately, programs in either language can use pointer variables.

buflen
Number of bytes of data to be written. This is a 4-byte integer.

OUTPUT PARAMETERS

seek-key
Unique key identifying the location of the data written, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

The seek key allows random access to the output data by a subsequent STREAM_$SEEK call.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$PUT_CHR writes the specified number of bytes to the object, but does not terminate a record.

Use this procedure to write data in nonrecord-structured files or to compose records piece by piece. Be sure to call STREAM_$REDEFINE to set STREAM_$UNDEF as the record type, and HDR_UNDEF_$UID as the object type, before writing any output.

Records of fixed-length format automatically change to variable-length if this write operation extends the current record beyond the length of existing records. In this case, no error occurs. For files with explicit fixed-length records, an error occurs if this write operation extends the current record beyond the fixed-record size.

For files with variable-length records, no record length checking is performed. Therefore, take care not to alter the count field of the following record.

Record size can increase as a result of this call, but cannot decrease. For instance, after you overwrite the first 20 bytes of a 32-byte record, the last 12 bytes still contain the original data, and the count field remains the same. To terminate a record and update its count field, use STREAM_$PUT_REC.

STREAM_$PUT_CHR and STREAM_$PUT_REC operate identically when applied to SIO lines, UASC files, and keyboards.

STREAM_$PUT_REC

Writes data to an object and terminates the current record, if one exists.

FORMAT

```
STREAM_$PUT_REC (stream-id, bufptr, buflen, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the data to be written, in UNIV_PTR format. This is a 4-byte integer.

FORTRAN programs can use IADDR to obtain the buffer address for the bufptr parameter. Pascal programs can use ADDR. Alternately, programs in either language can use pointer variables.

buflen
Number of bytes of data to be written. This is a 4-byte integer.

For files with explicit fixed-length records, an error occurs if the total record length is not equal to the fixed-record length.

If you specify a buflen of zero, STREAM_$PUT_REC simply terminates the current record and updates its count field.

OUTPUT PARAMETERS

seek-key
Unique key identifying the location of the data in the object, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

The seek key allows random access to the output data by a subsequent STREAM_$SEEK call.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$PUT_REC queues data for output to the object. It does not necessarily write the data on the COMMENT. COMMENT writes may be performed asynchronously.

Records of default format (implicit fixed-length) automatically change to variable-length if the new record differs in length from any existing records. No error occurs.

Existing data in variable-length records are overwritten if the stream position is not at EOF. No error occurs.

STREAM_$PUT_REC never inserts newline characters in the object on its own. You must do this yourself if you want newlines to appear in the object.

STREAM_$PUT_CHR and STREAM_$PUT_REC operate identically when applied to SIO lines, UASC files, and keyboards.

STREAM_$REDEFINE

Changes one or more attributes of an object that is open on a stream.

FORMAT

```
STREAM_$REDEFINE (stream-id, input-mask, attributes, error-mask, status)
```

INPUT PARAMETERS

stream-id

Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

input-mask

An integer bit mask showing which attributes you want to redefine, in STREAM_$REDEF_MASK_T format. This is a 4-byte integer.

Bits 4 through 9 are the ones most commonly changed. The following lists the predefined symbols for each bit position in the mask.

Bit 1	STREAM_$OBJ_NAME	Object name
Bit 1	STREAM_$OBJ_NAMLEN	Object name length
Bit 2	STREAM_$REC_LGTH	Record length
Bit 3	STREAM_$TEMPORARY	Temporary or permanent
Bit 4	STREAM_$EXPLICIT_TYPE	Explicit record type
Bit 5	STREAM_$AB_FLAG	ASCII or binary file
Bit 6	STREAM_$EXPLICIT_ML	Explicit move mode
Bit 7	STREAM_$CC	Carriage control
Bit 8	STREAM_$REC_TYPE	Record type
Bit 9	STREAM_$CONC	Object concurrency (not implemented)
Bit 10	STREAM_$OCONC	Concurrency at open
Bit 11	STREAM_$OPOS	Access type
Bit 21	STREAM_$SPARSE	Sparsely written file
Bit 22	STREAM_$OTYPE	Object type
Bit 23	STREAM_$CLOSE_ON_EXEC	Close stream on DOMAIN/IX Exec call
Bit 24	STREAM_$NDELAY	Forced STREAM_$GET_CONDITIONAL
Bit 25	STREAM_$APPEND_MODE	File in append mode
Bit 26	STREAM_$FORCED_LOCATE	Force locate mode

Pascal and C programs specify these predefined values as members of a set. FORTRAN programs must add the desired values to each other to result in a correct input-mask value.

Note that some bit numbers are missing (0, 12 - 20). This is because STREAM_$INQUIRE and STREAM_$REDEFINE use the same attribute record, however certain attributes that can be inquired upon cannot be redefined.

attributes

Record containing new values for attributes, in STREAM_$IR_REC_T format.

The attribute parameter is able to specify redefinition of a large number of attributes by passing information in many fields of one record (including a number of bit fields). These record fields are listed below along with their size and a brief explanation of the information they transmit.

Stream-id	STREAM_$ID_T. A 2-byte integer. Not redefinable.
Name-length	a 2-byte integer. Name-length of the object. Specified on input in conjunction with the STREAM_$NAME_CONDITIONAL and STREAM_$NAME_UNCONDITIONAL inquiry-type.
Record length	A longword. If the object has variable-length records, the length of the longest record is returned; if fixed-length records, the fixed record length is returned.
Flag1	A field of 16 bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number	*Attribute*	*Data Type*	*Explanation*
Bit 0	Temporary	Boolean	TRUE if the object is temporary.
Bit 1	Explicit type	Boolean	TRUE if record type is explicit.
Bit 2	ASCII/Binary	Boolean	TRUE if data is ASCII, otherwise it is binary.
Bit 3	Force move mode	Boolean	TRUE if move mode is used. (only applies to open streams)
Bits 4,5	Unused 1		
Bit 6	Carriage control	STREAM_$CC_T	Either STREAM_$F77_CC (FORTRAN) or STREAM_$APOLLO_CC (DOMAIN).
Bits 7-9	Unused 2		
Bits 10,11	Record Type	STREAM_$RTYPE_T	Either STREAM_$F2 (fixed length), STREAM_$V1 (variable-length), or STREAM_$UNDEF (undefined).
Bits 12,13	Unused 3		
Bits 14,15	Object Concurrency	STREAM_$FCONC_T	Not implemented. Always STREAM_$N_AND_N.

Flag2	A word. This is a field of bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number	*Attribute*	*Data type*	*Explanation*
Bits 16,17	Unused 4		
Bits 18,19	Concurrency at open	STREAM_OMODE_T	Either STREAM_$UNREGULATED, STREAM_$NO_CONC_WRITE, or STREAM_$CONTROLLED_SHARING. Only returned from opened streams.
Bits 20,21	Unused 5		
Bits 2 -24	Access Type	STREAM_$OPOS_T	Either STREAM_$READ, STREAM_$WRITE, STREAM_$UPDATE, STREAM_$APPEND, or STREAM_$OVERWRITE. Only returned from opened streams.
Bit 25	Pre-existing	Boolean	Not redefinable.
Bits 26-31	Unused 5		

Unused 6

Header length A 2-byte integer. Not redefinable.

File length A 4-byte integer. Not redefinable.

Seek key STREAM_$SK_T. A three-element INTEGER*4 array. Not redefinable.

Current record length
A 4-byte integer. Not redefinable.

Current relative record number
A 4-byte integer. Not redefinable.

Blocks used A 4-byte integer. Not redefinable.

Date/Time Used TIME_$CLOCKH_T. A 4-byte integer. Not redefinable.

Date/Time Modified
TIME_$CLOCKH_T. A 4-byte integer. Not redefinable.

Object type UID_$T. A two element INTEGER*4 array. Type UID of the object. The following table lists valid UID types.

UASC_$UID	UASC file
RECORDS_$UID	Record-structured file
HDR_$UNDEF_$UID	Nonrecord-structured file
OBJECT_FILE_$UID	Object module file
PAD_$UID	Saved transcript pad
INPUT_PAD_$UID	Input pad
SIO_$UID	Serial line descriptor file
MBX_$UID	Mailbox object
MT_$UID	Magnetic tape descriptor file

Sparse flag	Boolean. When TRUE, file allocation may have "holes". See the *Programming With General System Calls* manual.
Flag3	A word. This is a field of bits containing Boolean and enumerated values. Each Boolean value occupies one bit in the flag. Enumerated types may occupy more than one bit depending on the number of possible values. The following table lists the bit number(s), the corresponding attribute, the data type of the attribute, and a brief explanation of possible values.

Bit Number	*Attribute*	*Data Type*	*Explanation*
Bits 16-27	Unused 7		
Bit 29	Close on exec	Boolean	TRUE causes stream to be closed upon an DOMAIN/IX exec call.
Bit 30	No delay mode	Boolean	When TRUE, any system call that reads data from a stream will act like STREAM_$GET_CONDITIONAL (returns if data not available).
Bit 31	Append mode	Boolean	When TRUE, any call to STREAM_$PUT_REC or STREAM_$PUT_CHR does a seek to EOF before writing any data.
Bit 28	Force locate mode	Boolean	Normally, if the force move mode bit (bit 6) is not set, streams may use either move mode or locate mode. If this bit is set, streams will only use locate mode; the caller need not supply a buffer. This option can only be set for file-type streams (UASC, REC, HDR_UNDEF, and CASE_HM).

Unused 8	A 4-byte integer.
Name	NAME_$PNAME_T. A character array of up to 256 elements. The name of the object. Specified on input with STREAM_$NAME_CONDITIONAL and STREAM_$NAME_UNCONDITIONAL inquiry types.

The array specified as the attribute parameter need not be large enough for every field, but just sufficient to span the required fields. For example, to redefine explicit move mode, only a six-element INTEGER*2 array is required (for a FORTRAN program), because the necessary flag is in FLAG1.

Accessing attribute record bit fields is discussed in detail in the *Programming With General System Calls* handbook.

OUTPUT PARAMETERS

error-mask

An integer bit-mask indicating any requested changes that were not made, in STREAM_$REDEF_MASK_T format. This is a 4-byte integer.

This procedure may complete with partial success if it can redefine some, but not all, of the requested attributes. If an attribute is not changed, STREAM_$REDEFINE sets the corresponding bit in the error mask and continues to redefine other attributes. In cases of partial success, the returned status code is nonzero. The program must check the error mask to find out where the error occurred.

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$REDEFINE changes one or more attributes of an object to which you have a stream open. Wherever bits are set in the input mask, STREAM_$REDEFINE tries to copy information from the corresponding fields of the attribute record to the object. Wherever bits are not set in the input mask, the corresponding attributes of the object do not change.

FORTRAN programs that use the stream manager to read files must call STREAM_$REDEFINE to request explicit move mode.

You can use STREAM_$REDEFINE only on streams with write access. However, you can use it to change read access to write access.

You cannot use STREAM_$REDEFINE to change the stream position (use STREAM_$SEEK instead), to change the object's length (use STREAM_$TRUNCATE instead), or to change a serial line's attributes (use SIO_$CONTROL).

STREAM_$REPLACE

Writes data to an object without changing the length of the current record.

FORMAT

