Multics Technical Bulletin MTB-077

To: Distribution

From: Don MacLaren

Subject: MPM Documentation for I/O System

Date: 05/15/74

This MTB contains draft sections of the Honeywell MPM for release 1 of Multics. The sections included are those that describe the new I/O system:

- 3.5 Storage System Files
- 4.1 Summary of I/O Facilities
- 4.3 The Multics I/O System
- 4.4 File I/O
- io_call, command
- iox_, subroutine
- discard_, I/O module
- ntape_, I/O module
- syn_, I/O module
- tty_, I/O module
- vfile_, I/O module
- Subsystem Writer's Guide Section 3.7, The I/O Control Block Subsystem
- Subsystem Writer's Guide Section 3.8, Writing an I/O Module
- io_call, command, Subsystem Writer's Guide
- iox_, subroutine, Subsystem Writer's Guide

Comments on errors and obscurities will be appreciated. Send them to me at CISL or by Multics mail (MacLaren Multics). Note that the system being described already exists. Therefore suggestions for changes and additions should be submitted through the usual channels.

SECTION 3.5

Storage System Files

In the storage system, a file is either a single segment or a multi-segment file. The latter is a directory of a special form, which is described below. System commands and subroutines that manipulate files handle the transition between the single-segment and multi-segment forms of a file automatically.

Note that every segment is considered to be a file. Thus one may create a file through a command that works only with segments (e.g., create or edm), and then manipulate it later through a command that works with segments or multi-segment files (e.g., dprint or delete).

Multi-Segment Files

A multi-segment file is a directory whose multi-segment file indicator has a value greater than zero. The segments contained in the directory are the components of the multi-segment file. The directory, f, and the components should have the following properties:

1. The names of the components are the ASCII representations of the numbers 0, 1, ..., n-1, where n is the value of the multi-segment file indicator. The names are unsigned, contain no blanks, and, except for "0", contain no leading zeros.

2. The directory, f, contains no other entries, and the components have no additional names. Thus a multi-segment file should not contain links or directories.

3. The ACL of each component is exactly the same as the initial ACL for segments of f.

4. The ACL of f (in its parent directory) is the same as the initial ACL for segments in f except that where the initial ACL for segments has access mode "r", the ACL for f has access mode "s", and where the initial ACL for segments has access modes "rw" the ACL for f has access modes "sma".

5. The ring brackets, safety switch, and maximum length attributes have the same values for all components.

Commands and subroutines that manipulate multi-segment files may not work properly on files that fail to have properties (1)-(5) above. If a multi-segment file is found to be inconsistent in this respect, it should be made consistent before further use of it as a file.

Access Requirements for Multi-Segment Files

To read a multi-segment file, f, the user must have "r" access on the components and "s" access on f itself. This is referred to simply as "r" accessn the file.

To modify a multi-segment file, f, the user must have "rw" access on the components and "sma" access on the file itself. This is referred to simply as "rw" access on the file. (If the user does not add or delete components, he needs only have "s" access on f, but the combination of "rw" access on a segment and "s" access on f should not occur.)

To turn a single segment into a multi-segment file or vice-versa, the user must have sma access on the directory containing the file (in addition to proper access on the file itself).

SECTION 4.1

SUMMARY OF I/O FACILITIES

This section is a brief guide to the vairous I/O facilities available in Multics and the related documentation.

Bulk I/O

For printing and punching of files see the MPM write-ups for:

dprint	prints files, using a line printer
dpunch	punches files on cards

For the format of output produced by these commands, and for an explanation of how cards are read into the storage system, see the MPM Reference Guide Section, Bulk I/O.

The I/O System

The Multics I/O System supports I/O in an essentially device-independent manner. To facilitate control of the sources and targets for I/O, the system makes use of a software construction called an I/O switch. An I/O switch is rather like a channel in that it controls the flow of data between a program's memory and devices, files, etc. Before I/O can be done through a switch, the switch must be attached. The attachment specifies the source/target for I/O operations and the particular I/O module that will perform the operations. For example, a switch may be attached to the user's console through the I/O module tty_ or to a file in the storage system through the I/O module vfile_.

A general description of the I/O system is contained in the MPM Reference Guide Section, the Multics I/O System. The basic tool for making attachments and performing I/O operations is the subroutine iox_. Its write-up gives detailed descriptions of the various operations.

Attachments and I/O operations can also be done from command level. See the MPM write-up of the command io_call.

The command print_attach_table prints descriptions of all current attachments. See its MPM write-up.

Obsolete Terminology

Earlier versions of Multics used a different, but similar, I/O system. Parts of the system documentation may still use the terminology of the old I/O system. In particular, the term "i/o stream" may be used instead of "I/O switch", and the terms "DIM" and "IOSIM" may be used instead of "I/O module". Also the documentation may speak of attaching to a device, even though the attachment may be to something other than a device, e.g., a file in the storge system.

Language I/O

Each programming language has its own I/O facilities, which use the Multics I/O system in a manner appropriate to the language. For details on a languages facilities, and for discussion of the usage of related Multics commands, see the reference manual and/or user's guide for that language.

The close_file command closes PL/I and fortran "files". The files in question are actually control blocks used by the language I/O routines.

System I/O Modules

The Multics system contains the following I/O modules.

discard_	a sink for output
ntape_	I/O from/to files on tape
syn_	establishes one switch as a synonym for another
tty_	I/O from/to terminals
vfile_	I/O from/to files in the storage system

For details on a particular I/O module (e.g., how to make an attachent through it), see the module's write-up in the MPM Section, subroutines.

User-Written I/O Modules

See the Subsystem Writer's Guide Section, Writing an I/O Module.

Console I/O

The command "io_call modes..." may be used to control characteristics of console I/O such as the line length, insertion of tabs, and erase and kill processing. See the MPM write-ups of the command io_call and the I/O module tty_.

The file output command cuases all subsequent ouput normally printed on the user's console to be written instead to a file in the storage system. The console_output command causes such output to be directed again to the console. See the MPM Command Sections describing these commands.

The contents of segments that contain exclusively Multics ASCII characters may be printed on the console by invoking the print command. The contents of any segment can be printed in octal using the dump_segment command or the Multics debugger, debug. See the MPM Command Sections describing these commands.

The subroutine ioa_ provides a convenient means for formatting output to be printed on the console, and it may be used for other output as well. See its write-up in the MPM Section, Subroutines.

NOTE ON 4.2

Use the current section 4.4 (Bulk Input and Output) with the following changes.

Change "segment" to "file" in the following places: p.1, l.7 (ignoring headers and blank lines), l.14, and l.19.

Change the section title to "Bulk I/O".

SECTION 4.3

The Multics I/O System

The Multics I/O system handles what might be called "logical I/O" rather than "hardware I/O". On the one hand, it includes I/O from/to files in the storage system, which really involves only the transfer of data from one memory location to another. On the other hand, it excludes the most important case of hardware I/O, namely the transfer of pages between secondary storage and main memory. Most I/O operations refer only to logical properties (e.g., the next record, the number of characters in a line) rather than to particular device characteristics or file formats. True hardware I/O is performed by routines that are not normally called by a user.

This section gives some general information on the I/O system, especially in regards "I/O switches", "attaching", and "opening". Full details on the various I/O operations are given in the MPM writeup of the subroutine iox_. All functions of the I/O system are accessable through calls to this routine.

The command io_call provides many of the same functions at command level. Its writeup gives summary descriptions of the I/O operations.

How to Perform I/O

To perform I/O, carry out the steps listed below. In general, a step may be performed by a call to iox_ or by use of the io_call command. The I/O facilities of the programming languages may also be used to carry out these steps, but that topic is outside the scope of this section.

Step 1. Attach an I/O switch. This step specifies a source/target for subsequent I/O operations and names the I/O module that will perform the operations. Example:

```
io_call attach input_sw vfile_ some_file
```

This command line attaches the switch named "input_sw" to a storage_system file whose relative pathname is "some_file". The I/O module is named "vfile_".

Step 2. Open the I/O switch. This step prepares the switch for a particular mode of processing (e.g., reading records sequentially) using the already established attachment. Example:

```
call iox_$open(iocb_ptr, 4, ., code);
```

The iocb_ptr identifies the switch (see the paragraph on I/O switches below). The arguments 4 and 1 mean that the opening is for sequential reading. See the description of the iox_ subroutine for full details.

Step 3. Perform the required data transfer and control I/O operations working through the switch. For example, read one record at a time until an end-of-information code is returned by the read operation. Example (of one read):

```
     call iox_$read_record (iocb_ptr, buffer_ptr, buffer_length,
          actual_record_length, code);
```

Step 4. Close the I/O switch. This step cleans up by writing out buffers, marking the end of a file, etc. The I/O switch is restored to the state it was in after Step 1, and the close could be followed by a repeat of Steps 2-4, perhaps with a different opening mode. Example:

```
     call iox_$close (iocb_ptr, code);
```

Step 5. Detach the I/O switch. After this step, the switch can be attached again for some other purpose. Example:

```
     io_call detach input_sw
```

In general, only Step 1 (attach) involves peculiarities of a particular type of device or a particular file format. It is often convenient to have this step and Step 5 (detach) performed from command level, while Steps 2-4 are performed by a program. This can make the program "device independent".

I/O Switches

Each I/O switch has four associated values that are of interest to users of the I/O system.

1. The switch name. This is a character string of length less than or equal to 32 (and greater than zero), not containing blanks.

2. The control block pointer. This is a pointer to a control block associated with the switch. The control block is maintained by the I/O system, and its contents are not of interest to the user.

3. The attach description. This is a character string describing the attachments of the switch. When the

string is empty, the switch is said to be detached. When the string is not empty the switch is said to be attached.

