MULTICS STAFF BULLETIN - 113

To: Distribution

From: M.D. MacLaren, M.G. Smith

Date: August 21, 1973

Subject: Improvements to I/O System

The new Multics customers will make much heavier use of conventional (e.g. Fortran-, Cobol-, PL/I-type) i/o. This will be true even though much of the i/o will actually be to/from files in the permanent storage system. To support this expanded use of i/o, two major improvements are proposed for the Multics i/o system:

1. The current file manager will be replaced by a set of procedures that support indexed sequential files and sequential record files. The sequential record files will be supported on demountable storage devices (e.g. tapes) as well as in the permanent storage system. Performance will be greatly improved.

2. The i/o switch will be changed to improve performance, to support a more useful set of i/o calls (including record i/o calls), and to allow sharing of streams across rings. Peaceful coexistence is possible; the old ios_ entries will be supported locally along with the new switch features. However, the Multics product will eventually use only the new switch.

Various improvements to Fortran and PL/I i/o will also be made. In particular, the same efficient form of sequential record i/o will be used in both languages, and the performance of PL/I stream i/o will be improved to the point where a simple put-statement should be the most efficient way to print a message.

This bulletin describes the proposed changes in some detail. For the writer's convenience the new i/o system is described using the present tense. However, the intent is to have a careful review by all concerned before implementation of the changes. Comments should be sent to M.D. MacLaren at CISL (575 Tech Sq.), or by Multics mail to MacLaren Multics, by Sept 10, 1973.

The remainder of the bulletin is organized as follows.

I. FILES

The term *file* denotes either a segment or a multisegment file (i.e., a directory whose bit count is non-zero). The i/o system recognizes three types of files: unstructured, sequential, and indexed. The type of a file is recorded in its *file type* attribute. The file type of a segment or multisegment file will be unstructured unless specifically set otherwise. Thus all files created before the new i/o system are unstructured files, as are object segments, segments created by editors, etc.

The file type attribute is a new attribute in the directory entry for the file. As with the bit count, setting it only requires write access for the branch, not modify access for the directory.

Unstructured Files

The file is considered to be simply a sequence of 9-bit bytes. It is processed by stream i/o operations.

get_line	reads a line from the file, i.e. a sequence of bytes the last of which is a new-line character.
get_chars	reads a specified number of characters from the file.
put_chars	adds a specified number of characters to the file.
position	positions to the beginning or end of the file, skips forwards 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 file also contains control words used by the i/o system. It is processed by record i/o operations.

read	reads a record.
read_length	obtains the length of the next record.
write	adds a record to the file.
rewrite	replaces an existing record in the file.
delete	deletes a record from the file.
position	positions to the beginning or the end of the file, skips forwards or backwards over a specified number of records.

Indexed Files

The file contains a sequence of records and an index, which associates each record with a key. A key is a string of ASCII characters with no trailing blanks. Its length must be less than or equal to 256 and may be zero. Distinct records have distinct keys. The records are in key order according to the usual PL/I rules for string comparison.

The file is processed by the i/o operations available for sequential files and by the following operations.

seek_key — positions the file to the record with a specified key or defines the key for insertion of a record. This operation deletes trailing blanks to obtain the actual key.

read_key — obtains the key of the next record in the file.

The file index is organized as a B*-tree (see Knuth, v.3, 6.2.4), each node of the tree occupying one page in memory. In general, very few page accesses are required to access a record. For example, in a file with 10,000,000 records and with all keys of length 32, at most five index pages are touched in a random retrieval.

Files on Tape

Unstructured files and sequential files are supported on Multics standard tapes. The data base that contains tape registry information indicates whether the file is unstructured or sequential.

Each file on a nonstandard tape is interpreted as a sequential file by treating each physical record (i.e. each block) as a single logical record.

In the future unstructured and sequential files may be supported on ANSI standard tapes and on removable disk packs.

Files on tape are processed in the same way as files in the storage system except for the extra feature of processing a multifile nonstandard tape. No other use of tape is supported by the i/o system.

II. THE I/O SWITCH

Problems with the Old Switch

Performance. Use of the old i/o switch involves a procedure call to ios_, a table lookup on the stream name, and another procedure call to the actual DIM. This overhead is painful in most cases and is unacceptable in high volume i/o to/from files in the storage system (where the actual transfer of data can be done by a single EIS move).

Rings. There is no way to share a stream across rings.

Function. The old ios_ entries do not permit one to program in an attachment independent manner. Many entries have little or no value (and little or no support -- see Appendix A), while ios_$read and ios_$write must be used for many different functions. For example, with tty_, ios_$read gets a line; with tape_, it gets a specified number of words (not characters); with nstd_, it gets an entire record. Decent Fortran and PL/I support of i/o to/from files requires a larger set of entries, each one performing a more specific function.

I/O Names and Control Blocks

To avoid confusion with the old i/o switch terminology and with PL/I stream i/o, the terms "stream" and "stream name" are replaced by "i/o name". Conceptually, all i/o operations are performed on i/o names. However the implementation works with i/o control blocks which are accessible by the user. Each i/o control block is associated with a 32-character i/o name on a per ring basis. An i/o operation is performed by calling an entry value stored in the i/o control block, using a previously obtained pointer to the block. For example:

```
call user_input_ptr->iocb.get_line(user_input_ptr, target,
        target_length, actual_length, status);
```

The call to ios_ and the table lookup on the stream name are eliminated in most cases.