```
STREAM_$REPLACE (stream-id, bufptr, buflen, seek-key, status)
```

INPUT PARAMETERS

stream-id
Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

bufptr
Pointer to the data to be written, in UNIV_PTR format. This is a 4-byte integer.

FORTRAN programs can use IADDR to obtain the buffer address for the bufptr parameter. Pascal programs can use ADDR. Alternately, programs in either language can use pointer variables.

buflen
Number of bytes of data to be written. This is a 4-byte integer.

OUTPUT PARAMETERS

seek-key
Unique key identifying the location of the output data in the object, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

The seek key allows random access to the output data by a subsequent STREAM_$SEEK call.

status
Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$REPLACE replaces a record in the object. Call STREAM_$PUT_REC to add records to an object.

For record-structured objects, this call terminates the current record. The length of the current record must be exactly the same as the length of the record it replaces. An error occurs if the record lengths are different. You can use STREAM_$PUT_REC and STREAM_$PUT_CHR to overwrite existing data in a file or pad with no record length checking.

For nonrecord-structured objects, this call writes the specified number of characters. No record-length errors can occur.

Like STREAM_$PUT_REC, STREAM_$REPLACE queues data for output to the object. It does not necessarily write the data on the COMMENT . COMMENT writes may be performed asynchronously.

STREAM_$SEEK

Moves the stream position.

FORMAT

```
STREAM_$SEEK (stream-id, seek-base, seek-type,
             {seek-key|signed-offset}, status)
```

INPUT PARAMETERS

stream-id

Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

seek-base

Type of data on which the seek is based, in STREAM_$PARM1_T format. Possible values are:

STREAM_$CHR
: Character-based seek.

STREAM_$EOF
: End-of-file based seek. (Any offset specified with STREAM_$EOF is ignored.)

STREAM_$KEY
: Keyed value-based seek.

STREAM_$REC
: Record-based seek.

seek-type

Value defining the relationship between the seek-base and the seek-key or signed-offset, in STREAM_$PARM2_T format. Possible values are:

STREAM_$RELATIVE
: Moves the stream position relative to the current stream marker. Relative positioning is only valid for character (STREAM_$CHR) and record (STREAM_$REC) based seeks. Specifying either STREAM_$EOF or STREAM_$KEY with STREAM_$RELATIVE results in an error status. A positive offset moves the stream position towards EOF and a negative offset moves the stream position towards the beginning of the object. Relative seeks start at 0; that is, an offset of 0 denotes the current position.

STREAM_$ABSOLUTE
: Seeks for an absolute position in the object. In absolute seeks, all four seek bases are valid. STREAM_$EOF and STREAM_$KEY can only be used in absolute seeks. Absolute character and record-based seeks start from the beginning of the object (past the header) if the offset is positive. If the offset is negative, the seek starts at EOF. Absolute seeks start at 1. An error occurs if you specify an offset of 0 for an absolute seek.

seek-key

Unique value identifying the data sought, in STREAM_$SK_T format. This is a three-element array of 4-byte integers.

signed-offset

Offset to be used in calculating the new stream position. This is a 4-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$SEEK moves the stream marker to a specific location, or to an offset from a known location. It does not move any data.

An error occurs if the object does not support random access.

An error occurs if the program attempts to move the stream position beyond the beginning or end of the file.

Character-based seeks in record-structured objects cannot move the stream position beyond the current record.

Record-based seeks apply only to objects with fixed-length records. For objects with variable-length records, save the seek keys returned when the object is written, then perform keyed seeks.

For UASC files, specifying STREAM_$REC simulates a fixed-length record seek by finding the length of the first record in the file and using that as the record length for all records. STREAM_$CHR-seeks work like they do in nonrecord-structured files.

The stream marker must be aligned on a record boundary when you specify STREAM_$REC. If alignment is incorrect, an error occurs. Similarly, if positioning is relative to EOF (that is, a negative offset), the current EOF must be on a record boundary.

The following examples illustrate the difference between absolute and relative seeks. The first example is an absolute seek for the 16th character in the object:

```
STREAM_$SEEK(stream_id, STREAM_$CHR, STREAM_$ABSOLUTE,
             16, status)
```

The second example positions the stream marker seven records closer to the beginning of the object than it was before the call.

```
STREAM_$SEEK(stream_id, STREAM_$REC, STREAM_$RELATIVE,
             -7, status)
```

STREAM_$TRUNCATE

Writes EOF at the current stream position.

FORMAT