4. The open description. This is a character string describing the opening of the switch. When the string is empty, the switch is said to be closed. When the string is not empty, the string is said to be open. A switch is never open unless it is also attached.

The switch name is used to refer to the I/O switch at command level, and in other contexts where reference by a character string name is appropriate. Most calls to iox_ reference an I/O switch by its control block pointer. Given the switch name, the subroutine iox_$find_iocb returns the control block pointer.

Note that each I/O switch belongs to a particular ring, normally the user ring. Within a ring, switch_names are unique, but switches in different rings may have the same name.

Attaching a Switch

To attach a switch, use the command "io_call attach..." or one of the subroutines iox_$attach_iocb and iox_$attach_ioname. In all cases an attach description must be given. This string has the following form:

```
module_name -option_1-...-option_n-
```

The substrings module_name, option_1, ..., option_n must not contain blanks and must be separated by one or more blanks. The whole attach description may contain trailing blanks but not leading blanks.

The substring module_name determines the I/O module for the attachment as follows: If it does not contain any instances of ">" or "<", then it is interpreted as a reference name, and the I/O module is found by the search rules. If module_name contains ">" or "<", then it is interpreted as the path name (absolute or relative) of the I/O module.

The substrings option_1, ..., option_n must conform to the requirements of the particular I/O module. See its MPM write-up for details.

When the attachment is made, if the I/O module is not already initiated by the specified reference name, it is so initiated. In the case where module_name is given as a pathname, the reference name is the final entry name in the pathname.

Note that the attach description associated with the attached switch (and accessible through the print_attach_table command) may not be exactly the same as the attach description given to io_call, iox_$attach_iocb, or iox_$attach_ioname. In general, the I/O module transforms the attach_description into a standard form. For example, the command

```
          io_call attach foo >ldd>sdd>vfile_ my_file
```

might generate the attach description

```
          vfile_ >udd>m>J_Doe>my_file
```

Opening a Switch

To open a switch, use the command "io_call open ..." or the subroutine iox_$open. In either case on one of the opening modes listed in Table 1 must be specified. As shown in Table 1, the opening mode determines which I/O operations may be carried out through the open switch. Whether or not opening in a particular mode is possible depends on the attachment of the switch. The relation between opening modes and file attachments is discussed in the MPM Section, File I/O. For other types of attachments see the MPM write-up of the particular I/O module.

TABLE 1 - OPENING MODES AND ALLOWED I/O OPERATIONS

I/O Operation Opening Mode	get_line	get_chars	put_chars	read_record	rewrite_record	delete_record	read_length	position	seek_key	read_key	close	control	modes	write_record
1. stream_input	x	x						2			x	1	1	
2. stream_output			x								x	1	1	
3. stream_input_output	x	x	x					2			x	1	1	
4. sequential_input				x			x	x			x	1	1	
5. sequential_output											x	1	1	x
6. sequential_input_output				x			x	x			x	1	1	x
7. sequential__update				x	x	x	x	x			x	1	1	
8. keyed_sequential_input				x			x	x		x	x	1	1	
9. keyed_sequential_output									x		x	1	1	x
10. keyed_sequential_update				x	x	x	x	x	x	x	x	1	1	x
11. direct_input				x			x				x	1	1	
12. direct_output									x		x	1	1	x
13. direct_update				x	x	x	x				x	1	1	x

1. Depends on the attachment

2. Allowed if attached to a file in the file system.

Synonym Attachments

By means of the I/O module syn_, one I/O switch, switch_1, may be attached as a synonym for another I/O switch, switch_2. In general, performing an I/O operation through switch_1 will then have the same effect as performing it through switch_2. There are two exceptions.

1. Detaching switch_1 simply breaks the synonymization and has no effect on switch_2.

2. The attach description for the synonym attachment may specify that certain operations are to be inhibited. An attempt to perform an inhibited operation through switch_1 will simply result in an error code.

Synonym attachments are especially useful when one wishes to switch the source/target for a set of I/O operations. For example, the I/O switch user_output is normally attached as a synonym for user_i/o (the user's console). The following commands switch the output to a file.

```
io_call attach file_switch vfile_ file_name -extend
io_call open file_switch stream_output
io_call detach user_output
io_call attach user_output syn_ file_switch
```

The following commands put things back to normal.

```
io_call detach user_output
io_call attach user_output syn_ user_i/o
io_call close file_switch
io_call detach file_switch
```

Note that this is only an example. There are special commands (file_output and console_output) to handle this particular case of switching.

It is possible to have a chain of synonyms, e.g., switch_1 as a synonym for switch_2 and switch_2 as a synonym for switch_3. The final switch in the chain is the actual I/O switch for all the other switches in the chain. A more precise definition is as follows. If an I/O switch, switch_1, is not attached as a synonym, then its associated actual I/O switch is itself. If switch_1 is attached as a synonym for switch_2, then the actual I/O switch associated with switch_1 is the same as the actual I/O switch associated with switch_2.

With the notion of the actual I/O switch, the effect of a synonym attachment of an I/O switch, switch_1, can be precisely described:

1. The open_description of switch_1 will be the same as the open_description of the actual I/O switch associated with switch_1. (Hence switch_1 is open or closed according as the actual switch is open or closed.)

2. If the I/O operation open or one of the I/O operations listed in Table 1 is performed through switch_1, then the effect is the same as if it were performed through the actual I/O switch associated with switch_1 with one exception. The exception is that if any synonym attachment in the chain (connecting switch_1 to the actual I/O switch) inhibits the operation, then the only effect is to return an error code.

Standard I/O Switches

Four I/O switches are attached as part of the standard initialization of a Multics process.

Switch	Normal Attachment
user_i/o	the user's console
user_input	synonym for user_i/o
user_output	synonym for user_i/o
error_output	synonym for user_i/o

These switches may be attached in other ways but user_input, user_output, and error_output must always be attached as synonyms.

The following external variables are initialized to point to the control blocks for the corresponding I/O switches. These variables must never be modified.

```
     iox_$user_io
     iox_$user_input
     iox_$user_output
     iox_$error_output
```

By using these variables one can avoid calls to iox_$find_iocb and avoid testing for initialization of one's own static variables. Thus

```
          call iox_$put_chars (iox_$user_output,buffptr,bufflen,code);
```

is the simple and efficient way to write to the console.

Language I/O

It is possible to perform I/O through a particular switch using both the facilities of a programming language and the facilities of the I/O system (invoked directly). However, a direct call to the I/O system will have no effect on control blocks and buffers maintained by the language I/O routines, and this is likely to cause garbled input or output. The following statements about this sort of sharing of switches apply in most cases, but the language manuals should be consulted for precise details.

1. The I/O system may be used to attach a switch or to attach and open it. The language I/O routines are prepared for this, and they will close (detach) a switch only if they opened (attached) it.

2. A switch opened for stream_input may be used both directly and through lanuage I/O if care is exercised. In general, the languages read a line at a time. Thus the order of input may get confused if a direct call is made to the I/O system while the language routines are processing a line. Trouble is most likly to arise after a quit.

3. A switch opened for stream_output may be used both directly and through language I/O if formatting by column number, line number, page number, etc. is not important. Some shuffling of output may be expected, especially if a direct call to the I/O system is made while the language I/O routines are processing an I/O statement. Quits are most likely to cause this.

4. If a switch is opened for record I/O (sequential_, keyed_sequential_, and direct_ modes), using it both directly and through language I/O is not recommended.

Interrupted I/O Operations

It may happen that an I/O operation being performed on a particular I/O switch, s, is interrupted by a signal, e.g., by a quit signal or an access violation signal. In general, until the interrupted operation is completed, or until s is closed, it is an error (with unpredictable consequences) to perform any I/O operation except close on s. However, some I/O modules, tty_, in particular, allow other operations on s in this situation. See the module writeups for details. If the switch s is closed while the operation is interrupted, control must not be returned to the interrupted operation.

SECTION 4.4

FILE I/O

The I/O system distinguishes three types of files: unstructured, sequential, and indexed. These types pertain to the logical structure of a file, not to the files representation in storage, on magnetic tape, etc. For example, in the storage system a file may be stored as a single segment or as a multi-segment file; but this does not affect the meaning of I/O operations on the file.

Unstructured Files

The file contains a sequence of 9-bit bytes. Normally the bytes are ASCII characters, but this is not required.

The following I/O operations apply to unstructured files.

get_line	reads a line from the file, i.e., a sequence of bytes ending with an ASCII new-line character.
get_chars	reads a specified number of bytes.
put_chars	adds bytes at the end of the file.
position	positions to the beginning or end of the file, skips forward or backwards over a specified number of lines.

Sequential Files

The file contains a sequence of records. Each _record_ is a string of 9-bit bytes. A record may be zero length.

The following I/O operations apply to sequential files.

read_record	reads the next record
read_length	obtains the length of the next record.
write_record	adds a record to the file.
rewrite_record	replaces a record
delete_record	deletes a record.
position	positions to the beginning or end of the file, skips forward or backwards over a specified number of records.

Indexed Files

The file contains a sequence of records and an index. Each record is a string of 9-bit bytes. A record may be zero length.

The index associates each record with a key. A key is a string of from zero to 256 ASCII characters containing no trailing blanks. No two records in the file have the same key. The order of records in the sequence is key order: record x precedes record y if and only if the key of x is less than the key of y according to the Multics PL/1 rules for string comparision (lexicographic order using the ASCII collating sequence).

All the I/O operations applicable to sequential files apply to indexed files as well. In addition the following two operations manipulate keys.

read_key obtains the key of the next record

seek_key positions to the record with a given key or defines the key that will be associated with a record to be added by a following write operation.

File Opening

When an I/O switch is attached to a file and is opened for input or update, the file must exist and must be compatible with the opening mode. Table 2 show the compatibility between file types and opening modes.