The remainder of this MSB describes i/o in terms of i/o control blocks. However the user working at command level or through a language's i/o facilities can ignore control blocks and think only of operations on i/o names.

Preparation of an i/o control block, iocb, for i/o operations requires two steps. The first is to call an appropriate attach routine. The second is to call iocb.open. The attach call establishes the source/target for the i/o. The open call establishes the processing mode, e.g., stream output or keyed

sequential input. A call to iocb.close terminates the switch block's opening but leaves it attached. A call to iocb.detach terminates the attachment. Thus the complete sequence of operations is:

attach
open
perform i/o
close
detach

The advantage of splitting up the old ios_$attach and ios_$detach in this way is that a user of the attach command need only specify the pathname, tape identification, etc. Details of the processing mode are supplied by the program when it calls iocb.open. The same attachment can hold through several open-close sequences, for example, when a file is to be processed by a sequence of programs.

Synonyms

One i/o control block, iocb1, may be attached as a synonym for a second i/o control block, iocb2. (See the attach command and the subroutines iox_$attach_syn and iox_$attach_syn_ring.) In general, performing an i/o operation through iocb1 will then have the same effect as performing it through iocb2. There are two exceptions.

1. Detaching iocb1 simply breaks the synonymization and has no effect on iocb2.

2. At the user's discretion, the attachment of iocb1 may inhibit some i/o operations. An attempt to perform an inhibited operation through iocb1 will have no effect. (The operation can still be performed through iocb2.)

The second i/o control block, iocb2, need not be attached at the time iocb1 is attached, but it must be attached before any i/o operations are performed.

It is possible for iocb1 and iocb2 to belong to different rings. The synonym attachment is allowed in this case provided

1) iocb1 belongs to a higher ring than iocb2,

2) the ring to which iocb1 belongs is less than or equal to the value of the ring bracket field in iocb2.

The ring bracket field in an i/o control block provides a basic control of cross ring sharing of i/o control blocks. More precise control can be accomplished through the inhibit feature. For example, suppose that in ring 2 user_i/o is attached to the terminal, and it is desired to allow normal terminal input in ring 4, but to have ring 2 monitor the output. To accomplish

this, set (in ring 2) the ring brackets of user_input and user_output to 4. Establish user_input as a synonym for user_i/o but with all except input operations inhibited. Make a special attachment of user_output so that i/o operations through it are handled by a monitor routine. Finally, attach user_input and user_output in ring 4 as synonyms for user_input and user_output in ring 2.

The implementation of the attachment of iocb1 as a synonym for iocb2 depends on their ring relationship.

1. If iocb1 and iocb2 are in the same ring, then the relevant entry values and pointers in iocb2 are also stored in iocb1. Any changes to these values in iocb2 are also made in iocb1 except for inhibited operations (see iox_$propogate.)

2. If iocb2 is in a lower ring, then the entry values in iocb1 are set to gates into the lower ring. The gate constructs a new argument list, validates the call, and calls the appropriate entry value in iocb2.

III. DEFINITIONS AND CONVENTIONS

Status Codes

All I/o routines return standard Multics status codes. The code will be zero, one of the error_table_ codes listed in Appendix A, or a code returned by some system routine called by the I/o system. If some form of attachment involves the possibility of errors too elaborate to describe by a standard status code, the most appropriate standard code is returned, and additional information is obtained by a subsequent call to iocb.control.

There are several entries in iox_ whose sole effect is to set their last argument to a particular status code. In this bulletin these entries are given names such as iox_$err_not_open, and no further description is given.

I/O Control Blocks

The following declaration defines that part of an I/o control block that is of interest to the user. Each block has a few more words at the end, which are used to maintain the per-ring table of I/o control blocks and to implement synonyms. These words are used only by the I/o system.

```
dcl 1 iocb based,
    2 name char(32),
    2 ring_bracket fixed bin,
    2 actual_switch_ptr ptr,
    2 attach_descrip_ptr ptr,
    2 attach_data_ptr ptr,
    2 open_descrip_ptr ptr,
    2 open_data_ptr ptr
    2 detach entry(ptr, fixed bin),
    2 open entry(ptr, fixed bin, bit(1) aligned, fixed bin),
    2 close entry(ptr, fixed bin),
    2 get_line entry(ptr, ptr, fixed bin, fixed bin, fixed bin),
    2 get_chars entry(ptr, ptr, fixed bin, fixed bin, fixed bin),
    2 put_chars entry(ptr, ptr, fixed bin, fixed bin),
    2 control entry(ptr, char(*), ptr, fixed bin),
    2 read entry(ptr, ptr, fixed bin, fixed bin, fixed bin),
    2 write entry(ptr, ptr, fixed bin, fixed bin),
    2 rewrite entry(ptr, ptr, fixed bin, fixed bin),
    2 delete entry(ptr, fixed bin),
    2 position entry(ptr, fixed bin, fixed bin, fixed bin),
    2 seek_key entry(ptr, char(256) varying aligned, fixed bin),
    2 read_key entry(ptr, char(256) varying aligned, fixed bin),
    2 read_length entry(ptr, fixed bin, fixed bin);
```

Many of the fields in iocb have already been mentioned. The significance of the others is as follows.