```
STREAM_$TRUNCATE (stream-id, status)
```

INPUT PARAMETERS

stream-id

Number of a stream on which the object is open, in STREAM_$ID_T format. This is a 2-byte integer.

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This is a 4-byte integer.

USAGE

STREAM_$TRUNCATE decreases the value of the file length attribute to match the stream pointer's current position. (Writing data to a stream that lengthens the object implicitly increases this attribute's value.) This sets EOF to the stream pointer's position, effectively deleting any data in the object past the stream pointer. If the stream position is already at EOF, truncating the object has no effect.

You can only truncate disk files and pads that the Display Manager is not using. Trying to truncate any other type of object returns an error status code.

Truncating an object does not close the stream.

ERRORS

STATUS_$OK
  Successful completion.

STREAM_$ALREADY_EXISTS
  STREAM_$WRITE specified on STREAM_$CREATE.

STREAM_$BAD_CHAR_SEEK
  Attempted character seek before start of current (variable length) record.

STREAM_$BAD_COUNT_FIELD_IN_FILE
  Count field for current record is bad.

STREAM_$BAD_FILE_HDR
  File header is no good (CRC error).

STREAM_$BAD_LOCATION
  Bad location parameter in create call.

STREAM_$BAD_OPEN_xp
  OPEN_XP must reference a stream that is already open on this node.

STREAM_$BAD_POS_ON_REC_SEEK
  Relative record seek is not legal unless the reference point is on record boundary.

STREAM_$BAD_RELATED_PAD
  PAD_$CREATE attempted with an invalid or unopened related pad.

STREAM_$BAD_SHARED_CURSOR_REFCNT
  Reference count on a shared file cursor went below zero.

STREAM_$BOF_ERR
  Attempted seek beyond BOF; e.g., offset=0.

STREAM_$CANT_DELETE_OLD_NAME
  WARNING : New name added but old cannot be deleted.

STREAM_$CANT_SWITCH
  Too many mapped objects to perform switch.

STREAM_$CLOSE_ANOMALY
  WARNING : Close successful but name of (temporary) object on this stream no longer references the same object.

STREAM_$CONCURRENCY_VIOLATION
  Requested access violates concurrency constraints.

STREAM_$DEVICE_MUST_BE_LOCAL
  Cannot open stream to remote device.

STREAM_$DIR_NOT_FOUND
  Could not find directory in pathname on create.

STREAM_$END_OF_FILE
  End of file.

STREAM_$EOF_PAD_PUT_ERR
  PUT_REC legal only at EOF on pads; EOF has moved.

STREAM_$FILE_TROUBLE_WARNING
WARNING: (SALVAGER) File trouble bit set in VTOCE.

STREAM_$FROM_STRID_NOT_OPEN
From stream is not open on switch request.

STREAM_$ID_OOR
Stream ID out-of-range (invalid).

STREAM_$ILL_FORCED_LOCATE
Forced locate is only legal for disk files.

STREAM_$ILLEGAL_NAME_REDEFINE
Attempted name change requires copying file.

STREAM_$ILLEGAL_OBJ_TYPE
Cannot open a stream for this type of object.

STREAM_$ILLEGAL_OPERATION
This operation is illegal on named stream.

STREAM_$ILLEGAL_PAD_CLOSE
Illegal to close transcript pad before related input pad.

STREAM_$ILLEGAL_PAD_CREATE_TYPE
PAD_CREATE illegal with this type of object.

STREAM_$ILLEGAL_PARAM_COMB
Illegal parameter combination for this operation.

STREAM_$ILLEGAL_W_VAR_LGTH_RECS
Operation illegal with variable length records.

STREAM_$INQUIRE_TYPE_ERR
Inquire (by name) about object that cannot be opened on a stream because of its type.

STREAM_$INQUIRE_WARNING
WARNING : Inquire-by-name is returning data only on first of multiple streams on which object is currently open.

STREAM_$INSUFF_MEMORY
Not enough virtual memory.

STREAM_$INSUFFICIENT_RIGHTS
Insufficient rights for requested access to object.

STREAM_$INTERNAL_FATAL_ERR
Internal fatal error on table reverification.

STREAM_$INTERNAL_MM_ERR
Internal fatal error in stream memory management (windowing).

STREAM_$INVALID_DATA
Bad data in call to VT_$PUT.

STREAM_$NAME_CONFUSION
Object already open under another name on another stream.

STREAM_$NAME_NOT_FOUND
Name not found.

STREAM_$NAME_REQD
STREAM_$OPEN without a name is illegal.

STREAM_$NEED_MOVE_MODE
Forced locate is set and could not do it.

STREAM_$NEVER_CLOSED
System (or process) crash prevented complete close.

STREAM_$NO_AVAIL_TARGET
No available target stream to switch to.

STREAM_$NO_MORE_STREAMS
No more streams.

STREAM_$NO_RIGHTS
No rights to access object.

STREAM_$NO_SUCH_VERSION
Specified DSEE version does not exist.

STREAM_$NO_TABLE_SPACE
Table space error; cover stream table exhausted.

STREAM_$NOT_OPEN
Operation attempted on unopened stream.

STREAM_$NOT_THRU_LINK
Cannot create file though link.

STREAM_$OBJ_DELETED
File has been deleted.

STREAM_$OBJECT_NOT_FOUND
Object associated with this name not found (may not exist).

STREAM_$OBJECT_READ_ONLY
Cannot open this object for writing.

STREAM_$OUT_OF_SHARED_CURSORS
Per-mode shared file cursor pool is exhausted.

STREAM_$PART_REC_WARN
WARNING : Partial record at the end of a file with fixed length records.

STREAM_$PERM_FILE_NEEDS_NAME
Only temporary files may be unnamed.

STREAM_$PUT_BAD_REC_LEN
Attempted put of wrong length record.

STREAM_$READ_ONLY_ERR
Attempted write to read-only stream.

STREAM_$REDEFINE_PAD_ERR
Cannot redefine this attribute of a pad.

STREAM_$REPLACE_LGTH_ERR
Attempted record length change on replace request.

STREAM_$RESOURCE_LOCK_ERR
Unable to lock resources required to process request.

STREAM_$SIO_NOT_LOCAL
SIO object not in /DEV.

STREAM_$SOMETHING_FAILED
Partial or complete failure of inquire or redefine (ERR_MASK is nonempty).

STREAM_$STREAM_NOT_FOUND
No stream found in conditional inquire.

STREAM_$XP_BUF_TOO_SMALL
Buffer supplied to GET_XP too small.

TIME

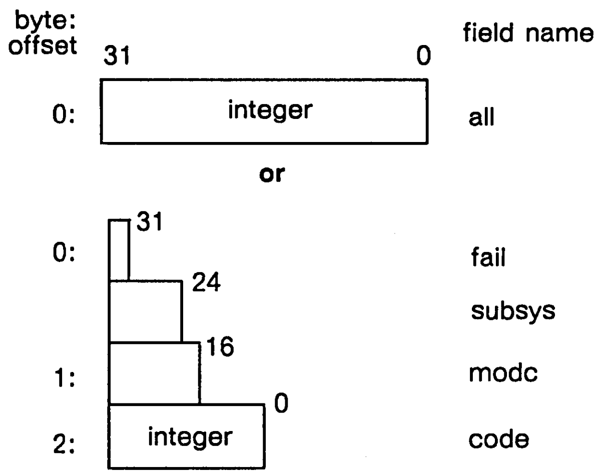
This section describes the data types, the call syntax, and the error codes for the TIME programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

CONSTANTS

TIME_$ABSOLUTE	Specifies absolute time.
TIME_$CLOCKH_KEY	Eventcount key value.
TIME_$RELATIVE	Specifys relative time.

DATA TYPES

EC2_$PTR_T	A 4-byte integer. Address of an eventcount.
STATUS_$T	A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

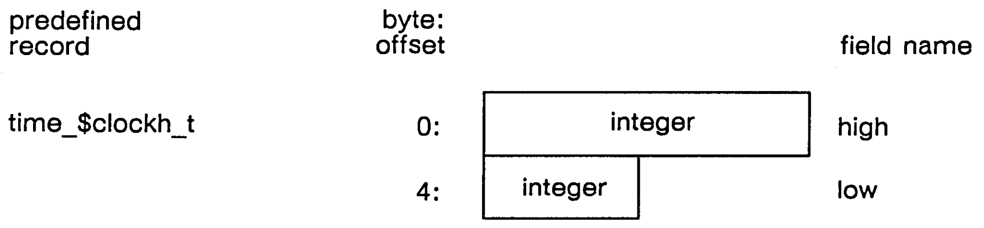
subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

TIME_$CLOCK_T

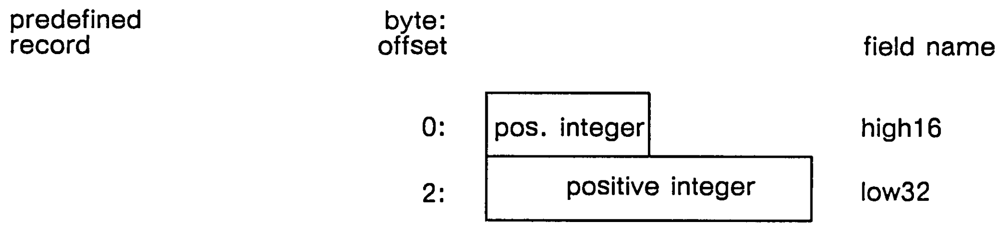
Internal representation of time. The diagram below illustrates the TIME_$CLOCK_T data type:



Field Description:

high
High 32 bits of the clock.

low
Low 16 bits of the clock.



Field Description:

high16
High 16 bits of the clock.

low32
Low 32 bits of the clock.

TIME_$KEY_T

A 2-byte integer. An event count key. One of the following pre-defined values:

TIME_$CLOCKH_KEY
Only permissible value.

TIME_$REL_ABS_T

A 2-byte integer. An indicator of type of time. One of the following pre-defined values:

TIME_$RELATIVE
Relative time.

TIME_$ABSOLUTE
Absolute time.

TIME_$CLOCK

Returns the current UTC time.

FORMAT

```
TIME_$CLOCK (clock)
```

OUTPUT PARAMETERS

clock

The current Coordinated Universal Time, in TIME_$CLOCK_T format. This data type is 6 bytes long. See the CAL Data Types section for more information.

USAGE

TIME_$CLOCK returns the current time of day, in UTC format. It is represented as the number of 4-microsecond periods that have elapsed since January 1, 1980 at 00:00.

To get the local time, use CAL_$GET_LOCAL_TIME instead of this procedure. To compute the local time from the value returned by TIME_$CLOCK, use CAL_$APPLY_LOCAL_OFFSET.

TIME_$GET_EC

Gets the address of the time eventcount, which is advanced about every 0.25 second.

FORMAT

```
TIME_$GET_EC (time-key, eventcount-pointer, status)
```

INPUT PARAMETERS

time-key

The key specifying which time eventcount the system should return, in TIME_$KEY_T format. This is a 2-byte integer.

Currently the only allowable value is TIME_$CLOCKH_KEY.

OUTPUT PARAMETERS

eventcount-pointer

The eventcount address to be obtained, in EC2_$PTR_T format. This is a 4-byte integer.

EC2_$PTR_T is a pointer to an EC2_$EVENTCOUNT_T array. See the EC2 Data Types section for more information.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the TIME_$ Data Types section for more information.

USAGE

EC2_$PTR_T is a pointer to an EC2_$EVENTCOUNT_T array. See the EC2 Data Types section for a description of eventcount data structures.

TIME_$GET_EC outputs an eventcount that gets advanced about every 1/4 second. Thus, it can be passed to EC2_$WAIT to wait for a specific interval of time to elapse.

The interval between successive advances of the eventcount is nominally 262,144 microseconds (about 0.25 second). The exact interval changes slightly with system load.

For a ten-second wait, you might use:

```
TIME_$GET_EC (....gets time_eventcount ....)
EC2_$READ    (....gets current_eventcount_value ....)
EC2_$WAIT    (....current_eventcount + 40, time_eventcount....)
```

See Eventcounts Chapter of the *Programming With General System Calls* manual for more information.

TIME_$WAIT

Suspends the calling process for a specified time.

FORMAT

```
TIME_$WAIT (rel-abs, clock, status)
```

INPUT PARAMETERS

rel-abs

Type of clock value supplied, in TIME_$REL_ABS_T format. This is a 2-byte integer. Specify only one of the following predefined values:

TIME_$RELATIVE	Clock specifies the number of 4-microsecond periods to wait before resuming process execution.
TIME_$ABSOLUTE	Clock contains the UTC system time for which to wait before resuming process execution.

clock

The relative or absolute time for which to wait before resuming process execution, in TIME_$CLOCK_T format. This data type is 6 bytes long. See the CAL Data Types section for more information.

Note that if you specify TIME_$ABSOLUTE in the rel_abs parameter, TIME_$WAIT expects a UTC time. (You can remove a local time offset with the CAL_$REMOVE_LOCAL_OFFSET call.)

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the CAL Data Types section for more information.

USAGE

TIME_$WAIT suspends the calling process until a relative time elapses or an absolute time occurs.

A nonzero status is returned if the operating system has insufficient internal table space to process the request.

ERRORS

STATUS_$OK	Successful completion.
TIME_$BAD_INT	Bad timer interrupt.
TIME_$BAD_KEY	Bad key to TIME_$GET_EC.
TIME_$NO_Q_ENTRY	Error from TIME_$ADVANCE.
TIME_$NOT_FOUND	Entry to be canceled not found.
TIME_$WAIT_QUIT	Wait interrupted by quit fault.

TONE

This section describes the call syntax for the TONE programming calls. The TONE calls do not use special data types or produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of call syntax format.

TONE_$TIME

Makes a tone. The tone remains on for the time indicated in the call.

FORMAT