When the opening is for output or input_output, and the file does not exist, a file of the appropriate type is created. The type of file created by a particular mode of opening is shown in Table 2.

When the opening is for output or input_output, and the file already exists, it is normally replaced by an empty file of the appropriate type. However, if either the attachment or the opening specifies extension of the file, the file is not replaced. In this case the file must be compatible with the opening mode.

Note that for files, opening for input_output means opening with the intent of first writing the file and then reading it during the same opening, and an existing file will be replaced by an empty file unless extension is specified.

TABLE 2 - COMPATIBLE FILE ATTCHCMENTS

File Type Opening Mode	unstructured	sequential	indexed
1. stream_input	x	1	1
2. stream_output	x,c		
3. stream_input_output	x,c		
4. sequential_input		x	x
5. sequential_output		x,c	
6. sequential_input_output		x,c	
7. sequential_update		2	x
8. keyed_sequential_input			x
9. keyed_sequential_output			x,c
10. keyed_sequential_update			x
11. direct_input			x
12. direct_output			x,c
13. direct_update			x

1. The structure of the file is ignored and everything in it is treated as d (including control words).

2. The file must be in the storage system.

3. This type of file is created by an output opening for the specified mode without -extend.

File Closing

When an I/O switch attached to a file has been opened for output, input_output, or update, a close operation should be performed on the switch before the process is terminated. If not, the file may be left in an inconsistent state, e.g. an end of file mark may not be written for a tape file, or the bit count of a segment may not be set for a storage system file.

The default handler for the finish condition closes all I/O switches.

File Position Designators

The I/O operations on files are defined in terms of four position designators. In cases where several I/O switches are open and attached to the same file, each opening has its own set of designators. The designators are:

next byte — the first byte to be read by the next get_line or get_chars operation

next record — the record to be read by the next read_record operation

current record — the record to be replaced or deleted by the next rewrite_record or delete_record operation

key for inserion — the key to be associated with the record added to an indexed file by the next write_record operation

The initial values for these designators are shown in Table 3.

I/O Modules For Files

The ntape_ I/O module supports files on magnetic tapes. The vfile_ I/O module supports files in the storage system. See the write-ups of these modules for details on their use. See also the general discussion in the MPM Section, Storage System Files.

TABLE 3 - FILE POSITION DESIGNATORS AT OPEN

Designator Opening Mode	next byte	next record	current record	key for insertion
1. stream_input	first byte			
2. stream_output	end of file			
3. stream_input_output	end of file			
4. sequential_input		first record		
5. sequential_output				
6. sequential_input_output		end of file		
7. sequential_update		first record	null	
8. keyed_sequential_input		first record		null
9. keyed_sequential_output				null
10. keyed_sequential_update		first record	null	null
11. direct_input				
12. direct_output				null
13. direct_update				null

In the openings where no value is indicated for a designator, the designator is not relevant.

COMMAND

Name: io_call, io

This command performs an operation on a designated I/O switch.

Usage

```
  io_call opname switchname -control_arg_1-...-control_arg_n-
```

1) opname designates the operation to be performed.

2) switchname is the name of the I/O switch.

3) control_arg_n depends on the particular operation.

The following opnames are permitted; they are described individually under usage below:

attach
close
control
delete
delete_record
detach
detach_iocb
find_iocb
get_chars
get_line
modes
open
position
put_chars
read
read_key
read_length
read_record
rewrite
rewrite_record
seek_key
write
write_record

Notes

Usage for each operation is shown below under the heading "Usage: opname". In some cases the opname has an abbreviation, and the heading is "Usage: opname, abbreviation".

If a control block for the I/O switch does not already exist, one is created.

The explanations of the operations cover only the main points of interest and, in general, treat only the cases where the I/O switch is attached to a file or device. For full details see the writeups of the subroutine iox_ and the I/O modules.

Usage: attach

```
   io_call attach switchname modulename -control_arg_1-
         -control_arg_n-
```

1) modulename is the name of the I/O module to be used in the attachment.

2) control_arg_i depends on the particular I/O module.

This command attaches the I/O switch using the designated I/O module. The attach description is the concatenation of modulename and control_arg_1,...,control_arg_n, separated by blanks. The attach description must conform to the requirements of the I/O module.

If a control block for the I/O switch does not already exist, one is created.

Usage: detach, detach_iocb

io_call detach switchname

This command detaches the I/O switch.

Usage: open

io_call open switchname mode -control_arg-

1) mode is one of the thirteen opening modes:

stream_input
stream_output
stream_input_output
sequential_input
sequential_output
sequential_input_output
sequential_update
keyed_sequential_input
keyed_sequential_output
keyed_sequential_update
direct_input
direct_output
direct_update

2) control_arg may be "-extend".

This command opens the I/O switch with the specified opening mode. If "-extend" is specified an existing file is not replaced. This option applies only to openings for output or input_output with the switch attached to a file (as opposed to a device).

Usage: close

io_call close switchname

This command closes the I/O switch.

Usage: get_line

io_call get_line switchname -n-

1) n may be a decimal greater than zero specifying the maximum number of characters to be read.

This command reads the next line from the file or device to which the I/O switch is attached and prints the line on the console. If n is given, and the line is longer than n, then only the first n characters are read.

Usage: get_chars

```
io_call get_chars switchname n
```

This command reads the next n characters from the file or device to which the I/O switch is attached and prints the characters on the console.

Usage: put_chars

```
io_call put_chars switchname string -control_arg-
```

1) string may be any character string.

2) control_arg may be "-nnl".

If the I/O switch is attached to a device, this command transmits the characters in string to the device. If the I/O switch is attached to an unstructured file, the string is added to the end of the file. In either case, a newline character is first added to the end of the string unless "-nnl" is specified.

Usage: read_record, read

```
io_call read_record switchname n
```

1) n is a decimal integer greater than zero.

This command reads the next record from the file to which the I/O switch is attached. The record is read into a buffer of length n. The record (or the part of it that fits into the buffer) is printed on the console.

Usage: write_record, write

```
io_call write_record switchname string
```

1) string is any string.

This command adds a record to the file to which the I/O switch is attached. The record is equal to string. If the file is a sequential file, the record is added at the end of the file. If the file is an indexed file, the record's key must have been defined by a preceding seek_key operation.

Usage: rewrite_record, rewrite

io_call rewrite_record switchname string

1) string is any string.

This command replaces the current record in the file to which the I/O switch is attached. The new record equals string. The current record must have been defined by a preceding read_record, seek_key, or position operation as follows:

read_record current record is record read.

seek_key current record is record with the designated key.

position current record is the record preceding the record to which the file was positioned.

Usage: delete_record, delete

io_call delete_record switchname

This command deletes the current record in the file to which the I/O switch is attaced. The current record is determined as in rewrite_record above.

Usage: position

io_call position switchname type -n-

1) type is -1, 0, or 1.

2) n is a decimal integer. It must be present if type is 0.

This command positions the file to which the I/O switch is attached. If type is -1, the file is positioned to its beginning, so that the next record is the first record, (structured files), or so that the next byte is the first byte (unstructured files). If type is +1, the file is positioned to its end; the next record (or next byte) is at the end of file position. If type is 0, the file is positioned forwards (n>=0) or backwards (n<0) over records (structured files) or lines (unstructured files). The number of records or lines skipped is determined by the absolute value of n.

In the case of unstructured files, the next byte position after the operation will be at a byte immdiately following a new line character (or at the first byte in the file or at the end of the file); and the number of newline characters moved over is the absolute value of n.

If the I/O switch is attached to a device, only forward skips (type=0, n>=0) are allowed. The effect is to discard the next n lines input from the device.

Usage: seek_key

```
     io_call seek_key switchname key
```

1) key   is a string of ASCII characters with length, <= length<=256.

This command positions the indexed file to which the I/O switch is attached to the record with the given key. The record's length is printed. Trailing blanks in the key are ignored.

If the file does not contain a record with the specified key, it becomes the key for insertion. A following write_record operation will add a record with this key.

Usage: read_key

```
     io_call read_key switchname
```

This command prints the key and record length of the next record in the indexed file to which the I/O switch is attached. The file's position is not changed.

Usage: read_length

```
     io_call read_length switchname
```

This command prints the length of the next record in the structured file to which the I/O switch is attached. The file's position is not changed.

Usage: control

```
     io_call control switchname order -string-
```

1) order   is one of the orders accepted by the I/O module used in the attachment of the I/O switch.

2) string   may be any string.

This command applies only when the I/O switch is attached via an I/O module that supports the control I/O operation. The command calls iox_$control (iocb_ptr, order, infoptr, code), where iocb_ptr designates the I/O switch. For full details see the writeup of iox_$control. If string is not given, a null infoptr will be passed. If string is given, infoptr will point to a varying string that equals string and whose maximum length is equal to the length of string. Upon return from iox_$control, if infoptr points to a varying string of length, 1<=length<=256, then the string is printed. Otherwise, if infoptr has some other nonnull value, it is printed.

Usage: modes

```
     io_call modes switchname -string-
```

1) string   may be a sequence of modes separated by commas.
                   The string not must not contain blanks.

This command applies only when the I/O switch is attached via an I/O module that supports modes. The command prints the existing modes and, if string is given, sets the new modes as specified by string.

If switch name is "user_i/o", the command refers to the modes controlling the user's console. See the writeup of the I/O module tty_ for a description of applicable modes.

Usage: find_iocb

```
     io_call find_iocb switchname
```

This command prints the location of the control block for the I/O switch. If it does not already exist, the control block is created.

MPM SUBROUTINE

Name: iox_

This procedure performs I/O operatons and some related functions. The user should be familiar with the contents of the MPM sections, the Multics I/O System and File I/O.