1. actual_switch_ptr points to iocb itself unless iocb is attached as a synonym for iocb2 in the same ring. In the

latter case, it equals iocb2.actual_switch_ptr.

2. attach_descrip_ptr is null if iocb is detached. Otherwise it points to a structure of the form:

```
declare 1 descrip based,
        2 length fixed bin (17) unal,
        2 string char (0 refer(length));
```

The string descrip.string contains a description of the attachment suitable for printing by the command print_attach_table.

3. attach_data_ptr is null or points to a data block set up by the routine that attached iocb.

4. open_descrip_ptr is null unless iocb is open. If iocb is open, open_descrip_ptr points to a structure like descrip in (2) above, and the string descrip.string contains, first, the meaning of the opening mode (Section IV, Table 1) and, second, "-append" if the opening is for output to a file with extension of the file. The string may contain additional information relevant to the particular opening/attachment.

5. open_data_ptr is null unless iocb is open or is attached as a synonym for iocb2 in the same ring. If iocb is open, open_data_ptr points to a data block set up by the routine that opened iocb. If iocb is attached as a synonym for iocb2 in the same ring, open_data_ptr equals iocb2.open_data_ptr.

6. control is the entry value for a catch-all I/O operation. Its meaning, if any, depends on the particular attachment/opening of the switch.

The Actual I/O Control Block

Each I/O control block, iocb, has an associated *actual I/O control block* defined as follows. If iocb is not attached as a synonym, then its actual I/O control block is the same as iocb. If iocb is attached as a synonym for iocb2, then the actual I/O control block of iocb is the same as the actual I/O control block of iocb2.

When iocb is attached, all I/O operations on iocb except detach and inhibited operations refer to the actual I/O control block. The treatment of actual_switch_ptr and open_data_ptr, enables the operation to be performed without additional overhead, provided the actual I/O control block belongs to the same ring as iocb.

States of an I/O Control Block

1. Detached. The pointers attach_descrip_ptr, attach_data_ptr, open_descrip_ptr, and open_data_ptr are null. The pointer actual_switch_ptr points to the i/o control block itself. The entries detach and open are iox_$err_not_attached. The other entries are all iox_$err_not_open. When initially created, an i/o control block is in the detached state, and the detach operation restores an i/o control block to the detached state.

2. Attached. The pointer attach_descrip_ptr points to a structure as specified above under Switch Blocks, item 2. If the i/o control block is attached as a synonym for iocb2 in the same ring, then actual_switch_ptr equals iocb2.actual_switch_ptr. Otherwise it points to the i/o control block itself. Unless it is attached as a synonym, the i/o control block is in either the open or closed state.

3. Open. In this state the i/o control block is attached but not as a synonym. The pointer open_descrip_ptr points to a structure as specified under Switch Blocks, item 4. The pointer open_data_ptr points to a control block appropriate for the attachment/opening. The entries for i/o operations enabled by the particular opening mode (Section IV, Table 1) are routines that will perform the operations. The entries open and detach are iox_$err_not_closed. The other entries are iox_$err_no_operation.

4. Closed. In this state the i/o control block is attached but not as a synonym. The pointers open_descrip_ptr and open_data_ptr are null. The entry open is a routine that will perform the open operation for the particular type of attachment. The entry detach is a routine that will perform the detach operation for the particular type of attachment. The entry control is iox_$err_not_open or is a routine that accepts orders that apply to a closed i/o control block for the particular type of attachment. (This would be an unusual attachment.) The other entries are iox_$err_not_open.

Attachment

A standard attachment associates an i/o control block with a file or device or establishes it as a synonym for another i/o control block. A standard attachment is made through the attach command (Section VI) or a call to one of the attach entries in iox_ (Section V).

A nonstandard attachment stores entry values and pointers in an i/o control block that will cause subsequent i/o operations on the i/o control block to behave in the desired way. For example, they might invoke routines that read and write logical records on a specially formatted tape. If the non-standard attachment is to be compatible with other system i/o functions (e.g., with PL/I i/o statements), then:

1. The I/O control block must be in the detached state before the special attachment is made.

2. After attachment, its state must be open or closed as described above.

3. All I/O operations subsequently performed through the I/O control block must conform to the conventions given in this document.

IV. I/O OPERATIONS

This section describes the i/o operations, that is, the effects of the routines that are invoked by calls to entries in an i/o control block.

Notes on Files and Designators

Four designators are referenced in the description of i/o operations on files: next-character-designator, next-record-designator, current-record-designator, and insert-key-designator. The first three define abstractly the current position in the file. The fourth holds the key of the next record to be inserted in an indexed file. The actual implementation of these designators is internal business of the i/o system.

Two distinct i/o control blocks should not be open at the same time for operations on the same file, whether in the same or different processes, unless both are opened for input. Although the i/o system detects a few cases of the problem, it is the user's responsibility to prevent its occurrence.