```
TONE_$TIME (time)
```

INPUT PARAMETERS

time
Length of the tone, in TIME_$CLOCK_T format. This data type in a 48-bit integer value.

USAGE

Only DOMAIN nodes shipped after April 19, 1982 contain a working speaker.

TPAD

This section describes the data types and the call syntax for the TPAD programming calls. The TPAD calls do not produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

TPAD_$MODE_T	A 2-byte integer. Cursor mode operations for the touchpad and mouse. They establish how movements of the finger affect the cursor position on the display. Note the only meaningful mode for a mouse is TPAD_$RELATIVE. One of the following pre-defined values:

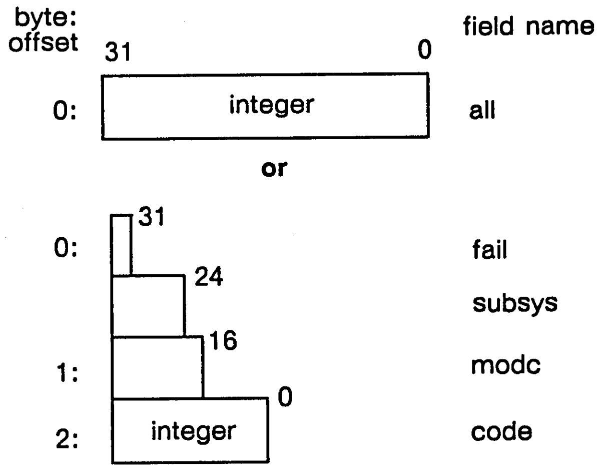
TPAD_$ABSOLUTE
Touchpad corresponds directly to the display. When a finger is placed on the touchpad, the cursor jumps to the corresponding position on the screen.

TPAD_$RELATIVE
Cursor movement is relative to the current position. It moves only when a finger moves across the pad, it does not move when a finger is placed on touchpad.

TPAD_$REL_ABS
Cursor moves when finger is placed on the touchpad and when a finger moves across the pad. It does not move if a finger is lifted and replaced quickly.

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

TPAD_$INQUIRE

Returns information about the current touchpad mode.

FORMAT

```
TPAD_$INQUIRE (mode, x-scale, y-scale, hysteresis, origin)
```

INPUT PARAMETERS

None.

OUTPUT PARAMETERS

mode

Cursor mode, in TPAD_$MODE_T format. This is a 2-byte integer. Specify one of the following predefined values:

TPAD_$ABSOLUTE

TPAD_$RELATIVE

TPAD_$REL_ABS

x-scale

Scale factor in the x dimension. This is a 2-byte integer.

y-scale

Scale factor in the y dimension. This is a 2-byte integer.

hysteresis

Hysteresis factor, in pixels. This is a 2-byte integer.

This hysteresis factor prevents the touchpad manager from responding to any minor movements you make unintentionally. This value defines a "box" around your finger. The touchpad manager does not move the cursor if your finger stays within this box.

If your finger moves beyond the box, the touchpad manager subtracts the hysteresis value from the distance moved, and moves the cursor the remaining distance. The default factor is 5.

origin

The point of origin for x and y in SMD_$POS_T format. This data type is 4 bytes long.

USAGE

Use this call to save the touchpad mode for later restoration, or to change one aspect of the mode without changing any other aspects. For example, you can use the output from this call as the input to the TPAD_$SET_MODE call.

TPAD_$INQ_DTYPE

Returns the name of the last type of locating device used.

FORMAT

```
device = TPAD_$INQ_DTYPE
```

INPUT PARAMETERS

None.

OUTPUT PARAMETERS

None.

RETURN VALUE

device

Value indicating the last locating device used to provoide input, in TPAD_$DEV_TYPE_T format. This data type is two bytes long. Specify one of the following predefined values:

```
TPAD_$UNKNOWN

TPAD_$HAVE_TOUCHPAD

TPAD_$HAVE_MOUSE

TPAD_$HAVE_BITPAD
```

USAGE

If no locator input has been detected since the node was last booted, this call returns TPAD_$UNKNOWN.

TPAD_$RE_RANGE

Re-establishes the touchpad raw data range.

FORMAT

```
TPAD_$RE_RANGE ()
```

INPUT PARAMETERS

None.

OUTPUT PARAMETERS

None.

USAGE

This call re-establishes the touchpad raw data range over the next 1000 data points. This is also done for you at system boot. See the section on Touchpad Modes in *Programming with General System Calls* for a description of the touch pad raw data range.

TPAD_$SET_CURSOR

Re-establishes the touchpad origin in relative mode. The call to TPAD_$SET_CURSOR can occur at any time and affects subsequent touchpad inputs.

FORMAT

```
TPAD_$SET_CURSOR (origin)
```

INPUT PARAMETERS

origin

A screen position that will be the origin for subsequent data points from the touchpad in relative mode or in absolute/relative mode, in SMD_$POS_T format. This data type is 4 bytes long.

OUTPUT PARAMETERS

None.

USAGE

The system *remembers* the last cursor position delivered by a locator device. When a new data point comes from the mouse, or from the touchpad or bitpad in relative mode, a **displacement** is computed and applied to the last locator position. The TPAD_$SET_CURSOR call makes the system *forget* the last locator position, and use the value passed in the call instead. The next locator data will then start from this new position instead of its former position.

You will rarely need to make this call, as GPR and the display manager make the call at appropriate times.

The origin is automatically re-established when you take your finger from the touchpad for more than one-eighth of a second. One effect of this is that the cursor typically doesn't move the next time you touch the pad in relative mode, unless you explicitly call TPAD_$SET_CURSOR before that next touch.

This call has meaning for relative and absolute/relative mode only. In absolute/relative mode, when you first touch the pad, the pad inputs coordinates in the absolute mode. To have effect, the call to TPAD_$SET_CURSOR must occur after this first touch, but during the relative part of this use of the touchpad (that is, before you lift your finger for more than one-half second.)

TPAD_$SET_MODE

Sets the touch pad mode.

FORMAT

```
TPAD_$SET_MODE (mode, x-scale, y-scale, hysteresis, origin)
```

INPUT PARAMETERS

mode

Cursor mode, in TPAD_$MODE_T format. This is a 2-byte integer. Specify one of the following predefined values:

```
TPAD_$ABSOLUTE

TPAD_$RELATIVE

TPAD_$REL_ABS
```

x-scale

Scale factor in the x dimension. This is a 2-byte integer.

y-scale

Scale factor in the y dimension. This is a 2-byte integer.

hysteresis

Hysteresis factor, in pixels. This is a 2-byte integer.

The hysteresis factor prevents the touchpad manager from responding to any minor movements you make unintentionally. This value defines a "box" around your finger. The touchpad manager does not move the cursor if your finger stays within this box.

If your finger moves beyond the box, the touchpad manager subtracts the hysteresis value from the distance moved, and moves the cursor the remaining distance. The default factor is 5.

origin

The point of origin for x and y in SMD_$POS_T format. This data type is 4 bytes long.

OUTPUT PARAMETERS

None.

USAGE

Use this call to set to mode, scale factors, and hysteresis factors of locator devices. You can also change the origin for relative or absolute/relative mode. This call applies to the touchpad, mouse, and bit pad locator devices. Note that the mouse uses only the scale and hysteresis factors and ignores the other mode settings, since it is an inherently relative device.

VEC

This section describes the call syntax for the VEC programming calls. The VEC calls do not use special data types or produce unique error messages. Refer to the Introduction at the beginning of this manual for a description of call syntax format.

The majority of the calls in this section have four versions: single-precision floating-point, double-precision floating-point, 16-bit integer (INTEGER*2), and 32-bit integer (INTEGER*4). The names of all single-precision vector routines are in the form VEC_$Iname. Double-precision routines are named VEC_$Dname. 16-bit integer routines are named VEC_$Iname16. 32-bit integer routines are named VEC_$Iname. For example, VEC_$DOT and VEC_$DDOT are single- and double-precision versions of DOT product routines. VEC_$IDOT and VEC_$IDOT16 are the 32-bit and 16-bit versions, respectively.

Each type of routine generally requires parameters of the same type. For the double-precision routines, all floating-point parameters are double-precision; for the single-precision routines, all floating-point parameters must be single precision; for the integer procedures and functions, the parameters and returned values are integers, etc.

Routine names that end in I denote "incremental" routines, which step through vectors at increments other than 1.

When calling any of the vector routines, make sure that the indices you pass are valid. For best performance, these routines do **not** check index values for validity; hence, passing a value of zero can cause a variety of errors.

NOTE: All matrices are assumed to be stored in FORTRAN (column-major) order. Because Pascal and C store matrix elements in row-major order, you may need to transpose or otherwise rearrange elements when calling vector routines from C or Pascal programs.

VEC_$ADD_CONSTANT

Adds a constant to a vector.

FORMAT

```
VEC_$ADD_CONSTANT (start_vec, length, constant, result_vec)
VEC_$DADD_CONSTANT (start_vec, length, constant, result_vec)
VEC_$IADD_CONSTANT (start_vec, length, constant, result_vec)
VEC_$IADD_CONSTANT16 (start_vec, length, constant, result_vec)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to which the value will be added.

length
Number of elements to add. This is a 4-byte integer.

constant
Value to be added to each element of start_vec.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

USAGE

These routines add a constant to a vector, returning the result in a second vector. The routines perform the following operation:

```
      DO 10 I + 1,LENGTH
        RESULT_VEC(I) = CONSTANT + START_VEC(I)
   10 CONTINUE
```

VEC_$ADD_CONSTANT_I

Adds a constant to a vector, stepping through the vector by increments.

FORMAT

```
VEC_$ADD_CONSTANT_I (start_vec, inc1, length, constant, result_vec, inc2)
VEC_$DADD_CONSTANT_I (start_vec, inc1, length, constant, result_vec, inc2)
VEC_$IADD_CONSTANT_I (start_vec, inc1, length, constant, result_vec, inc2)
VEC_$IADD_CONSTANT16_I(start_vec, inc1, length, constant, result_vec, inc2)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to which the value will be added.

inc1
Increment for the index of start_vec. This is a 4-byte integer.

length
Number of elements to add. This is a 4-byte integer.

constant
Value to be added to each element of start_vec.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

INPUT PARAMETERS

inc2
Increment for the index of result_vec. This is a 4-byte integer.

USAGE

These routines add a constant to a vector, stepping through their elements by user-specified increments. The routines perform the following operation:

```
      J=1
      K=1
      DO 10 I = 1,LENGTH
        RESULT_VEC(K) = CONSTANT+START_VEC(J)
        K = K+INC2
        J = J+INC1
   10 CONTINUE
```

VEC_$ADD_VECTOR

Adds two vectors.

FORMAT

```
VEC_$ADD_VECTOR (start_vec, add_vec, length, result_vec)
VEC_$DADD_VECTOR (start_vec, add_vec, length, result_vec)
VEC_$IADD_VECTOR (start_vec, add_vec, length, result_vec)
VEC_$IADD_VECTOR16 (start_vec, add_vec, length, result_vec)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to which the value will be added.

add_vec
Floating-point or integer vector to be added.

length
Number of elements to add. This is a 4-byte integer.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

USAGE

These routines add two vectors, returning the sum in a third vector. The routines perform the following operation:

```
      DO 10 I = 1,LENGTH
        RESULT_VEC(I) = START_VEC(I) + ADD_VEC(I)
   10 CONTINUE
```

VEC_$ADD_VECTOR_I

Adds two vectors, stepping through them by increments.