Each entry has an argument denoting the particular I/O switch involved in the operation. For an entry that requires the I/O switches to be in the attached state, the description of the entry's function applies only when the switch is attached to a file or is attached to a device via the I/O module tty_. For the meaning of operations on a switch attached as a synonym, see the MPM Section, The Multics I/O System. For other attachments, see the write_up of the particular I/O module.

When an entry requires the I/O switch to be opened, and it is not open, the state of the switch is not changed, and the code error_table_$not_open is returned. If the I/O switch is open but not in one of the allowed opening modes, the state of the switch is not changed, and the code error_table_$no_operation is returned.

Operations pertaining to files reference four position designators: the next byte, the next record, the current record, and the key for insertion. Their use is explained in the MPM Section, File I/O.

Several operations involve a buffer. This is a block of storage provided by the caller of the operation as the target for input or the source for output. The only restriction on a buffer is that it be byte aligned, i.e., its bit address must be divisible by nine.

Note that the status code returned by an entry may not be a system status code in cases where the I/O switch is attached via a nonsystem I/O module.

Entry: iox_$attach_iocb

This operation attaches an I/O switch in accordance with a specified attach description. The form of an attach description is given in the MPM Section, the Multics I/O System. If the switch is not in the detached state, its state is not changed, and the code error_table_$not_detached is returned.

Usage

```
     declare iox_$attach_iocb entry (ptr, char(*), fixed (35));
     call iox_attach_iocb (iocb_ptr, atd, code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. atd   is the attach description. (Input)

3. code   is a status code (Output)

Entry iox_$attach_ioname

This operation is the same as iox_$attach_iocb except that the I/O switch is designated by name and a pointer to its control block is returned. The control block is created if it does not already exist.

Usage

```
     declare iox_$attach_ioname entry (char(*), ptr, char(*),
          fixed(35));
     call iox_$attach_iocb (switchname, iocb_ptr,atd,code);
```

1. switchname   is the name of the I/O switch. (Input)

2. iocb-ptr   points to the switch's control block. (Output)

3. atd   is the attach description. (input)

4. code   is a status code. (Output)

Entry: iox_$close

This operation closes an I/O switch. If the switch is not open, its state is not changed, and the code error_table_$not_open is returned.

Usage

```
     declare iox_$close entry (ptr,fixed (35));
     call iox_$close (iocb_ptr,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. code   is a status code. (Output)

Entry: iox_$control

This operation performs a specified control order on an I/O switch. The allowed control orders depend on the attachment of the switch. If a control order is not supported for a particular attachment, the code error_table_$no_operation is returned if the switch is open. In the case where the switch is closed, the code error_table_$not_open or error_table_$no_operation is returned, the latter code only by I/O modules that support orders with the switch closed. For details on control orders, see the write-up of the particular I/O module used in the attachment.

Usage

```
     declare iox_$control entry(ptr,char(*),ptr,fixed (35));
     call iox_$control(iocb_ptr,order,info_ptr,code);
```

1. iocb_ptr   points to the switch's control block (Input)

2. order   is the name of the control order. (Input)

3. info_ptr   is null or points to data whose form depends on the attachment. (Inout)

4. code   is a status code. (Output)

Entry: iox_$delete_record

This operation deletes the current record from the file to which an I/O switch is attached. The switch must be open for sequential_update, keyed_sequential_update, or direct_update. If the current record is null, the file's position is not changed, and the code error_table_$no_record is returned.

If the deletion takes place, the current record position is set to null. For keyed_sequential_update, the next record position is set to the record following the deleted record or to end of file (if there is no such record). For direct update, the next record position is set to null.

Usage

```
     declare iox_$delete_record entry (ptr,fixd(35));
     call iox_$delete_record (iocb_ptr, code);
```

1. iocb_ptr   points to the switch's control block. (input)

2. code   is a status code. (Output)

Entry: iox_$detach_iocb

This operation detaches an I/O switch. If the switch is already detached, its state is not changed, and the code error-table_$not_attached is returned. If the switch is open, its state is not changed, and the code error_table_$not_closed is returned.

Usage

```
      declare iox_$detach_iocb entry (ptr,fixed(35));
      call iox_$detach_iocb (iocb_ptr, code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. code   is a status code. (Output)

Entry: iox_$find_iocb

This entry returns a pointer to the control block for an I/O switch. The control block is created if it does not already exist.

Usage

```
     declare iox_$find_iocb entry (char(*),ptr,fixed (35));
     call iox$find_iocb (switchname,iocb_ptr,code);
```

1. switchname   is the name of the I/O switch. (Input)

2. iocb_ptr   points to the switch's control block. (Output)

3. code   is a status code. (Output)

Entry: iox_$get_chars

This operation reads 9-bit bytes from the unstructured file or device to which an I/O switch is attached. The switch must be open for stream_input or stream_input_output. The desired number of bytes, n, is specified in the call. Some I/O modules may actually read fewer than n bytes into the buffer, even though n bytes are available from the file or device. In this case the code error_table_$short_rcord is returned. When this code is returned, the caller may again call iox_$get_chars to get more bytes.

If the switch is attached to a file, bytes are read beginning with the next byte, and the next byte position designator is advanced by the number of bytes read, which may get it to end of file. If the next byte position is already at end of file, the code error_table_$end_of_info is returned.

Usage

```
declare iox_$get_chars entry (ptr,ptr,fixed(21), fixed(21),
     fixed(35));
call iox_$get_char_(iocb_ptr,buff_ptr,n,n_read,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. buff_ptr   points to the byte-aligned buffer into which bytes are to be read. (Input)

3. n   is the number of bytes to be read. It must be greater than or equal to zero. (Input)

4. n_read   is the number of bytes actually read. If code is zero, n_read equals n. (Output)

5. code   is a status code. (Output)

Entry: iox_$get_line

This operation reads 9-bit bytes from the unstructured file or device to which an I/O switch is attached. The switch must be open for stream_input or stream_input_output. Bytes are read until the input buffer is filled, a new line character is read, or end of file is reached, whichever occurs first. A code of zero is returned if and only if a new line character is read into the buffer (and it will be the last character read). If the input buffer is filled without reading a new line character, the code error_table_$long_record is returned.

If the switch is attached to a file, bytes are read beginning with the next byte, and the next byte position designator is advanced by the number of bytes read. If the end of file is reached without reading a new line character, the next byte position designator is set to end of file and the code error_table_$end_of_info is returned.

Usage

```
declare iox_$get_line entry (ptr,ptr,fixed (21),fixed(21),
     fixed(35));
call iox_$get_line (iocb_ptr,buff_ptr,buff_len,n_read,code):
call iox_$get_line (iocb_ptr,buff_ptr,buff_len,n_read,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. buff_ptr   points to a byte-aligned buffer. (Input)

3. buff_len   is the length of the buffer in bytes. (Input)

4. n_read   is the number of bytes read into the buffer. (Output)

5. code   is a status code. (Output)

Entry: iox_$modes

This operation is used to obtain or set modes that affect the subsequent behavior of an I/O switch. The switch must be attached via an I/O module that supports modes. If the switch is not attached, the code error_table_$not_attached is returned. If the switch is attached, but modes are not supported, the code error_table_$no_operation is returned for an open switch and the code error_table_$not_open is returned for a closed switch. If the switch is attached and modes are supported, but an invalid mode is given, the code error_table_$bad_mode is returned.

Each mode is a sequence of non blank characters. A mode string is a sequence of modes, seperated by commas and containg no blanks. For a list of valid modes, see the particular I/O module involved.

Usage

```
declare iox_$modes entry (ptr,cha(*), char(*), fixed (35));
call iox_$modes (iocb_ptr,new_modes,old_modes,code);
```

1. iocb_ptr — points to the switches control block. (Input)

2. new_modes — is the mode string containing the modes to be set. Other modes are not affected. If this argument is the null string no modes are changed. (Input)

3. old_modes — is the string of modes in force when the call is mode. If this argument has length zero, this information is not returned. (Output)

4. code — is a status code. (Output)

Entry: iox_$open

This operation opens an I/O switch. The switch must be attached via an I/O module that supports the specified opening mode, and it must be in the closed state. If the switch is not attached, its state is not changed, and the code error_table_$not_attached is returned. If the switch is already open, the code error_table_$not_closed is returned.

If the switch is attached to a file, the appropriate file postion designators are established, and an existing file may be replaced by an empty file. This replacement may be avoided by specifying extension of the file in the attach description or in the call to iox_$open. See the MPM Section, File I/O for full details.

Usage

```
declare iox_$open(ptr,fixed,bit(1) aligned, fixed(35));
call iox_$open (iocb_ptr,mode,ext,code);
```

1. iocb_ptr — points to the switch's control block. (Input)

2. mode — is the number of the mode as shown in table 1 in the MPM Section, The Multics I/O System, e.g. 1 for stream_input, 2 for stream_output. (Input)

3. ext — is "1"b to specify extension of a file and is "0"b otherwise. The value "1"b is allowed only

if the opening is for stream_output, stream_input_output, sequential_output, sequential_input_output, keyed_sequential_output, or direct_output. (Input)

4. code is a status code. (Output)

Entry: iox_$position

For an I/O switch attached to a file, this operation positions to the beginning or end of the file, or skips forwards or backwards over a specified number of lines (unstructured files) or records (structured files). For an I/O switch attached to a device, this operation reads and discards characters until a specified number of new line characters have been skipped.

The switch must be opened for stream_input, stream_input_output, sequential_input, sequential_input_output, sequential_update, keyed_sequential_input or keyed_sequential_update. In addition, for keyed openings, the next record position should not be null. If it is null, the code error_table_$no_record is returned.

Usage