In general, the bit counts for a file's segments are set only by the open and close operations.

Notes on Devices

The only devices considered in this bulletin are terminals attached through tw_ or a DIM with similar behavior (e.g., ntty). The treatment of quits is not defined in this bulletin. Printers, punches, and card readers may be handled in the future.

General Conventions

The following arguments have the same meaning in all operations.

1) iocb_ptr points to the i/o control block. (Input)

2) status is a standard Multics status code. (Output)

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

4) buff_len is the length in bytes of the buffer. It must be nonnegative. (Input)

No changes are made to an i/o control block except those indicated in the operation descriptions. In particular, the only operations that change i/o control blocks are open, close, detach, and (possibly) control. Any changes made are to the actual control block, and they are reflected to certain other control blocks by iox_$propagate.

Operation: open

This operation opens the actual i/o control block, aiocb, for i/o operations. Those operations specified by mode (Table 1) will be supported through aiocb. The other operations will return with status equal error_table_$no_operation.

Usage

```
call iocb.open(iocb_ptr, mode, append, status);
```

1) mode — is one of the thirteen modes listed in Table 1. (Input)

2) append — is meaningful only if the mode is stream output, stream input-output, sequential output, sequential input-output, or keyed sequential output, and if aiocb is attached to a file. If append is "1"b or if the attachment specified append, then the opening is for extension of the file. Otherwise the opening is for truncation of the file.

Notes

If aiocb cannot be opened, it is left unchanged, and a nonzero status is returned.

If aiocb can be opened, it is changed to the open state, and the original value of iocb.detach is saved. The open operation then calls iox_$propogate(addr(aiocb)).

File Opening

When aiocb is attached to a file, the effect of open depends on the file as follows.

1) If the file already exists, and the opening is for input or update or is for output or input-output with extension of the file, then the opening is carried out only if the file type is compatible with the opening mode (Table 2). Further, if the opening is for input or update, the file must not be empty.

2) If the file already exists, and the opening is for output or input-output with truncation of the file, the file is set empty. Moreover if the file's type is not compatible with the opening mode (Table 2), the file's type is changed to the default type for that opening mode. (Table 2)

3) If the file does not exist, and the opening is for input or update, the opening is not carried out.

4) If the file does not exist, and the opening is for output or input-output, a file is created with the specified

pathname. The file's type is the default type for the opening mode (Table 2).

Open sets the file position designators as shown in Table 3.

Device Openings

If alocb is attached to a terminal, the only allowed opening modes are stream input, stream output, and stream input-output.

It is possible that the system routine that attaches a device, also opens alocb.

Operation: close

This operation restores the actual i/o control block, alocb, to its state immediately before opening.

Usage

```
call iocb.close(iocb_ptr, status);
```

Notes

If alocb is not open, or if it cannot be closed for some other reason, it is left unchanged and a nonzero status is returned.

After alocb is restored to its state before opening, the close operation calls iox_$propogate(addr(alocb)).

Operation: detach

This operation restores the i/o control block, iocb, pointed to by iocb_ptr to the detached state.

Usage

```
call iocb.detach(iocb_ptr, status);
```

Notes

If iocb is not attached, if it is open, or if it cannot be detached for some other reason, then it is left unchanged and a nonzero status is returned.

After iocb is restored to the detached state, the detach operation calls iox_$propogate(iocb_ptr).

TABLE 1 - ALLOWED I/O OPERATIONS

Opening Mode-Meaning \ i/o operation	get_line	get_chars	put_chars	read	write	rewrite	delete	read_length	position	seek_key	read_key	close	control
1 - stream input	x	x							4			x	1
2 - stream output			x						4			x	1
3 - stream input-output	x	x	x						4			x	1
4 - sequential input				x				x	x			x	1
5 - sequential output					x							x	1
6 - sequential input-output				x	x			x	x			x	1
7 - sequential update				x		3	x	x	x			x	1
8 - keyed sequential input				x				x	x		x	x	1
9 - keyed sequential output					x					2		x	1
10 - keyed sequential update				x		x	x	x	x	x	x	x	1
11 - keyed nonsequential input				x				x		x		x	1
12 - keyed nonsequential output					x					x		x	1
13 - keyed nonsequential update				x	x	x	x	x		x		x	1

1. Depends on the attachment.
2. Keys must be in order.
3. If attached to a sequential file, the length of the new record must equal the length of the replaced record.
4. Allowed if attached to a file in the file system.

TABLE 2 - COMPATIBLE FILE ATTACHMENTS

Opening Mode-Meaning \ File Type	unstructured	sequential	indexed
1 - stream input	x	1	1
2 - stream output	x,D		
3 - stream input-output	x,D		
4 - sequential input		x	x
5 - sequential output		x,D	
6 - sequential input-output		x,D	
7 - sequential update		2	x
8 - keyed sequential input			x
9 - keyed sequential output			x,D
10 - keyed sequential update			x
11 - keyed nonsequential input			x
12 - keyed nonsequential output			x,D
13 - keyed nonsequential update			x

D indicates the default file type for the opening mode.