FORMAT

```
VEC_$ADD_VECTOR_I(start_vec, inc1, add_vec, inc2, length, result_vec, inc3)
VEC_$DADD_VECTOR_I(start_vec, inc1, add_vec, inc2, length, result_vec,
                   inc3)
VEC_$IADD_VECTOR_I(start_vec, inc1, add_vec, inc2, length, result_vec,
                   inc3)
VEC_$IADD_VECTOR16_I(start_vec, inc1, add_vec, inc2, length, result_vec,
                     inc3)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to which the value will be added.

inc1
Increment for the index of start_vec. This is a 4-byte integer.

add_vec
Floating-point or integer vector to be added.

inc2
Increment for the index of add_vec. This is a 4-byte integer.

length
Length of the vector. This is a 4-byte integer.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

INPUT PARAMETERS

inc3
Increment for the index of result_vec. This is a 4-byte integer.

USAGE

These routines add two vectors, stepping through their elements by user-specified increments. The routines perform the following operation:

```
      J=1
      K=1
      DO 10 I = 1,LENGTH
        VEC3(J) = VEC1(K) + VEC2(L)
        J = J + INC3
        K = K + INC1
        L = L + INC2
   10 CONTINUE
```

VEC_$COPY

Copies elements from one vector to another.

FORMAT

```
VEC_$COPY (start_vec, result_vec, length)
VEC_$DCOPY (start_vec, result_vec, length)
VEC_$ICOPY (start_vec, result_vec, length)
VEC_$ICOPY16 (start_vec, result_vec, length)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector from which elements will be copied.

length
Number of elements to copy. This is a 4-byte integer.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector to which elements will be copied.

USAGE

These routines copy elements from one vector to another. The routines perform the following operation:

```
   DO 10 I = 1,LENGTH
     RESULT_VEC(I) = START_VEC(I)
10 CONTINUE
```

VEC_$COPY_I

Copies elements from one vector to another, stepping through the vectors by increments.

FORMAT

```
VEC_$COPY_I (vec1, inc1, vec2, inc2, length)
VEC_$DCOPY_I (vec1, inc1, vec2, inc2, length)
VEC_$ICOPY_I (vec1, inc1, vec2, inc2, length)
VEC_$ICOPY16_I (vec1, inc1, vec2, inc2, length)
```

INPUT PARAMETERS

vec1
Floating-point or integer vector from which elements will be copied.

inc1
Increment for the index of vec1. This is a 4-byte integer.

inc2
Increment for the index of vec2. This is a 4-byte integer.

length
Number of elements to copy. This is a 4-byte integer.

OUTPUT PARAMETERS

vec2
Floating-point or integer vector to which elements will be copied.

USAGE

These routines copy a vector but use an increment to step through the vector. VEC_$COPY_I moves 32 bits regardless of data type and VEC_$DCOPY_I moves 64 bits. The routines perform the following operation:

```
      J = 1
      K = 1
      DO 10 I = 1,LENGTH
         VEC2(J) = VEC1(K)
         J = J+INC2
         K = K+INC1
   10 CONTINUE
```

VEC_$DOT

Calculates the dot product of two vectors.

FORMAT

```
result = VEC_$DOT (vec1, vec2, length)
result = VEC_$DDOT (vec1, vec2, length)
result = VEC_$IDOT (vec1, vec2, length)
result = VEC_$IDOT16 (vec1, vec2, length)
```

RETURN VALUE

result
Floating-point or integer dot product.

INPUT PARAMETERS

vec1, vec2
Floating-point or integer vectors.

length
Number of elements to add. This is a 4-byte integer.

USAGE

These routines calculate the dot product of two vectors. The routines perform the following operation:

```
   DO 10 I = 1,LENGTH
    TEMP = TEMP + VEC1(I) * VEC2(I)
10 CONTINUE
   VEC_$DOT = TEMP
```

VEC_$DOT_I

Calculates the dot product of two vectors, stepping through the vectors by increments.

FORMAT

```
result = VEC_$DOT_I (vec1, inc1, vec2, inc2, length)
result = VEC_$DDOT_I (vec1, inc1, vec2, inc2, length)
result = VEC_$IDOT_I (vec1, inc1, vec2, inc2, length)
result = VEC_$IDOT16_I (vec1, inc1, vec2, inc2, length)
```

RETURN VALUE

result
Floating-point or integer dot product.

INPUT PARAMETERS

vec1
Floating-point or integer vector.

inc1
Increment for the index of vec1. This is a 4-byte integer.

vec2
Floating-point or integer vector.

inc2
Increment for the index of vec1. This is a 4-byte integer.

length
Number of elements to add. This is a 4-byte integer.

USAGE

These routines calculate the dot product of two vectors, stepping through the vectors with a user-supplied increment. The routines perform the following operation:

```
      J = 1
      K = 1
      DO 10 I = 1,LENGTH
       TEMP = TEMP + VEC1(J) * VEC2(K)
       J = J + INC1
       K = K + INC2
   10 CONTINUE
      VEC_$DOT = TEMP
```

VEC_$DP_SP

Copies a double-precision vector to a single precision vector.

FORMAT

```
VEC_$DP_SP (dp_vec, sp_vec, length)
```

INPUT PARAMETERS

dp_vec
Floating-point double-precision vector.

OUTPUT PARAMETERS

sp_vec
Floating-point single-precision resultant vector.

INPUT PARAMETERS

length
Number of elements to copy. This is a 4-byte integer.

USAGE

VEC_$DP_SP copies a double-precision vector to a single-precision resultant vector. The routine performs the following operation:

```
      DO 10 I=1, LENGTH
        SP_VEC(I) = SNGL(DP_VEC(I))
   10 CONTINUE
```

VEC_$DP_SP_I

Copies a double-precision vector to a single-precision vector, stepping through the vectors incrementally.

FORMAT

```
VEC_$DP_SP_I (dp_vec, inc1, sp_vec, inc2, length)
```

INPUT PARAMETERS

dp_vec
Double-precision floating-point vector.

inc1
Increment for the index of dp_vec. This is a 4-byte integer.

OUTPUT PARAMETERS

sp_vec
Single-precision floating-point resultant vector.

INPUT PARAMETERS

inc2
Increment for the index of sp_vec. This is a 4-byte integer.

length
Number of elements to copy. This is a 4-byte integer.

USAGE

VEC_$DP_SP_I copies a double precision vector to a single precision resultant vector, stepping through the vectors by user-supplied increments. The routine performs the following operation:

```
      J=1
      K=1
      DO 10 I=1, LENGTH
         SP_VEC(K) = SNGL (DP_VEC(J))
         J = J+INC1
         K = K+INC2
   10 CONTINUE
```

VEC_$INIT

Initializes a vector with a constant.

FORMAT

```
VEC_$INIT (vector, length, constant)
VEC_$DINIT (vector, length, constant)
VEC_$IINIT (vector, length, constant)
VEC_$IINIT16 (vector, length, constant)
```

INPUT/OUTPUT PARAMETERS

vector

Upon input	Floating-point or integer vector to be initialized.
Upon output	Floating-point or integer vector which has been initialized to a constant.

INPUT PARAMETERS

length
Number of elements to initialize. This is a 4-byte integer.

constant
Floating-point or integer constant to be assigned to elements of vector.

USAGE

These routines perform the following operation:

```
      DO 10 I = 1, LENGTH
         VECTOR (I) = CONSTANT
   10 CONTINUE
```

VEC_$MAT_MULT

Multiplies two 4 x 4 matrices, returning the result in a 4 x 4 matrix.

FORMAT

```
VEC_$MAT_MULT (matrix1, matrix2, out_matrix)
VEC_$DMAT_MULT (matrix1, matrix2, out_matrix)
VEC_$IMAT_MULT (matrix1, matrix2, out_matrix)
VEC_$IMAT_MULT16 (matrix1, matrix2, out_matrix)
```

INPUT PARAMETERS

matrix1
A 4 x 4 floating-point or integer matrix to be multiplied.

matrix2
A 4 x 4 floating-point or integer matrix to be multiplied.

OUTPUT PARAMETERS

out_matrix
Floating-point or integer matrix containing the product.

USAGE

These routines multiply two 4 x 4 matrices, returning the result as a third 4 x 4 matrix. They are intended for use in graphics applications, and perform the following operation:

```
      DO 10 J = 1,4
      DO 10 I = 1,4
        CALL VEC_$POST_MULT (MATRIX1,MATRIX2(J,I),OUT_MATRIX(J,I))
   10 CONTINUE
```

VEC_$MAT_MULTN

Multiplies two variably-dimensioned matrices, returning the result in a third matrix.

FORMAT

```
VEC_$MAT_MULTN (matrix1, matrix2, m, n, s, out_matrix)
VEC_$DMAT_MULTN (matrix1, matrix2, m, n, s, out_matrix)
VEC_$IMAT_MULTN (matrix1, matrix2, m, n, s, out_matrix)
VEC_$IMAT_MULTN16 (matrix1, matrix2, m, n, s, out_matrix)
```

INPUT PARAMETERS

matrix1
First floating-point or integer matrix of dimensions m x n to be multiplied.

matrix2
Second floating-point or integer matrix of dimensions n x s to be multiplied.

m, n, s
The various matrix dimensions. These are 4-byte integers.

OUTPUT PARAMETERS

out_matrix
Floating-point or integer matrix of dimensions m x s containing the product.