```
declare iox_$position entry (ptr,fixed,fixed (21),fixed (35));
call iox_$position (iocb_ptr,type,n,code);
```

1. iocb_ptr points to the switch's control block. (Input)

2. type is -1 for positioning to the beginning of the file, +1 for positioning to the end of the file, or 0 for skipping new line characters or records. (Input)

3. n is the number of lines or records to be skipped (forward skip) or the negative of that number (backward skip). It may be zero. (Input)

4. code is a status code. (Output)

Notes

Positioning to the beginning of a non-empty file sets the next record position to be at the first record in the file (sequential and keyed_sequential openings) or sets the next byte postion to be at the first byte in the file (stream openings). Positioning to the end of a file, or to the beginning of an empty file, sets the relevant position designator to the end of the file position.

Successfully skipping records (sequential and keyed_sequential openings), moves the next record position forward or backwards by the specified number of records, n, provided that many record exist in the indicated direction. For example, suppose that when iox_$position is called, the next record is the mth record in the file, and n record are to be skipped. Then for a successful forwards skip, the file must contain at least (m+n-1) records, and the next record will be set to record (m+n) (if there are at leat m+n records in the file) or to end of file (if there are m+n-1 records in the file. For a successful backwards skip, m must be greater than n, and the next record position is set to record (m - n).

Successfully skipping forward over new-line characters (stream openings) advances the next byte position over the specified number, n, of new line characters, leaving it at the byte following the nth new line character or at end of file (if the nth new line character is the last byte in the file). Successfully skipping backwards over n new line characters moves the next byte position backwards to the nth preceding new line character and then moves it further backwards as far as is possible without encountering another new line character. The effect is to set the next byte position to the first character in a line.

If the relevant part of the file contains too few records or new line characters, the next record position or next byte position is set to the first record or byte (backwards skip with non empty file) or end of file (all other cases), and the code error_table_$end_of_info is returned.

When a call to iox_$position specifies skipping zero lines or records, the skip is successful, and the next record position is undisturbed.

In openings for update, the current record position is set as follows. If the positioning is a successful skip, and if there is a record preceding the resulting next record, the current record position is set to the immediately preceding record. In all other cases, the current record position is set to null.

In the case of keyed_sequential_update, the key for insertion is set to null.

Entry: iox_$put_chars

This operation writes a specified number of 9-bit bytes to the unstructured file or device to which and I/O switch is attached. The switch must be open for stream_output or stream_input_output.

In the case of a file, if the opening is for stream_output, the bytes are simply added at the end of the file. However, if the opening is for stream_input_output, and the next byte position is not at end of file, the file is first truncated so that the byte preceding the next byte becomes the last byte in the file. The bytes being written are then added at the end of the file, and the next byte position is set to end of file.

Usage

```
   declare iox_$put_chars entry (ptr,ptr,fixed(21),  fixed(35));
   call iox_$put_chars (iocb_ptr,buff_ptr, n, code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. buff_ptr   points to a byte-aligned buffer containing the bytes to be written. (Input)

3. n   is the number of bytes to be written. It must be >=0. (Input)

4. code   is a status code. (Output)

Entry: iox_$read_key

This operation returns both the key and length of the next record in an indexed file attached to an I/O switch. The switch must be open for keyed_sequential_input or keyed_sequential_update. If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned. The file position designators are not changed by this entry.

Usage

```
     declare iox_$read_key entry (ptr,char(256) varying,fixed(21),
               fixed(35));
     call iox_$read_key (iocb_ptr,key,rec_len,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. key   is the next record's key. (Output)

3. rec_len   is the next record's length in bytes. (Output)

4. code is a status code. (Output)

Entry: iox_$read_length

This operation returns the length of the next record in a structured file attached to an I/O switch. The switch must be open for sequential_input, sequential_input_output, sequential_update, keyed_sequential_input, keyed_sequential_update, direct_input, or direct_update. If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned.

Usage

```
     declare iox_$read_length entry (ptr,fixed(21),fixed(35));
     call iox_$read_length (iocb_ptr,rec_len,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. rec_len   is the next tecords length in bytes. (Output)

3. code   is a status code. (Output)

Entry: iox_$read_record

This operation reads the next record in a structured file to which an I/O switch is attached. The switch must be open for sequential_input, sequential_input_output,sequential_update, keyed_sequential_input, keyed_sequential_update, direct_input, or direct_update. The read is successful if the next record position is at a record. If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned.

In sequential and keyed_sequential openings, a successful read advances the next record position by one record; an unsuccessful read leaves it at the end of file or null. In direct openings, this operation always sets the next reciord position to null. In openings for keyed_sequential_ update and direct_update, a successful read sets the current record position to the record just read; an unsuccessful read, sets it to null. In openings for keyed_sequential_update and direct_update, the key for insertion is always set to null.

If the record is too long for the specified buffer, the first part of the record is read into the buffer, and the code error_table_$long_record is returned. As far as setting position indicators is concerned, this is considered a successful read.

<u>Usage</u>

```
     declare iox_$read_record entry (ptr,ptr,fixed(21),fixed(21),
          code);
     call iox_$read (iocb_ptr,buff_ptr,buff_len,rec_len,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. buff_ptr   points to a byte-aligned buffer, into which the record is to be read. (Input)

3. buff_len   is the length of the buffer in bytes. (Input)

4. rec_len   is the length of the record in bytes. (Output)

5. code   is a status code. (Output)

<u>Entry</u>: iox_$rewrite_record

This operation replaces the current record in a structured file to which an I/O switch is attached. The switch must be open for sequential_update, keyed_sequential_update, or direct_update. If the current record position is null, the code error_table_$no_record is returned.

For keyed_sequential_update, this operation sets the next record position to the record immediately following the current record or to end of file (if no such record exists). (Note that the next record position may already be at this point). For direct_update, the next record position is set to null. No other changes are made to the position designators.

Usage

```
     declare iox_$rewrite_record(iocb_ptr,buff_ptr,rec_len,code);

1. iocb_ptr      points to the switch's control block. (Input)

2. buff_ptr      points to a byte aligned buffer containing the new
                 record. (Input)

3. rec_len       is the length of the new record. (Input)

4. code          is a status code. (Output)
```

Entry: iox_$seek_key

This operation searches for the record with a given key in an indexed file to which an I/O switch is attached. It also serves to define the key for a record to be added by a following write_record operation. The switch must be open for keyed_sequential_input, keyed_sequential_output, keyed_sequential_update, direct_input, direct_output, or direct_update.

For keyed_sequential_output, the given key should be greater (according to the rules for characters string comparision!) than the key of the last record in the file. If it is, the code error_table_$no_record is returned, and the key for insertion is set to the given key. Otherwise the code error_table_$key_order is returned, and the key for insertion is set to null.

For other openings, this operation works as follows:

If the file contains a record with the given key, a code of zero is returned. the records length is returned, the next reocrd position and current record position are set to the record, and the key for insertion is set to null. (Not all of these position designators are applicable in all openings).

If the file does not contain a record with the given key, the code error_table_$no_record is returned, the next record postiion and current record positior are set to null, and the key for insertion is set to the given key. (Not all of these positior designators are applicable in all openings).

<u>Usage</u>

```
declare iox_$seek_key entry (ptr,char(256) varying,
     fixed(21), fixed(35));
call iox_$seek_key (iocb_ptr, key,rec_len,code);
```

1. iocb_ptr   points to the switch's control block. (Input)

2. key   contains the given key. All trailing blanks are removed from key to obtain the given key, and the result may be the null string. (Input)

3. rec_len   is the length in bytes of the record with the given key (Output)

4. code   is a status code. (Output)

<u>Entry</u>: iox_$write_record

This operation adds a record to a structured file to which an I/O switch is attached. The switch must be open for sequential_output, sequential_input_output, keyed_sequential_output, keyed_sequential_update, direct_output, or direct_update.

If the switch is open for sequential output, the record is added at the end of the file. If the switch is open for input_output, and the next record position is not at the end of the file, the file is truncated so that the record preceding the next record becomes the last record in the file. The new record is then added at the end of the file.

If the switch is open for keyed_sequential_output,keyed_sequential_update, direct_output, or direct_update, the key for insertion designator should designate a key. If it does not, the code error_table_$no_key is returned, and nothing is changed. If there is a key for insertion, the new record is added to the file with that key, and the key for insertion is set to null. For keyed_sequential_update, the next record position is set to the record immediately following the new record or to end of file (if there is no such record). For keyed_sequential_update and direct_update, the current record position is set to the new record.

Usage

```
declare iox_$write_record entry (ptr,ptr,fixed(21),code);
call iox_$write_record (iocb_ptr,buff_ptr,rec_len, code);
```

1. iocb_ptr points to the switch's control block. (Input)
2. buff_ptr points to a byte-aligned buffer containing the new record. (Input)
3. rec_len is the length of the new record in bytes. (Input)
4. code is a status code. (Output)

I/O MODULE

Name: discard_

This I/O module provides a sink for output. It supports output operations, but the operations have no effect.

Entries in the module are not called directly by users; rather the module is accessed through the I/O system. See the MPM Section, the Multics I/O System, for a general description of the I/O System.

Attach Description

The attach description has the following form:

    discard_

No options are allowed.

Opening

This module opening modes supported are: stream_output_, sequential_output, keyed_sequential_output, and direct_output.

Control Operation

This module supports the conrol operation when the opening is for stream output. All orders are accepted; but they have no effect.

Modes Operations

This module supports modes operation when the opening is for stream_output. It always returns a null string for the old modes.

Seek Key Operation

When the opening is for keyed_sequential_output or direct_output, the seek_key operation returns the code error_table_$no_record.

I/O MODULE

Name: ntape_

This I/O module supports I/O from/to files on magnetic tape

Entries in the module are not called directly by users; rather, the module is accessed through the I/O system. See the MPM Section, the Multics I/O System, for a general description of the I/O System, and see the MPM Section, File I/O, for a discussion of files.

Attach Description

The attach_description has the following form:

```
ntape_ reelnum -opti- -optn-
```

1. reelnum — is the tape reel number. If the tape is 7-track, reelnum must contain "7-track." If the tape is 9-track, reelnum may contain with "9-track".

2. opti — may be one of the following options. An option may only occur once and "-raw" must occur.

-raw — means that each physical record (block) on the tape represents one logical record.

-write — means that the tape is to be mounted with a write ring. This option must occur if the I/O switch is to be opened for output or input-output.

-extend — specifies extension of the file if it already exists on the tape.

Opening

The opening modes supported are sequential_input, sequnential_output, and sequential_input_output. If an I/O switch attached via ntape_ is to be opened for output or input_output, the option "-write" must occur in the attach description.

Control Operation

This I/O module does not support the control operation.

Modes Operation

This I/O module does not support the modes operation.

Note

With a "-raw" attachment the relation between logical and physical record is as follows. On input the logical record contains m=4*ceil(n/36) bytes, where n is the nnumber of data bits in the physical reord. The first n bits of the input record are the data bits, the last (9+m-n)bits are zero. On output the physical record contains n=k*ceil((36*ceil(m/4))/k) data bits, where k+1 is the number of tracks on the tape, and m is the length of the logical rrecord. The first 9+m data bits of the physical record contain the bits of the logical record (i.e. The output buffer). The last (n-9+m) bits of the physical record are zero.

I/O MODULE

Name: syn_

Attaching an I/O switch, x, via this I/O module establishes the switch as a synonym for another switch, y. Thereafter, performing an operation other than attach or detach on x has the same effect as performing it on y. There is one exception to this: if the attach_description specifies that an operation is to be inhibited, performing that operation on x results in an error code.

Entries in the module are not called directly by users: rather the module is accessed through the I/O system. See the MPM Section, the Multics I/O system, for a general description of the I/O system and a discussion of synonym attachments.

Attach Descriptions

syn_ switchname2 -inhib- -opname1- -opname*n*-

1) switchname2 — is the name of the I/O switch, y, for which the attached switch, x, is to be a synonym.

2) inhib — may be "-inh" or "-inhibit".

3) opname*i* — is the name of an I/O operation to be inhibited. If opname*1*...opname*n* occur they must be preceded by "-inhibit" or "-inh". The opname*i* must be from the following list: "open", "close", "get_line", "put_chars", "read_record", "write_record", "rewrite_record", "delete_record", "read_length", "position", "seek_key", "read_key", "close", "control", and "modes".

Detach Operation

The detach operation detaches the switch x (the switch attached via syn_). It has no effect on the switch y for which x is a synonym.

Inhibited Operations

An inhibited operation returns the code error_table_$no_operation.

I/O MODULE

Name: tty_

This I/O module supports I/O from/to devices that can be operated in a typewriter like manner, e.g. the user's console.

Entries in the module are not called directly by users; rather the module is accessed through the I/O system. See the MPM section, The Multics I/O System, for a general description of the I/O system.

Attach Description

The attach description has the form:

tty_ device

1) device identifies the particular device. Normally the user is only interested in his own console, and this is attached when the process is initialized.

Opening

The opening modes supported are: stream_input, stream_output, and stream_input_output.

Editing

On both input and output, data is automatically edited as described in the MPM section, Typing Conventions. To control the editing, use the modes operation. Details on the various modes are given below.

Buffering

In general, this I/O module reads input data into an intermediate buffer as the device makes it available. The operations get_line and get_chars get the data from the buffer later. Similarly output data is stored in a buffer and then transmitted to the device. This allows the process to proceed without waiting for the device.

The amount of buffering is unpredictable. To flush the buffers, use the control operation with the order resetread, resetwrite, or abort.

Interrupted Operations

When an I/O operation, except detach, being performed on a switch attached by this I/O module is interrupted by a signal, other operations may be performed on the switch during the interruption. The effect, as seen by the user, is that the interrupted operation is completely performed before the interruption or is not started until after the interruption.

Control Operation

The following orders are supported when the I/O switch is open. Except as noted, the info ptr should be null.

abort — flushes the input and output buffers.

resetread — flushes the input buffer.

resetwrite — flushes the output buffer.

hangup — causes the telephone line connection of the terminal to be disconnected, if possible.

listen — cause a wakeup to be sent to the process if the line associated with this device ID is dialed up.

info — causes information about the device to be returned. The infoptr should point to the following structure that is filled in by the call.

```
    declare 1 info_structure aligned,
              2 id char(4) unaligned,
              2 reserved char(8) unaligned,
              2 tw_type fixed bin;
```

1) id — is the identifier of the specific device as told to Multics by the device when the device is initialized.

2) reserved — is space reserved for compatibility purposes.

3) tw_type — identifies the type of device:

1 = device similar to IBM 1050;
2 = device similar to IBM 2741 (with M.I.T. modifications);
3 = device similar to Teletype model 37
device without answerback or device with
unrecognized answerback;

4 = device similar to Terminet 300;
5 = device similar to ARDS;
6 = device similar to IBM 2741 (standard);
7 = device similar to Teletype models 33 or 35;
8 = device similar to Teletype model 38.

quit_enable

causes quit processing to be enabled for this device. (Quit processing is initially disabled.)

quit_disable

causes quit processing to be disabled for this device.

start

causes a wakeup to be signalled on the event channel associated with this device. This request is used to restart processing on a device whose wakeup may have been lost or discarded.

printer_off

causes the printer mechanism of the console to be temporarily disabled if it is physically possible for the terminal to do so.

printer_on

causes the printer mechanism of the terminal to be reenabled.

Modes Operation

The modes operation is supported when the I/O switch is open. The recognized modes are listed below. Some modes have a complement indicated by the character "^" (e.g. "^erkl") that turns the mode off. For these modes the complement is displayed along with the mode.

erkl, ^erkl

specifies that erase-and-kill processing is to be performed on input. (Default is on.)

can, ^can

indicates that standard canonicalization is to be performed. (Default is on.)

rawi, ^rawi

indicates that the data specifies is to be read from the device directly without any conversion or processing. (Default is off.)

rawo, ^rawo

indicates that data is to be written to the device directly without any conversion or processing. (Default is off.)

tabs, ^tabs — indicates that tabs are to be inserted in output in place of spaces when appropriate. (Default is off for model 33, 35 and 38 teletypes; default is on for all other terminal types).

edited, ^edited — causes printing of characters for which there is no defined Multics equivalent on the device referenced to be suppressed. If edited mode is off, the 9-bit octal representaiton of the character is printed. (Default is off.)

esc, ^esc — enables escape processing (see the MPM Reference Guide section, Typing Conventions) on all input read from the device. (Default is on.)

red, ^red — specifies that red and black shifts are to be spent to the terminal. (Default is off for devices similar to terminet 300s and for all terminals without an answerback identifier; default is on for all other terminals.)

crecho, ^crecho — specifies that a carriage return is to be echoed when a line feed is typed. (Default is off; this mode is only functional with devices similar to model 33, 35, 37 and 38 Teletypes or to Terminet 300s.

ll<u>n</u> — specifies the length in character positions of a console line. If an attempt is made to output a line longer than this length, the excess characters are placed on the next line. (Default line length is 130 for devices similar to IBM 1050s, 125 for IBM 2741s, 88 for TTY37s, 118 for Terminet 300s, for ARDS, 74 for TTY33s and TTY35s, and 125 for TTY38s.)

pl<u>n</u> — specifies the length in lines of a page. When an attempt is made to exceed this length, an ARDS "DEL" character is printed; when the user types an erase character, the output continues with the next page. This mode is functional only for ARDS-like terminals. (Default page length is 50 for ARDS-like terminals.)

hndlquit, ^hndlquit

specifies that when a quit is detected, a new line character is echoed and a resetread of the associated stream is performed. (Default is on.)

default

is a shorthand for erkl, can, ^rawi, ^rawo, and esc. The settings for other modes are not affected.

I/O MODULE

Name: vfile_

This I/O module supports I/O from/to files in the storage system. All logical file types are supported.

Entries in this module are not called directly by users; rather, the module is accessed through the I/O system. See the MPM section, the Multics I/O system, for a general description of the I/O system, and see the MPM section, file I/O, for a discussion of files.

Attach Description

The attach description has the following form:

```
vfile_  pathname --extend-
```

1) pathname is the absolute or relative pathname of the file.

2) -extend specifies extension of the file if it already exists.

To form the attach description actually used in the attachment, the pathname is expanded to obtain an absolute pathname.

Opening

All opening modes are supported. If the opening is for input only, "r" access is required on the file. In all other cases "rw" access is required.

Rewrite Operation

If the file is a sequential file, the new record must be the same length as the replaced record. If not, the code returned is error_table_$long_record or error_table_$short_record.

Delete Operation

If the file is a sequential file, the record is logically deleted, but the space it occupies is not recovered.

Control Operation

This operation is not supported.

Modes Operation

This operation is not supported.

Status Codes

Two status codes are of special importance. The code error_table_$file_busy (console message: "File already busy for other I/O activity") is returned by open if the open routine detects that another I/O switch (in any process) is already open for output, input_output, or update on this file. It is also returned by some operations if they detect multiple simultaneous uses of a single switch, e.g., as a result of trying to perform an I/O operation after quitting out of an I/O operation. Note that the vfile_ module does not detect all problems of this sort, and that an undetected problem may result in destroying the contents of the file.

The code error_table_$bad_file (console message: "File is not a structured file or is inconsistent") may be returned by operations on structured files. It means that an inconsistency has been detected in the file. Possible causes are:

1) The file is not a structured file of the required type.

2) Interruption of some operation on the file left it in an inconsistent state, and the vfile_ module cannot make it consistent.

3) A program accidentally modified some words in the file.

In the last two cases an earlier consistent copy of the file should be reloaded.

SWG SECTION 3.7

THE I/O CONTROL BLOCK

Each I/O switch has an associated I/O control block. The I/O system creates a control block for a switch the first time a call to iox_$find_iocb requests a pointer to the control block. The control block remains in existence for the life of the process unless explicitly destroyed by a call to iox_$destroy_iocb.

The principal components of an I/O control block are pointer variables and entry variables whose values describe the attachment and opening of the I/O switch. There is one entry variable for each I/O operation except attach. To perform an I/O operation through the switch the corresponding entry value in the control block is called. For example, if iocbptr is a pointer to an I/O control block, the call

```
     call iox_$put_chars (iocbptr,buffptr,bufflen,code);
results in the call

     call iocbptr->iocb.put_chars(iocbptr,buffptr,bufflen,code);
```

Some system routines make the latter call directly, without going through iox_; but all other routines must call iox_.

Structure of the I/O Control Block

The following declaration describes the first part of an I/O control block. Only a few I/O system programs use the remainder of the I/O control block, and only these programs declare the entire I/O control block. All discussion of I/O control blocks in this Subsystem Writers Guide refers only to the first part of the control block. Thus the statement "no other changes are made to the control block" means that no other changes are made to the first part of the control block. The I/O system might make changes to the remainder to the block, but these are only of interest to the I/O system.

Comments in the declaration briefly describe the various components. The significance of the pointers is explained in more detail in later paragraphs. For full detail on the entry variables see the MPM write-ups of the corresponding entries in iox_.