1. Allowed only if the attachment specifies that the file is to be read as unstructured.
2. File must be in file system.

TABLE 3 - FILE POSITION DESIGNATORS AT OPEN

Mode-Meaning \ Designator	next-character-designator	next-record-designator	current-record-designator	insert-key-designator
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		end-of-file		
6 - sequential input-output		end-of-file		
7 - sequential update		first record	null	
8 - keyed sequential input		first record		null
9 - keyed sequential output		end-of-file		null
10 - keyed sequential update		first-record	null	null
11 - keyed nonsequential input				
12 - keyed nonsequential output				null
13 - keyed nonsequential update				null

In openings where no value is indicated for a designator, the designator is ignored in all subsequent i/o operations, even though the operation description may refer to it.

Operation: get_line

This operation reads characters from a file or device into a buffer until the buffer is full or a new line character is read.

Usage

```
call iocb.get_line(iocb_ptr, buff_ptr, buff_len, actual_len,
     status);
```

1) actual_len is the number of characters read into the buffer. (Output)

Notes

The operation returns with status equal to zero if and only if the last character read was a new line character.

If the buffer is filled without reading a new line character, status is set to error_table_$long_record.

If the actual i/o control block is attached to a file, the next character_designator is set to designate the character following the last character read or, if that character does not exist, to end-of-file. If the read exhausts the file without filling the buffer or reading a new line character, status is set to error_table_$end_of_inf.

Operation: get_chars

This operation reads characters from a file or device into a buffer.

Usage

```
call iocb.get_chars(iocb_ptr, buff_ptr, buff_len, actual_len,
     status);
```

1) actual_len is the number of characters read into the buffer. (Output)

Notes

The operation returns with status equal to zero if and only if the number of characters read equals the buffer length.

If the actual i/o control block is attached to a file, the next-character-designator is set to designate the character following the last character read or, if that character does not exist, to end-of-file. If the read exhuasts the file without filling the buffer, status is set to error_table_$end_of_inf.

Operation: put_chars

This operation write characters from a buffer to a file or device.

Usage

```
call iocb.put_chars(iocb_ptr, buff_ptr, buff_len, status);
```

Notes

If the actual i/o control block is attached to a file, the characters are added at the end of the file and the next-character-designator is set to end-of-file. However, if, at the beginning of the operation, the next-character-designator designates a character (possible if the opening is for input-output), then the file is first truncated to the preceding character (if any).

Operation: control

This operation is used to request control functions which are special to a particular attachment/opening.

Usage

```
call iocb.control(iocb_ptr, order, info_ptr, status);
```

1) order — is the name of the particular control function requested.

2) info_ptr — is null or points to data whose form depends on the particular attachment/opening of the actual i/o control block.

Notes

If order is not a control function recognized by the particular attachment/opening, status is set to error_table_$no_operation.

This operation corresponds to the old ios_ entries order, resetread, resetwrite, abort, and changemode.

Operation: read

This operation reads the next record from a sequential file or indexed file into a buffer.

Usage

```
call iocb.read(iocb_ptr, buff_ptr, buff_len, rec_len, status);
```

1) rec_len — is the length in bytes of the record in the file. (Output)

Notes

The operation returns with status equal zero if and only if

1) The next-record-designator designates a record.

2) The record's length equals buff_len.

3) The record is successfully read into the buffer.

If the record's length is less than buff_len, the record is read into the first part of the buffer, and status is set to error_table_$short_record. If the record's length is greater than buff_len, the first part of the record is read into the buffer, the remainder of the record is ignored, and status is set to error_table_$long_record. In all cases rec_len is set to the length of the record.

With any form of sequential opening, the next-record-designator is set to designate the record following that read or, if no such record exists, to end-of-file. With any form of nonsequential opening, the next-record-designator is set to null.

The current-record-designator is set to designate the record read by this operation.

Operation: write

This operation adds the contents of the buffer to the file as a new record. If the file is indexed, the key of the record must have been defined by a preceding seek_key operation.

Usage

```
call iocb.write(iocb.ptr, buff_ptr, buff_len, status);
```

Notes

The record added to the file is the byte string contained in the buffer. The buffer length may be zero (i.e. the byte string may be null); but if the file is a nonstandard tape, no record is added in this case.

If the file is sequential, the record is added at the end of the file, and the next-record-designator is set to end-of-file. However, if, at the beginning of the operation, the next record designates a record (possible if the opening is for input-output), then the file is first truncated to the preceding record or, if there is no preceding record, to empty.

If the file is indexed, the insert-key-designator must designate a key. The record is added to the file with this key. The current record designator is set to the record just written; and, for keyed sequential update, the next record designator is

set to the following record or, if there is no following record, to end-of-file.

Operation: rewrite

This operation replaces the record designated by the current-record-designator by the contents of the buffer.

Usage

```
call iocb_rewrite(iocb_ptr, buff_ptr, buff_len, status);
```

Notes

This operation returns with status equal zero if and only if

1) The current-record-designator designates a record.

2) In the case of a sequential file, the length of the record designated by the current record designator must equal buff_len. If this does not hold, the record in the file is not replaced.