USAGE

These routines multiply two matrices with dimensions specified by you, returning the result as a third matrix. Note that the matrices are assumed to be stored in FORTRAN (column-major) order. These routines perform the following operation:

```
   DO 10 I = 1,M
   DO 10 J = 1,S
     OUT_MATRIX(I,J) = 0.0
     DO 10 K = 1,N
     OUT_MATRIX(I,J) = OUT_MATRIX(I,J) + MATRIX1(I,K) * MATRIX2(K,J)
10 CONTINUE
```

VEC_$MAX

Finds the element with the greatest maximum absolute value in a vector and returns its value and location.

FORMAT

```
VEC_$MAX (start_vec, length, result, result_loc)
VEC_$DMAX (start_vec, length, result, result_loc)
VEC_$IMAX (start_vec, length, result, result_loc)
VEC_$IMAX16 (start_vec, length, result, result_loc)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to be searched.

length
Number of elements to examine. This is a 4-byte integer.

OUTPUT PARAMETERS

result
Floating-point or integer maximum absolute value of all the elements searched.

result_loc
Location of value within the vector. This is a 4-byte integer.

USAGE

These routines search through a vector and return the maximum absolute value found and its location within the vector. The routines perform the following:

```
      RESULT = ABS (START_VEC(1))
      RESULT_LOC = 1
      DO 10 I=2, LENGTH
         IF (ABS(START_VEC(I)).GT.RESULT) THEN
            RESULT_LOC = I
            RESULT = ABS(START_VEC(I))
         ENDIF
   10 CONTINUE
```

VEC_$MAX_I

Searches a vector for the maximum absolute value, stepping through the vector by increments greater than 1.

FORMAT

```
VEC_$MAX_I (start_vec, inc, length, result, result_loc)
VEC_$DMAX_I (start_vec, inc, length, result, result_loc)
VEC_$IMAX_I (start_vec, inc, length, result, result_loc)
VEC_$IMAX16_I (start_vec, inc, length, result, result_loc)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector to be searched.

inc
Increment used to step through the start_vec. This is a 4-byte integer.

length
Number of elements to examine. This is a 4-byte integer.

OUTPUT PARAMETERS

result
Floating-point or integer maximum absolute value of all the elements searched.

result_loc
Location of value within given vector. This is a 4-byte integer.

USAGE

These routines search through a vector by a positive increment and return the greatest absolute value found and its location within the vector. The routines perform the following:

```
   RESULT = ABS(START_VEC(1))
   RESULT_LOC = 1
   J = 1+INC1
   DO 10 I=2,LENGTH
        IF (ABS(START_VEC(J)).G.T.RESULT) THEN
             RESULT_LOC = I
             RESULT = ABS(START_VEC(J))
        ENDIF
        J = J+INC1
10 CONTINUE
```

VEC_$MULT_ADD

Multiplies a vector by a constant and adds the result to a second vector.

FORMAT

```
VEC_$MULT_ADD (add_vec, mult_vec, length, constant, result_vec)
VEC_$DMULT_ADD (add_vec, mult_vec, length, constant, result_vec)
VEC_$IMULT_ADD (add_vec, mult_vec, length, constant, result_vec)
VEC_$IMULT_ADD16 (add_vec, mult_vec, length, constant, result_vec)
```

INPUT PARAMETERS

add_vec
Floating-point or integer vector to be added.

mult_vec
Floating-point or integer vector to be multiplied by the constant.

length
Number of elements to add. This is a 4-byte integer.

constant
Floating-point or integer constant to be multiplied by mult_vec.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

USAGE

These routines multiply one vector (input as mult_vec) by a constant, and add the result to a second vector (input as add_vec). The result is returned in a third vector. The routines perform the following operation:

```
      DO 10 I = 1, LENGTH
         RESULT_VEC(I) = ADD_VEC(I) + CONSTANT*MULT_VEC(I)
   10 CONTINUE
```

VEC_$MULT_ADD_I

Multiplies a vector by a constant and adds the result to a second vector, stepping through both vectors and the result by increments.

FORMAT

```
VEC_$MULT_ADD_I (add_vec, inc1, mult_vec, inc2,
                 length, constant, result_vec, inc3)
VEC_$DMULT_ADD_I (add_vec, inc1, mult_vec, inc2,
                  length, constant, result_vec, inc3)
VEC_$IMULT_ADD_I (add_vec, inc1, mult_vec, inc2,
                  length, constant, result_vec, inc3)
VEC_$IMULT_ADD16_I (add_vec, inc1, mult_vec, inc2,
                    length, constant, result_vec, inc3)
```

INPUT PARAMETERS

add_vec
Floating-point or integer vector to be added.

inc1
Increment for the index of add_vec. This is a 4-byte integer.

mult_vec
Floating-point or integer vector to be multiplied by the constant.

inc2
Increment for the index of mult_vec. This is a 4-byte integer.

length
Number of elements on which to operate. This is a 4-byte integer.

constant
Floating-point or integer constant to be multiplied by mult_vec2.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the sum.

INPUT PARAMETERS

inc3
Increment for the index of result_vec. This is a 4-byte integer.

USAGE

These routines multiply one vector by a constant and add the result to a second vector. The result is returned in a third vector. The indices to all three vectors are incremented by user-specified values. The routines perform the following operation:

```
         J = 1
         K = 1
         L = 1
         DO 10 I = 1,LENGTH
           RESULT(J) = ADD_VEC(K) + CONSTANT*MULT_VEC(L)
           J = J + INC3
           K = K + INC1
           L = L + INC2
      10 CONTINUE
```

VEC_$MULT_CONSTANT

Multiplies a vector by a scalar constant and returns the result in a second vector.

FORMAT

```
VEC_$MULT_CONSTANT (mult_vec, length, constant, result_vec)
VEC_$DMULT_CONSTANT (mult_vec, length, constant, result_vec)
VEC_$IMULT_CONSTANT (mult_vec, length, constant, result_vec)
VEC_$IMULT_CONSTANT16 (mult_vec, length, constant, result_vec)
```

INPUT PARAMETERS

mult_vec
Floating-point or integer vector to be multiplied.

length
Number of elements to multiply. This is a 4-byte integer.

constant
Floating-point or integer constant to multiply by mult_vec.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the product.

USAGE

These routines multiply one vector by a scalar constant, returning the result in a second vector. The routines perform the following operation:

```
      DO 10 I = 1,LENGTH
        RESULT_VEC(I) = CONSTANT * MULT_VEC(I)
   10 CONTINUE
```

VEC_$MULT_CONSTANT_I

Multiplies a vector by a scalar constant, returns the result in a second vector, and steps through the vectors by increments.

FORMAT

```
VEC_$MULT_CONSTANT_I (mult_vec, inc1, length, constant, result_vec, inc2)
VEC_$DMULT_CONSTANT_I (mult_vec, inc1, length, constant, result_vec, inc2)
VEC_$IMULT_CONSTANT_I (mult_vec, inc1, length, constant, result_vec, inc2)
VEC_$IMULT_CONSTANT16_I(mult_vec, inc1, length, constant, result_vec, inc2)
```

INPUT PARAMETERS

mult_vec
Floating-point or integer vector from which data will be copied.

inc1
Increment for the index of mult_vec. This is a 4-byte integer.

length
Number of elements on which to operate. This is a 4-byte integer.

constant
Floating-point or integer constant to multiply by mult_vec.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the result.

INPUT PARAMETERS

inc2
Increment for the index of result_vec. This is a 4-byte integer.

USAGE

These routines multiply elements of a vector by a scalar constant and store the result in a second vector. The routines step through both vectors by increments. The routines perform the following:

```
      J = 1
      K = 1
      DO 10 I=1,LENGTH
           RESULT(J) = MULT_VEC(K) * CONSTANT
           J = J+INC2
           K = K+INC1
   10 CONTINUE
```

VEC_$POSTMULT

Postmultiplies a 4 x 4 matrix by a 4 x 1 column vector, returning the result in a second vector.

FORMAT

```
VEC_$POSTMULT (matrix, col_vec, result_vec)
VEC_$DPOSTMULT (matrix, col_vec, result_vec)
VEC_$IPOSTMULT (matrix, col_vec, result_vec)
VEC_$IPOSTMULT16 (matrix, col_vec, result_vec)
```

INPUT PARAMETERS

matrix
A 4 x 4 floating-point or integer matrix to be postmultiplied.

col_vec
A 4 x 1 floating-point or integer column vector.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the product.

USAGE

These routines postmultiply a 4 x 4 matrix by a 4 x 1 column vector, and return a 4 x 1 column vector. They are intended for use in graphics applications, and perform the following operation:

```
      DO 10 J = 1,4
         RESULT_VEC(J) = 0.0
         DO 10 I = 1,4
           RESULT_VEC(J) = RESULT_VEC(J) + COL_VEC(I)*MATRIX(J,I)
   10 CONTINUE
```

VEC_$POSTMULTN

Postmultiplies a variably-dimensioned matrix by an n x 1 vector, returning the result in a second vector.

FORMAT

```
VEC_$POSTMULTN (matrix, col_vec, m, n, result_vec)
VEC_$DPOSTMULTN (matrix, col_vec, m, n, result_vec)
VEC_$IPOSTMULTN (matrix, col_vec, m, n, result_vec)
VEC_$IPOSTMULTN16 (matrix, col_vec, m, n, result_vec)
```

INPUT PARAMETERS

matrix
An m x n floating-point or integer matrix to be postmultiplied.

col_vec
An n x 1 floating-point or integer column vector.

m, n
Dimensions. of the matrices. These are 4-byte integers.

OUTPUT PARAMETERS

result_vec
An m x 1 floating-point or integer vector containing the product.

USAGE

These routines postmultiply a m x n matrix by a n x 1 column vector, and return an m x 1 column vector. They perform the following operation:

```
      DO 10 I = 1,M
         RESULT_VEC(I) = 0.0
         DO 10 J = 1,N
            RESULT_VEC(I) = RESULT_VEC(I) + COL_VEC(J) * MATRIX(I,J)
   10 CONTINUE
```

VEC_$PREMULT

Premultiplies a 4 x 4 matrix by a 1 x 4 row vector, returning the result in a second vector.