```
dcl  1 iocb aligned,
     2 iocb_version fixed init (1), /* = 1 */
     2 name char (32), /* Name of the I/O switch. */
     2 actual_iocb_ptr ptr, /* ptr to the actual iocb. */
     2 attach_descrip_ptr ptr,/* Ptr to attach description. */
     2 attach_data_ptr ptr, /* Ptr to attach data structure. */
     2 open_descrip_ptr ptr,/* Ptr to open description. */
     2 open_data_ptr ptr,/* Ptr to open data structure. */
     2 reserved bit (72), /* Reserved for future use. */
     2 detach_iocb entry (ptr,fixed(35)),/* detach_iocb(p,code)*/
     2 open entry (ptr,fixed,bit(1)aligned,fixed(35)),
       /*open (p,mode,extend,code)*/
     2 close entry (ptr,fixed (35)),/* close (p, code) */
     2 get_line entry (ptr,ptr,fixed (21),fixed(21),fixed(35)),/*
       get_line (p,buffptr,bufflen,actlen,);
     2 get_chars entry (ptr,ptr,fixed (21)fixed(21),fixed(35)),/*
       get_chars(p,buffptr,bufflen,code) */
     2 put_chars entry (ptr,ptr,fixed(21)fixed(35)),/* put_chars
       (p,buffptr,bufflen,code) */
     2 modes entry(ptr,char(*),char(*),fixed(35)),/*modes(p,newmode,
       oldmode,code)*/
     2 position entry (ptr,fixed,fixed(21),fixed(35)),/*position(p,
       type,n,code) */
     2 control entry(ptr,char(*),ptr,fixed(35)),/*
       control (p,order,infaptr,code)*/
     2 read_record entry (ptr,ptr,fixed(21),fixed(21),fixed(35)),/*
       read_record (p,buffptr,bufflen,rec_len,code)*/
     2 write_record entry(ptr,ptr,fixed(21),fixed (35)),/*
       write_record(p,buffptr,bufflen,code)*/
     2 rewrite_record entry (ptr,ptr,fixed(21),fixed(35)),
       /*rewrite_record (p,buffptr,bufflen,code) */
       (p, buffptr,bufflen,code) */
     2 delete_record entry(ptr,fixed(35)),/*delete_record (p,code)*/
     2 seek_key entry(ptr,char(256)varying,fixed(21),fixed (35)),/*
       seek_key (p,key,len,code)*/
     2 read_key entry (ptr,char(256)varying,fixed (21),fixed(35)),/*
       read_key (p,key,len,code)*/
     2 read_length entry (ptr,fixed(21),fixed(35));/*read_length
       (p,len,code)*/
```

Attach Pointers

If the I/O switch is detached, the value of iocb.attach_descrip_ptr is null. If the I/O switch is attached, the value is a pointer to a structure of the following form:

```
dcl 1 attach_descrip based,
      2 length fixed (17),
      2 string char (  refer (length));
```

The value of the variable attach_descrip.string is the attach description. See the MPM Section, the Multics I/O System, for details on the attach description.

If the I/O switch is detached, the value of iocb.attach_data_ptr is null. If the I/O switch is attached, the value may be null, or it may be a pointer to data used by the I/O module that attached the switch.

To test if the I/O switch is attached, test the value of iocb.attach_descrip_ptr.

Open Pointers

If the I/O switch is closed (whether attached or detached), the value of iocb.open_descrip_ptr is null. If the switch is open, the value is a pointer to a structure of the following form:

```
dcl 1 open_descrip based,
     2 length fixed (17),
     2 string char (  refer(length));
```

The value of the variable open_descrip.string is the open description. It has the following form:

```
     mode -ext- -info-
```

1. mode is one of the opening modes listed in Table 1, MPM Section, The Multics I/O System. (e.g. "stream_input").
2. ext is "-extend" if the opening specified file extension.
3. info may be other information about the opening.

If ext or info occurs in the string it is preceded by one blank. If both occur, each is preceded by one blank and ext occurs before info.

If the I/O switch is closed, the value of iocb.open_data_ptr is null. If the I/O switch is open, the value may be null, or it may be a pointer to data used by the I/O module that opened the switch.

Entry Variables

The entry variables in an iocb always have a value that is an entry point of an exernal procedure. When the I/O switch is in a state that supports a particular operation, the value of the corresponding entry variable is a routine that will perform the operation. When the I/O switch is in a state that does not support the operation, the value of the entry variable is a routine that will return the error code specified in the description of the corresponding entry point in the iox_.

Synonyms

When a I/O switch, x, is attached as a synonym for an I/O switch, y. The values of all entry variables in the iocb for x will be the same as those in the iocb for y with the exception of iocb.detach. Thus a call

```
     call iocbx_ptr->iocb.op(iocbx_ptr,...);
```

immediately goes to the correct routine.

The values of iocb.open_descrip_ptr and iocb.open_data_ptr for x are also the same as those for y. Thus the I/O routine has access to its open data (if any) through the iocb pointed to by iocbx_ptr.

The value of iocb.actual_iocb_ptr for x is a pointer to the control block for the switch that is the ultimate target of a chain of synonyms. (When the switch x is not attached as synonym, this pointer points to the control block for x itself). I/O modules use this pointer to access the ultimate I/O control block whose contents are to be changed, e.g., when a switch is opened. The I/O system then propogates the changes to other control blocks as required by synonym attachments.

SWG SECTION 3.8

WRITING AN I/O MODULE

This section contains information pertaining to the design and programming of an I/O module. In particular, it sets forth conventions that must be followed if the I/O module is to work correctly with the I/O system. The reader should be familiar with the MPM Sections, the Multics I/O System and File I/O, the MPM write-up of iox_, and the Subsystem Writers Guide Section, The I/O Control Block.

Possible I/O Modules

The following list indicates some of the possibilities for I/O modules.

1. Psuedo Device or File. An I/O module might simulate I/O to/from a device or file. For example, it might provide a sequence of random numbers in response to input request. The system I/O module discard_ is a trivial example of this sort of module.

2. A New File Type. An I/O module might support a new type of file in the storage system, perhaps a file in which records have multiple keys.

3. Reinterpreting a File. An I/O module might place a new structure (relative to the standard file types) on a standard type of file. For example, an unstructured file might be interpreted as a sequential file by considering 80 characters to be a record.

4. Monitoring a Switch. An I/O module might pass operations along to another module while monitoring them in some way, e.g. by copying input data to a file.

5. Unusual Devices. Working through the tty_ I/O module (in the raw mode), another I/O module might transmit data from/to a device that is not of a standard Multics device type (in regards character codes, etc.).

The last three examples involve a rather common arrangement. The user attaches an I/O switch, x, using an I/O module, A. To implement the attachmment, module A attaches another switch, y, using another I/O module B. When the user calls module A through the switch x, module A in turn calls module B through the switch y. Any nonsystemm I/O module that performms true I/O will work in this way, because it (or some module it calls) must call a system I/O module. There are system I/O routines that are more primitive than the I/O modules, but user written I/O modules must not call these routines.

General Design Considerations

Before programming an I/O module, one should develop clear specifications as to what it is supposed to do. In particular, one should list the opening modes to be supported and then consider the meaning of each I/O operation supported for those opening modes. (Table 1 in MPM Section, The Multics I/O System). The specifications in the write-up of iox_ must be related to the particular I/O module, e.g., what should seek_key mean for the I/O module discard_?

An I/O module contains routines to perform attach, open, close, detach and the operations allowed by the supported opening modes. Typically, all routines are in one object segment, but this is not required. If the module is a bound segment, only the attach entry need be retained as an external entry. The other routines are accessed only through entry variables in I/O control blocks.

An I/O module may have several open routines, several get_line routines, etc. to handle different situations, e.g. one get_line routine for stream_input openings, another for stream_input_output openings. Whenever the situation changes (e,g, at opening), the module stores the appropriate entry values in the I/O control block.

Rules

The following rules apply to the implementation of all I/O operations. Rules for particular operations are given later. In the rules, iocb is a based variable declared as in the Subsystem Writers' Guide Section, The I/O Control Block, and iocb_ptr is an argument of the operation in question.

1. Except for attach, the usage (entry declaration and parameters) of a routine that implements an I/O operation is the same as the usage of the corresponding entry in iox_. See the MPM write-up of iox_ for details.

2. Except for attach and detach, the actual I/O control block to which an operation applies (i.e. the control block attached by the called I/O module) must be referenced using the value of iocb_ptr→iocb.actual_iocb_ptr. It is incorrect to simply use iocb_ptr, and it is incorrect to remember the location of the control block from a previous call (e.g. by storing it in a data structure pointed to by iocb.open_data_ptr).

3. The value of iocb_ptr→iocb.open_data_ptr always equals the value of iocb_ptr→iocb.actual_iocb_ptr→iocb.open_data_ptr, and the value of ptr→iocb.open_descrip_ptr always equals the value of iocb_ptr_iocb.actual_iocb_ptr_iocb.open_descrip_ptr. Thus the data structures related to an opening may be accessed without going through iocb.actual_iocb_ptr.

4. If an I/O operation changes any values in an I/O control block, it must be the actual I/O control block (Rule 1); and, before returning, the operation must execute the call