3) Replacement of the record in the file by the contents of the buffer is successful.

This operation leaves the file position designators unchanged.

Operation: delete

This operation deletes the record designated by the current-record-designator from the file.

Usage

```
call iocb.delete(iocb_ptr, status);
```

Notes

This operation sets the current-record-designator to null. In a sequential opening, it sets the next-record-designator to the record following the deleted record or, if no such record exists, to end-of-file.

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

Operation: position

This operation positions to the beginning or end of a file or skips forward or backwards over a specified number of lines or records.

Usage

```
call iocb.position(iocb_ptr, code, skip, status);
```

1) code — equals -1 for positioning to the beginning of the file, equals +1 for positioning to the end of the file, and equals 0 for skipping lines or records. (Input)

2) skip — is meaningful only if code equals 0. It specifies the number of lines (unstructured file) or records (sequential file or indexed file) to be skipped. A negative value of skip causes the file to be positioned backwards. A value of zero leaves the file position unchanged. (Input)

Notes

Positioning to the beginning of the file sets the next-record-designator (next-character-designator) to designate the first record (first character) of the file or, if the file is empty, to end-of-file. Positioning to the end of file sets the next-record-designator or next-character-designator to end-of-file.

For skipping records, let the next-record-designator point to the mth record in the file, and let n be the number of records to be skipped (i.e., the absolute value of skip). Then a successful backwards skip sets the next-record-designator to record m-n. A successful forward skip sets it to record m+n or, if there are only (m+n-1) records in the file, to end-of-file. The current-record-designator is set to the record immediately proceeding that designated by the next-record-designator or, if there is no such record, to null.

Successfully skipping n lines backwards (skip <0) moves the next character designator backwards over n new-line characters and then over additional characters so that it finally points to the character immediately following the (n+1)st new-line character or to the first character in the file. Successfully skipping n lines forwards (skip >0) moves the next character designator forwards over n new-line characters, leaving it pointing at the character immediately after the nth new-line character, or, if no such character exists, leaving it set to end-of-file.

If the relevant part of the file contains too few lines or records for a successful skip, then the next-record-designator (next-character designator) is set to the first record in the file (first character in the file) if skip <0, and it is set to end-of-file if skip >0. In these cases status is set to error_table_$end_of_inf.

Operation: seek_key

This operation positions an indexed file to the record with a given key or, if no such record exists, prepares for the insertion of a record with the given key.

Usage

```
call locb.seek_key(locb_ptr, key, status);
```

1) key — is the given key. It must contain only ASCII characters. Trailing blanks are ignored. (Input)

Notes

If a record with the given key exists in the file, then status is set to zero, both the current-record-designator and the next-record-designator are set to that record, and the insert-key-designator is set to null.

If a record with the given key does not exist, then status is set to error_table_$no_key, the insert-key-designator is set to the given key with trailing blanks removed, and both the current-record-designator and the next-record-designator are set to null.

Operation: read_key

This operation returns the key of the record designated by the next-record-designator.

Usage

```
call locb.read_key(locb_ptr, key, status);
```

1) key — is the next record's key. (Output)

Operation: read_length

This operation returns the length (in 9-bit bytes) of the record designated the next-record-designator.

Usage

```
call locb.read_length(locb_ptr, length, status);
```

1) length — is the next record's length. (Output)

V. SUBROUTINES

This section describes "iox_$..." entries that handle attachment and related functions. Note, however, that in this MSB the section is not complete, e.g. tape attachment is not covered.

Entry: iox_$attach_broadcast.

This entry performs a nonstandard attachment of an i/o control block, iocb, that enables the operations put_chars and control to be broadcast to a set of target i/o control blocks, iocb1...iocbn. That is, a call to iocb.put_chars or iocb.control results in calls to iocb1.put_chars or iocb1.control for i=1,2,...n. Each call to iox_$attach_broadcast connects iocb to one target i/o control block

Usage

```
declare iox_$attach_broadcast entry(ptr, ptr, fixed bin);

call iox_$attach_broadcast(iocb_ptr, target_ptr, status);
```

1) iocb_ptr points to the i/o control block, iocb, to be attached. It must currently be detached or be attached by this entry. (Input)

2) target_ptr points to the i/o control block, iocb1, to which iocb is to be attached. (Input)

3) status is a standard Multics status code. (Output)

Note

With this attachment, the entries iocb.put_chars and iocb.control are set to special routines that broadcast calls. The entry iocb.detach is set to a routine that detaches iocb from all target i/o control blocks restoring it to the detached state. (See iox_$detach_broadcast for detaching from a single target). All other entries in iocb are set to iox_$err_no_operation.

Entry: iox_$attach_discard_output.

This operation attaches an i/o control block so that it can be opened for stream output, sequential output, keyed sequential output, or keyed nonsequential output. When the switch block is opened, the i/o operations supported for the opening will return with status equal to zero but will not transmit any data or have any other effects.

Usage

```
declare iox_$attach_discard_output entry(ptr, fixed bin);
```

```
call iox_$attach_discard_output(iocb_ptr, status);
```

1) iocb_ptr points to the i/o control block to be attached. (Input)

2) status is a standard Multics code. (Output)

Entry: iox_$attach_file.