FORMAT

```
VEC_$PREMULT (row_vec, matrix, result_vec)
VEC_$DPREMULT (row_vec, matrix, result_vec)
VEC_$IPREMULT (row_vec, matrix, result_vec)
VEC_$IPREMULT16 (row_vec, matrix, result_vec)
```

INPUT PARAMETERS

row_vec
A 1 x 4 floating-point or integer row vector.

matrix
A 4 x 4 floating-point or integer matrix to be premultiplied.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the product.

USAGE

These routines premultiply a 4 x 4 matrix by a 1 x 4 row vector, and return a 1 x 4 row vector. They perform the following operation:

```
   DO 10 I = 1,4
      RESULT_VEC(I) = 0.0
      DO 10 J = 1,4
         RESULT_VEC(I) = RESULT_VEC(I) + ROW_VEC(J) * MATRIX(J,I)
10 CONTINUE
```

VEC_$PREMULTN

Premultiplies a variably-dimensioned matrix by a 1 x n row vector, returning the result in a second vector.

FORMAT

```
VEC_$PREMULTN (row_vec, matrix, m, n, result_vec)
VEC_$DPREMULTN (row_vec, matrix, m, n, result_vec)
VEC_$IPREMULTN (row_vec, matrix, m, n, result_vec)
VEC_$IPREMULTN16 (row_vec, matrix, m, n, result_vec)
```

INPUT PARAMETERS

row_vec
A 1 x m floating-point or integer row vector.

matrix
An m x n floating-point or integer matrix to be premultiplied.

m, n
Dimensions of the matrices. These are 4-byte integers.

OUTPUT PARAMETERS

result_vec
One by n floating-point or integer vector containing the product.

USAGE

These routines premultiply a variably-dimensioned matrix by a 1 x m row vector, and return a 1 x n row vector. They perform the following operation:

```
         DO 10 I = 1,N
            RESULT_VEC(I) = 0.0
            DO 10 J = 1,M
               RESULT_VEC(I) = RESULT_VEC(I) + ROW_VEC(J) * MATRIX(J,I)
      10 CONTINUE
```

VEC_$SP_DP

Copies a single-precision vector to a double-precision vector.

FORMAT

```
VEC_$SP_DP (sp_vec, dp_vec, length)
```

INPUT PARAMETERS

sp_vec
Single-precision floating-point vector.

OUTPUT PARAMETERS

dp_vec
Double-precision floating-point resultant vector.

INPUT PARAMETERS

length
Number of elements to copy. This is a 4-byte integer.

USAGE

VEC_$SP_DP copies a single-precision vector to a double-precision resultant vector. The routine performs the following:

```
      DO 10 I=1, LENGTH
         DP_VEC(I) = DBLE (SP_VEC(I))
   10 CONTINUE
```

VEC_$SP_DP_I

Copies a single-precision vector to a double-precision vector, stepping through the vectors incrementally.

FORMAT

```
VEC_$SP_DP_I (sp_vec, inc1, dp_vec, inc2, length)
```

INPUT PARAMETERS

sp_vec
Floating-point, single-precision vector.

inc1
Increment for the index of sp_vec. This is a 4-byte integer.

OUTPUT PARAMETERS

dp_vec
Floating-point double-precision resultant vector.

inc2
Increment for the index of dp_vec. This is a 4-byte integer.

INPUT PARAMETERS

length
Number of elements to copy. This is a 4-byte integer.

USAGE

VEC_$SP_DP_I copies a single-precision vector to a double-precision resultant vector, stepping through the vectors by user-supplied increments. It does the following:

```
      J = 1
      K = 1
      DO 10 I=1, LENGTH
         DP_VEC(K) = DBLE (SP_VEC(J))
         J = J+INC1
         K = K+INC2
   10 CONTINUE
```

VEC_$SUB

Subtracts one vector from another.

FORMAT

```
VEC_$SUB (start_vec, sub_vec, length, result_vec)
VEC_$DSUB (start_vec, sub_vec, length, result_vec)
VEC_$ISUB (start_vec, sub_vec, length, result_vec)
VEC_$ISUB16 (start_vec, sub_vec, length, result_vec)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector from which the values will be subtracted.

sub_vec
Floating-point or integer vector to be subtracted.

length
Number of elements to subtract. This is a 4-byte integer.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the difference.

USAGE

These routines subtract one vector from another, returning the result in a third vector. The routines perform the following operation:

```
      DO 10 I = 1,LENGTH
       RESULT_VEC(I) = START_VEC(I) - SUB_VEC(I)
   10 CONTINUE
```

VEC_$SUB_I

Subtracts one vector from another, stepping through the vectors by increments.

FORMAT

```
VEC_$SUB_I (start_vec, inc1, sub_vec, inc2, length, result_vec, inc3)
VEC_$DSUB_I (start_vec, inc1, sub_vec, inc2, length, result_vec, inc3)
VEC_$ISUB_I (start_vec, inc1, sub_vec, inc2, length, result_vec, inc3)
VEC_$ISUB16_I (start_vec, inc1, sub_vec, inc2, length, result_vec, inc3)
```

INPUT PARAMETERS

start_vec
Floating-point or integer vector from which the values will be subtracted.

inc1
Increment for the index of start_vec. This is a 4-byte integer.

sub_vec
Floating-point or integer vector to be subtracted.

inc2
Increment for the index of sub_vec. This is a 4-byte integer.

length
Number of elements to subtract . This is a 4-byte integer.

OUTPUT PARAMETERS

result_vec
Floating-point or integer vector containing the difference.

INPUT PARAMETERS

inc3
Increment for the index of result_vec. This is a 4-byte integer.

USAGE

These routines subtract one vector from another, returning the result in a third vector. The indices to all three vectors are incremented by user-specified values. The routines perform the following operation:

```
      DO 10 I = 1,LENGTH
        RESULT_VEC(J) = START_VEC(K) - SUB_VEC(L)
        J = J + INC3
        K = K + INC1
        L = L + INC2
   10 CONTINUE
```

VEC_$SUM

Sums the elements of a vector.

FORMAT

```
sum = VEC_$SUM (vec, length)
sum = VEC_$DSUM (vec, length)
sum = VEC_$ISUM (vec, length)
sum = VEC_$ISUM16 (vec, length)
```

RETURN VALUE

sum
Floating-point or integer sum of first "length" elements of vec.

INPUT PARAMETERS

vec
Floating-point or integer vector to be summed.

length
Number of elements to sum. This is a 4-byte integer.

USAGE

These routines sum the elements of a vector. The routines perform the following operation:

```
      DO 10 I = 1,LENGTH
        SUM = SUM + VEC(I)
   10 CONTINUE
```

VEC_SUM_I

Sums the elements of a vector, stepping through the vector by increments.

FORMAT

```
sum = VEC_$SUM_I (vec, inc, length)
sum = VEC_$DSUM_I (vec, inc, length)
sum = VEC_$ISUM_I (vec, inc, length)
sum = VEC_$ISUM16_I (vec, inc, length)
```

RETURN VALUE

sum
Floating-point or integer sum of the elements.

INPUT PARAMETERS

vec
Floating-point or integer vector to be summed.

inc
Increment for the index of vec. This is a 4-byte integer.

length
Number of elements to sum. This is a 4-byte integer.

USAGE

These functions step through a vector incrementally, summing its elements. The sum is returned as the value of the function. The routines do the following:

```
      J = 1
      SUM = 0.0
      DO 10 I=1,LENGTH
           SUM = SUM+VEC(J)
           J = J+INC1
   10 CONTINUE
```

VEC_$SWAP

Swaps the elements of two vectors.

FORMAT

```
VEC_$SWAP (vec1, vec2, length)
VEC_$DSWAP (vec1, vec2, length)
VEC_$ISWAP (vec1, vec2, length)
VEC_$ISWAP16 (vec1, vec2, length)
```

INPUT/OUTPUT PARAMETERS

vec1, vec2
Floating-point or integer vectors to be swapped.

INPUT PARAMETERS

length
Number of elements to swap. This is a 4-byte integer.

USAGE

These routines swap the elements of two vectors. They perform the following operation:

```
      DO 10 I = 1,LENGTH
        TEMP = VEC1(I)
        VEC1(I) = VEC2(I)
        VEC2(I) = TEMP
   10 CONTINUE
```

VEC_$SWAP_I

Swaps the elements of two vectors, stepping through the vectors by increments.

FORMAT

```
VEC_$SWAP_I (vec1, inc1, vec2, inc2, length)
VEC_$DSWAP_I (vec1, inc1, vec2, inc2, length)
VEC_$ISWAP_I (vec1, inc1, vec2, inc2, length)
VEC_$ISWAP16_I (vec1, inc1, vec2, inc2, length)
```

INPUT/OUTPUT PARAMETERS

vec1
Floating-point or integer vector to be swapped.

INPUT PARAMETERS

inc1
Increment for the index of vec1. This is a 4-byte integer.

INPUT/OUTPUT PARAMETERS

vec2
Floating-point or integer vector to be swapped.

INPUT PARAMETERS

inc2
Increment for the index of vec2. This is a 4-byte integer.

length
Number of elements to swap. This is a 4-byte integer.

USAGE

These routines step through two vectors by increments, swapping their elements. They perform the following operation:

```
      J = 1
      K = 1
      DO 10 I=1,N
        TEMP = VEC1(J)
        VEC1(J) = VEC2(K)
        VEC2(K) = TEMP
        J = J + INC1
        K = K + INC2
   10 CONTINUE
```

VEC_$ZERO

Zeros a vector.

FORMAT

```
VEC_$ZERO (vector, length)
VEC_$DZERO (vector, length)
VEC_$IZERO (vector, length)
VEC_$IZERO16 (vector, length)
```

INPUT/OUTPUT PARAMETERS

vector
Floating-point or integer vector to be zeroed.

INPUT PARAMETERS

length
Number of elements to zero. This is a 4-byte integer.

USAGE

These routines zero the elements of a vector. They perform the following operation:

```
      DO 10 I = 1,LENGTH
        VEC(I) = 0.0
   10 CONTINUE
```

VEC_$ZERO_I

Zeros a vector by increments.

FORMAT