```
     call iox_$propagate (p);
```

where p points to the changed control block. The routine iox_$propagate reflects changes to other control blocks attached as synonyms. It also makes certain adjustments to the entry variables in the control block when the I/O switch is attached, opened, closed, or detached.

5. All I/O operations must be external procedures.

Attach

The name of the attach routine is the concatenation of the name of the I/O module and "attach", e.g. "discard_attach" for the I/O module discard_. The routine has the following usage

```
  declare moduleattach entry (ptr,(*)char(*),bit(1)aligned,
       fixed(35));

  call moduleattach (iocb_ptr,option_array,com_err_switch,code);
```

1. iocb_ptr — points to the control block of the I/O switch to be attached. (Input).

2. option_array — contains the options in the attach description. If there are no options, its bounds are (0:0). Otherwise, its bounds are (1:n) where n is the nummber of options. (Input).

3. com_err_switch — is "1" b if the attach routine should call com_err_ in cases where an error is detected. It is "0" b if com_err_ should not be called. (Input).

4. code — is a system status code (Output).

The following rules apply to coding an attach routine.

1. If the I/O switch is already attached (i.e., if iocb_ptr->iocb.attach_descrip_ptr is not equal to null), return the error code error_table_$not_detached; do not make the attachmment.

2. If, for any reason, the switch cannot be attached, return an appropriate non-zero error code; do not modify the control block. Call com_err_ if and only if com_err_ switch is "1"b. If the attachment can be made, follow the remaining rules and return with code = 0.

3. Set iocb_ptr->iocb.open and iocb_ptr->iocb.detach_iocb to the appropriate open and detach routines. Set iocb_ptr->attach_descrip_ptr to point to a structure as described in the Subsystem Writers' Guide Section, the I/O Control Block. Note that the attach description in this structure must be fabricated from the options in the argument option array, and there may be some modification of options, e.g., expanding a pathname.

4. You may set iocb_ptr->iocb.attach_data_ptr, iocb_ptr->iocb.modes, and iocb_ptr->iocb.control. Make no other modifications to the control block.

Open

An open routine will only be called when the actual I/O switch is attached (via the module containing the routine) but not open. The following rules apply to coding an open routine.

1. If, for any reason, the opening cannot be performed, return an appropriate error code; do not modify the I/O control block. If the opening can be performed, follow the remaining rules, and return with code = 0.

2. Set iocb_ptr->iocb.actual_iocb_ptr->iocb.op to an appropriate routine. This applies for each op that is allowed for the specified opening mode (Table 1 in the MPM Section, the Multics I/O system).

3. If either the modes operation or the control operation is enabled with the switch closed but not with it open, set iocb_ptr->iocb.actual_ iocb_ptr->iocb.op (op = modes or control) to iox_$err_no_operation.

4. Set open_descrip_ptr to point to a structure as described in the Subsystem Writers' Guide Section, the I/O Control Block.

5. You may set iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr. Do not make any other modifications to the control block.

Close

A close routine will only be called when the actual I/O switch is open, the opening having been made by the I/O module containing the close routine. The following rules apply to coding a close routine.

1. Set iocb_ptr->iocb.actual_iocb_ptr->iocb.open and iocb_ptr->iocb.actual_ iocb_ptr->iocb.detach_iocb to the appropriate open and detach routines. Set iocb_ptr->iocb.actual_iocb_ptr->iocb.open_descrip_ptr to null.

2. If either the modes operation or the control operation is enabled with the switch open, set iocb_ptr->iocb.actual_iocb_ptr->iocb.op, where op is modes or control. Unless the operation is enabled with the switch closed, set the entry variable to iox_$err_no_operation.

3. Do not make any other modifications to the control block.

4. The close routine must not return without closing the switch.

Detach

A detach routine will only be called when the actual I/O switch is attached but not open, the attachment having been made by the I/O module containing the detach routine. The following rules apply to coding detach routines.

1. Set iocb_ptr->iocb.attach_descrip_ptr to null.

2. Do not make any other modifications to the control block.

3. The detach routine must not return without detaching the switch.

Modes and Control

These operations may be accepted with the I/O switch attached but closed; but it is generally better practice to accept them only when the switch is open.

If the control operation is supported, it must return the code error_table_$no_operation when given an invalid order. In this situation the state of the I/O switch must not be changed.

If the modes operation is supported, it must return the code error_table_$bad_mode, when given an invalid mode.

Other Operations

Routines for the other operations will only be called when the actual I/O switch is attached and open in a mode for which the operation is allowed, the opening and attachment having been made by the I/O module containing the routine. In coding these routines, you may make only the following modifications to the I/O control block of the actual I/O switch.

1. Resetting iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr.

2. Resetting an entry variable set by the open routine, e.g., to switch from one put_chars routine to another.

3. Closing the switch in an error situation. In this case the rules above for close must be followed.

SWG SUBROUTINE

Name: iox_

This procedure performs I/O operations and some related functions. The following entry points are documented in the MPM.

iox_$attach_iocb
iox_$attach_ioname
iox_$close
iox_$control
iox_$delete_record
iox_$detach_iocb
iox_$find_iocb
iox_$get_chars
iox_$get_line
iox_$modes
iox_$open
iox_$position
iox_$put_chars
iox_$read_key
iox_$read_key
iox_$read_length
iox_$read_record
iox_$rewrite_record
iox_$seek_key
iox_$write_record

For a general description of the I/O system see the MPM section, the Multics I/O System. For information regarding the use of iox_ in writing an I/O module, see the Subsystem Writer's Guide Section, Writing an I/O Module.

Entry: iox_$destroy_iocb

This entry frees the storage used by the control block for an I/O switch. The switch must be in the detached state. Any existing pointers to the control block become invalid.

Usage

```
declare iox_$destroy_iocb entry (ptr, fixed(35));

call iox_$destroy_iocb (iocb_ptr, code);
```

1) iocb_ptr points to the I/O control block to be freed. (Input)

2) code is a system status code. (Output)

Entry: iox_$err_no_operation

This entry accepts any number of arguments, the last of which is fixed(35). It sets the last argument to the code error_table_$no_operation. This entry is only called through entry variables in an I/O control block. See the Subsystem Writers Guide Section, Writing an I/O Module, for instructions on when to assign this entry to such an entry variable.

Usage

declare iox_$err_no_operation entry;

Entry: iox_$find_iocb_n

This entry may be used to find all existing I/O control blocks, whether attached or detached. It returns a pointer to the nth control block in the calling ring, the numbering being arbitrary. If there are fewer than n control blocks, a null pointer and the code error_table_$no_iocb are returned.

Usage

declare iox_$find_iocb_n entry (fixed, ptr, fixed(35));

call iox_$find_iocb_n (n, iocb_ptr, code);

1) n is the number of the I/O control block. (input)

2) iocb_ptr is a pointer to the control block. (Output)

3) code is a system status code. (Output)

Entry: iox_$look_iocb

This entry returns a pointer to the control block for a specified I/O switch. If the control block does not exist, it is not created (in contrast to ios_$find_iocb), and a null pointer and the code error_table_$no_iocb are returned. Creating or destroying control blocks during a sequence of calls to this entry should be avoided, as it causes unpredictable changes to the numbering.

Usage

declare iox_$look_iocb entry (char(*), ptr, fixed(35));

call iox_$look_iocb (switchname, iocb_ptr, code);

1) switchname is the name of the I/O switch. (Input)

2) iocb_ptr is a pointer to the control block. (output)

3) code is a system status code. (Output)

Entry: iox_$propagate

This entry adjusts certain pointers and entry variables in an I/O control block as required in changing between the states detached, attached-closed, attached-open. It also reflects modifications to a control block to other control blocks that are synonyms (immediate or chained) for it. This entry must be called at certain points in the code of an I/O module, and it must not be called in any other crcumstances. See the Subsystem Writer's Guide Section, Writing an I/O Module, for instructions on when to call iox_$propagate.

Usage

```
     declare iox_$propagate entry (ptr);

     call iox_$propagate (iocb_ptr);
```

1) iocb_ptr is a pointer to the control block. (Input)

SWS_COMMAND

Name: io_call, io

This command performs an operation on a designated I/O switch.

Usage

```
     io_call opname switchname
```

1) opname designates the operation to be performed.

2) switchname is the name of the I/O switch.

Notes

Usage for each operation is shown below. For other operations see the MPM writeup of io_call. for full details on the operations, see the writeup of the subroutine iox_.

Usage: look_iocb

```
     io_call look_iocb switchname
```

This command prints the location of the I/O switch's control block if it exists. If the control block does not exist, it is not created.

Usage: destroy_iocb

```
     io_call destroy_iocb switchname
```

This command frees the storage used by the control block for the I/O switch. The switch must be in the detached state. Any existing pointers to the control block become invalid.

Usage: print_iocb, piocb

```
     io_call print_iocb switchname
```

This command prints the contents of the I/O switch's control block. If the control block does not exist, it is not created.