This entry attaches an i/o control block to a file in the storage system.

Usage

```
declare iox_$attach_file entry(ptr, char(*), bit(1) aligned,
        fixed bin, fixed bin);

call iox_$attach_file(iocb_ptr, pathname, append, interp,
        status);
```

1) iocb_ptr points to the i/o control block to be attached. (Input)

2) pathname is the pathname of the file. (Input)

3) append is "1"b if the file is to be extended when opened for output and is "0" b otherwise. (Input)

4) interp is zero if the file is to be interpreted normally. If this argument is one, a sequential or indexed file will be treated as an unstructured file. (Input)

5) status is a standard Multics status code. (Output)

Notes

The specified file does not have to exist. See the description of the open operation. (Section IV).

If the argument interp is one, the file will be processed as an unstructured file regardless of its true file type. The opening must be for stream input and all bytes in the file (including control words) will be read.

Entry: iox_$attach_syn

This entry attaches an i/o control block, iocb, as a synonym for a target i/o control block in the same ring. After the attachment, an operation on iocb has the same effect as an operation on the target except for detach and except for operations specifically inhibited.

Usage

```
declare iox_$attach_syn entry(ptr, ptr, bit(36) aligned,
        fixed bin);

call iox_$attach_syn(iocb_ptr, target_ptr, inhibit, status);
```

1) iocb_ptr points to the i/o control block, iocb, to be attached. (Input)

2) target_ptr points to the i/o control block for which iocb is to be a synonym. (Input)

3) inhibit indicates which operations, if any, are to be inhibited. If bit i of inhibit is "1"b, the ith operation in iocb, beginning with open, is inhibited. Thus, if inhibit equals "00001"b, put_chars is inhibited. (Input)

4) status is a standard Multics status code. (Output)

Notes

The target i/o control block does not have to be attached when iox_$attach_syn is called. If it is attached as a synonym, its actual i/o control block must not be iocb.

Inhibited operations return status equal to error_table_$no_operation.

Entry: iox_$attach_syn_ring

This entry attaches an i/o control block as a synonym for a target i/o control block in a specified ring.

Usage

```
declare iox_$attach_syn_ring entry(ptr, char(32),
        bit(36) aligned, fixed bin, fixed bin);

call iox_$attach_syn_ring(iocb_ptr, name, inhibit, ring,
        status);
```

1) iocb_ptr points to the i/o control block to be attached. (Input)

2) name is the name of the target i/o control block (Input)

3) inhibit indicates which operations are to be inhibited. See iox_$attach_syn. (Input)

4) ring is the ring of the target i/o control block. It must be less than or equal to the calling ring.

(Input)

5) status Is a standard Multics status code. (Output)

Notes

The target i/o control block must already exist and its ring_bracket must be greater than or equal to the calling ring.

Operations will be forwarded to the target i/o control block only as long as its ring_bracket is greater than or equal to the calling ring.

Entry: iox_$detach_broadcast

This entry detaches a broadcasting i/o control block from a single target i/o control block.

Usage

```
declare iox_$detach_broadcast (ptr, ptr, fixed bin);

call iox_$detach_broadcast(iocb_ptr, target_ptr, status);
```

1) iocb_ptr points to an i/o control block, iocb, that was attached by iox_$attach_broadcast. (Input)

2) target_ptr points to an i/o control block, iocb1, that is currently a target for iocb. (Input)

3) status Is a standard Multics status code. (Output)

Entry: iox_$find_iocb

This entry accepts the name of an i/o control block as its input. Its output is a pointer to the i/o control block with that name and in the current ring. If the i/o control block does not already exist it is created and initialized to the detached state with its ring bracket equal to the current ring number.

Usage

```
declare iox_$find_iocb entry(char (32), ptr, fixed bin);

call iox_$find_iocb (name, iocb_ptr, status);
```

1) name Is the name of the i/o control block. (Input)

2) iocb_ptr Is a pointer to the i/o control block. (Output)

3) status Is a standard Multics status code (Output)

Entry: iox_$find_iocb_n

This entry may be used to find all i/o control blocks in the calling ring. It accepts an integer, n, as input. If the table of i/o control blocks in the calling ring contains an i/o control block in the nth position, it returns a pointer to this block.

Usage

```
declare iox_$find_iocb_n entry(fixed bin, ptr, fixed bin);

call iox_$find_iocb_n (n, iocb_ptr, status);
```

1) n is the number of the i/o control block to be located. (Input)

2) iocb_ptr points to the i/o control block. (Output)

3) status is a standard Multics status code. (Output)

Notes

If the i/o control block does not exist, iocb_ptr is set to null, and status is set to error_table_$no_iocb. The i/o control blocks are numbered 1,2,3,..., so a return with status equals error_table_$no_iocb means that no block with number greater than or equal to n exists, provided that n is greater than zero.

Entry iox_$propogate

This entry reflects changes made to an i/o control block, iocb, to all i/o control blocks in the same ring whose actual i/o control block is iocb. It must be called by all attach routines and all i/o operations when they change an i/o control block.