```
VEC_$ZERO_I (vector, inc, length)
VEC_$DZERO_I (vector, inc, length)
VEC_$IZERO_I (vector, inc, length)
VEC_$IZERO16_I (vector, inc, length)
```

INPUT/OUTPUT PARAMETERS

vector
Floating-point or integer vector to be zeroed.

INPUT PARAMETERS

inc
Increment for the index of vector. This is a 4-byte integer.

length
Number of elements to zero. This is a 4-byte integer.

USAGE

These routines step through a vector by increments, zeroing its elements. VEC_$ZERO_I zeros 32 bits regardless of data type and VEC_$DZERO_I zeros 64 bits. The routines perform the following:

```
       J = 1
       DO 10 I=1,LENGTH
           VEC(J) = 0.0
           J = J+INC1
    10 CONTINUE
```

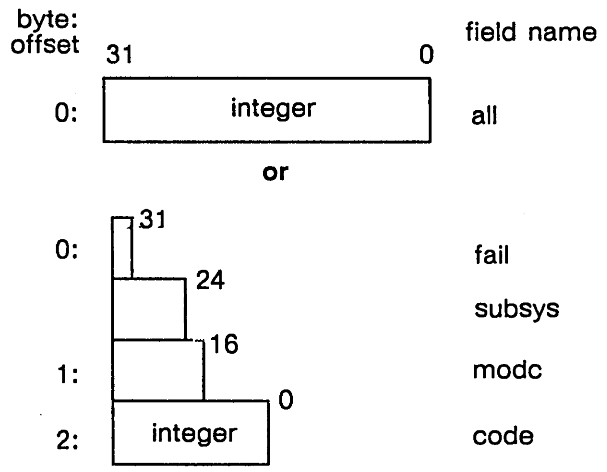
VFMT

This section describes the data types, the call syntax, and the error codes for the VFMT programming calls. Refer to the Introduction at the beginning of this manual for a description of data-type diagrams and call syntax format.

DATA TYPES

STATUS_$T

A status code. The diagram below illustrates the STATUS_$T data type:



Field Description:

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

subsys
The subsystem that encountered the error (bits 24 - 30).

modc
The module that encountered the error (bits 16 - 23).

code
A signed number that identifies the type of error that occurred (bits 0 - 15).

VFMT_$STRING_T

An array of up to 200 characters. Access control string.

VFMT_$DECODE

Decodes data from a text buffer and writes the decoded data into program variables.

FORMAT

```
return-value = VFMT_$DECODE{2|5|10} (control-string, text-buffer, size,
                                     count, status, a1, a2, ... a10)
```

RETURN VALUE

return-value

Position in the text buffer of the last decoded character. This is a 2-byte integer.

The return-value indicates the position in the text buffer of the last decoded character. The first buffer position is 1.

INPUT PARAMETERS

control-string

Character string giving instructions for decoding the input data. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

text-buffer

Buffer containing data to be decoded, in VFMT_$STRING_T format. This is an array of up to 200 characters.

size

Number of bytes of data in the text buffer. This is a 2-byte integer.

OUTPUT PARAMETERS

count

Number of fields successfully decoded. This is a 2-byte integer.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the VFMT Data Types section for more information.

a1, a2, ... a10

Up to ten variables containing decoded data.The number required depends on the number immediately following VFMT_$DECODE. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are decoding ASCII text strings, you must provide two variables for each text string: a character string to contain the decoded string, and a 2-byte integer variable to contain the length of the decoded string, which is returned by VFMT.

The returned length of the output string depends on what you specify in the control string. VFMT determines the output string length using the M directive or, if M is omitted, using the previous value of the string length variable. In the latter way you may use this integer variable to input a maximum string length.

If you specify a maximum string length with the M directive, and the length of the string decodes is less than the maximum, VFMT returns the actual length (the lesser value) in the string-length variable.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT_$DECODE2, VFMT_$DECODE5, and VFMT_$DECODE10. The number immediately following VFMT_$DECODE indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

VFMT_$ENCODE

Encodes and writes data into a text buffer.

FORMAT

```
VFMT_$ENCODE{2|5|10}  (control-string, text-buffer, capacity, size,
                       a1, a2 , ... a10)
```

INPUT PARAMETERS

control-string

Character string giving instructions for encoding the output data. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

OUTPUT PARAMETERS

text-buffer

Buffer to contain the encoded data, in VFMT_$STRING_T format. This is an array of up to 200 characters.

INPUT PARAMETERS

capacity

Maximum number of characters that may be placed in the text-buffer. This is a 2-byte integer.

OUTPUT PARAMETERS

size

Number of characters placed in the buffer. This is a 2-byte integer.

INPUT PARAMETERS

a1, a2 , ... a10

Up to ten variables containing data for encoding. The number required depends on the number immediately following VFMT_$ENCODE. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are encoding ASCII text strings, you must provide two variables for each text string: a character string containing the string, and a 2-byte integer variable containing the length of the string.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT_$ENCODE2, VFMT_$ENCODE5, and VFMT_$ENCODE10. The number immediately following VFMT_$ENCODE indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

VFMT_$READ

Reads character data from standard input and decodes them into variables.

FORMAT

```
VFMT_$READ{2|5|10} (control-string, count, status, a1, a2, ... a10)
```

INPUT PARAMETERS

control-string
A character string giving instructions for decoding the input data. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

OUTPUT PARAMETERS

count
Number of fields successfully decoded. This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the VFMT Data Types section for more information.

a1, a2, ... a10
Up to ten variables containing decoded data. The number required depends on the number immediately following VFMT_$READ. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are decoding ASCII text strings, you must provide two variables for each text string: a character string to contain the decoded string, and a 2-byte integer variable to contain the length of the decoded string, which is returned by VFMT.

The returned length of the output string depends on what you specify in the control string. VFMT determines the output string length using the M directive or, if M is omitted, using the previous value of the string length variable. In the latter way you may use this integer variable to input a maximum string length.

If you specify a maximum string length with the M directive, and the length of the string decodes is less than the maximum, VFMT returns the actual length (the lesser value) in the string-length variable.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT_$READ2, VFMT_$READ5, and VFMT_$READ10. The number immediately following VFMT_$READ indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

VFMT _ $RS

Reads character data from a stream and decodes them into variables.

FORMAT

```
VFMT_$RS{2|5|10} (stream-id, control-string, count, status, a1, a2, ...a10)
```

INPUT PARAMETERS

stream-id

Number of the stream from which data are read, in STREAM _ $ID _ T format. This is a 2-byte integer.

control-string

A character string giving instructions for decoding the input data. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

OUTPUT PARAMETERS

count

Number of fields successfully decoded. This is a 2-byte integer.

status

Completion status, in STATUS _ $T format. This data type is 4 bytes long. See the VFMT Data Types section for more information.

a1, a2, ... a10

Up to ten variables containing decoded data. The number required depends on the number immediately following VFMT _ $RS. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are decoding ASCII text strings, you must provide two variables for each text string: a character string to contain the decoded string, and a 2-byte integer variable to contain the length of the decoded string, which is returned by VFMT.

The returned length of the output string depends on what you specify in the control string. VFMT determines the output string length using the M directive or, if M is omitted, using the previous value of the string length variable. In the latter way you may use this integer variable to input a maximum string length.

If you specify a maximum string length with the M directive, and the length of the string decoded is less than the maximum, VFMT returns the actual length (the lesser value) in the string-length variable.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT _ $RS2, VFMT _ $RS5, and VFMT _ $RS10. The number immediately following VFMT _ $RS indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

VFMT_$WRITE

Encodes data and writes them to standard output.

FORMAT

```
VFMT_$WRITE{2|5|10}  (control-string, a1, a2 , ... a10)
```

INPUT PARAMETERS

control-string
A character string containing the control information for encoding. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

a1, a2 , ... a10
Up to ten variables containing data for encoding. The number required depends on the number immediately following VFMT_$ENCODE. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are encoding ASCII text strings, you must provide two variables for each text string: a character string containing the string, and a 2-byte integer variable containing the length of the string.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT_$WRITE2, VFMT_$WRITE5, and VFMT_$WRITE10. The number immediately following VFMT_$WRITE indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

Any individual VFMT_$WRITE[2|5|10] call can write out a maximum of 512 bytes per call.

VFMT_$WS

Encodes data and writes them to a stream.

FORMAT

```
VFMT_$WS{2|5|10}  (stream-id, control-string, a1, a2 , ... a10)
```

INPUT PARAMETERS

stream-id
The number of the stream to which data are written, in STREAM_$ID_T format. This is a 2-byte integer.

control-string
A character string containing the control information for encoding. See the VFMT chapter of the *Programming With General System Calls* manual for details about how to construct control strings.

a1, a2 , ... a10
Up to ten variables containing data for encoding. The number required depends on the number immediately following VFMT_$ENCODE. The number of variables can not exceed this number. In Pascal programs, you must specify exactly this number of variables, using dummy variables if necessary.

If you are encoding ASCII text strings, you must provide two variables for each text string: a character string containing the string, and a 2-byte integer variable containing the length of the string.

USAGE

This is actually a description of three separate DOMAIN system calls: VFMT_$WS2, VFMT_$WS5, and VFMT_$WS10. The number immediately following VFMT_$WS indicates the maximum number of variables the call can handle. The number must follow immediately, with no embedded spaces.

Any individual VFMT_$WS[2|5|10] call can write out a maximum of 512 bytes per call.

ERRORS

VFMT_$UNTERMINATED_CTL_STRING
Unterminated control string.

VFMT_$INVALED_CTL_STRING
Invalid control string.

VFMT_$TOO_FEW_ARGS
Too few arguments supplied for read/decode.

VFMT_$FW_REQUIRED
Field width missing on "(" designator.

VFMT_$EOS
Encountered end of string where more text was expected.

VFMT_$NULL_TOKEN
Encountered null token where numeric token was expected.

VFMT_$NONNUMERIC_CHAR
Non-numeric character found where numeric was expected.

VFMT_$SIGN_NOT_ALLOWED
Sign encountered in unsigned field.

VFMT_$VALUE_TOO_LARGE
Value out of range in text string.

VFMT_$NONMATCHING_CHAR
Character in text string does not match control string.

VFMT_$NONMATCHING_DELIMITER
Terminator in text string does not match specified terminator.

STATUS_$OK
Successful completion.