Usage

```
declare iox_$propogate entry(ptr, fixed bin);

call iox_$propogate(iocb_ptr, status);
```

1) iocb_ptr points to the i/o control block, iocb, that has been changed. (Input)

2) status is a standard Multics status code. (Output)

Notes

For each i/o control block, x, attached as a synonym for iocb, iox_$propogate performs the following steps.

1) Set x.open_data_ptr = iocb.open_data_ptr.
2) For each i/o operation f other than detach, if f was not inhibited by the attachment of x, set x.f = iocb.f.

3) Call lox_$propogate (addr (x), status);

VI. COMMANDS

This section briefly discusses i/o system commands and general commands that apply to files (e.g., copy, list). Detailed specifications will be issued later. Note that some of the commands already work as specified.

Command: attach

This command attaches an i/o control block.

Usage

attach name type -opt1-...-optn-

1) name — is the name of an i/o control block.

2) type — specifies the type of attachment. It is one of -pn, -tape, -syn, -discard_output, -broadcast.

3) opt1 — depends on type.

Notes

In general, this command will be extended to handle any type of attachment for which there is an "iox_$attach" entry.

Command: copy

This command will copy files whether single segment or multisegment. The source or destination may be a tape.

Command: create

This command creates files of a specified type (e.g. indexed) in the storage system. The default is unstructured, i.e., is the same as the old create command.

Command: dprint

This command causes specified files to be queued for printing on a line printer.

Command: dpunch

This command causes specified files to be queued for punching.

Command: delete

This command deletes files from the storage system.

Command: io_op

This command causes a specified i/o operation to be performed on an i/o control block.

Command: iocall

This command is not part of the Multics product.

Command: list

This command lists information about directory entries. The default is to list information about files.

Command: move

This command moves a file to a new position in the storage system.

Command: print

This command prints a specified ASCII file through the i/o control block user_output.

Command: status

This command recognizes the various types of files.

Command: truncate

This command truncates a file. Unstructured files are truncated to a specified bit count. Sequential and indexed files are truncated to a specified record.

Appendix A - Old Dims

Table A-1 summarizes the properties of the principle DIMS and IOSIMs supported through the old i/o system and indicates how they are supported in the new system. The support classes and notes are as follows.

Support Class A

For these DIMS and IOSIMS there is complete compatibility between the old and new i/o systems. The DIM or IOSIM may be attached by a call to ios_$attach or by use of the corresponding attach routine or command in the new system. Once the i/o control block is attached and opened, i/o may be performed by calls to ios_$read or ios_$write and/or by the operations get_line, get_chars, and put_chars. A call to the ios_$attach also opens the i/o control block, a call to ios_$detach closes and detaches the i/o control block.

Support Class B

This applies only to the file_ IOSIM. There is compatibility with the following exceptions.

1) The only allowed element sizes are 9 and 36, and the element size applies only to calls to ios_$read and ios_$write, not to get_chars, etc.

2) When the element size is 9, the delimiter for ios_$read is NL. When the element size is 36, there are no delimiters.

3) Calls to ios_$read, etc. may only be used when the i/o control block is opened for stream i/o.

4) Calls to seek, tell, get_delim, and set_delim are not allowed.

Support Class C

The DIM or IOSIM may only be used by calls to ios_$read, etc. When a i/o control block is attached to such a DIM or IOSIM, all i/o operations return with status equal error_table_$old_dim.

All DIMS and IOSIMS not listed in Table A-1 fall in this class. However, if a DIM or IOSIM has essentially the same properties as tw_, it may be promoted to Class A with modest effort.

Note 1: The only orders are error_count and rewind. The only seek allowed means rewind (according to the MPM).

Note 2: The only effect is to return old information.

Note 3: Orders as for tw_ plus set_terminal_type.

Note 4: Orders as for tw_ plus set_arg_list.

Note 5: Abort and resetread cause logout. Resetwrite is ignored.

General Note: Apparently no DIMS or IOSIMS support readsynch or writesynch or allow setting of breaks.

TABLE A-1

DIM	file_	tw_	tape_	nstd_	broadcast	discard output	mrdim	ocdim	ntty	absentee_	tek_
support class	B	A	C	C	A	A	A	A	A	A	A
element size	any	9	36	36			9	9	9	9	9
delims	any	NL	none	none			NL	NL	NL	NL	NL
read	X	X	X	X			X	X	X	X	X
write	X	X	X	X	X	X	X	X	X	X	X
order		X	1	X			X	2	3	4	X
abort		X			X	X	X	X	X	5	X
seek	X		1								
getsize	X	X	X	X				X	X	X	X
setsize	X										
resetread		X					X	X	X	5	X
resetwrite		X			X	X	X	X	X	5	X
getdelim	X										
setdelim	X										
changemode		X		X			X	2	X		X
tell	X										

APPENDIX B - STATUS CODES

The following standard Multics Status Codes are used by the I/O system.

```
error_table_$already_attached
error_table_$circular_syn
error_table_$empty_file
error_table_$end_of_info
error_table_$incompatible_attach
error_table_$key_order
error_table_$long_record
error_table_$no_device
error_table_$no_file
error_table_$no_key
error_table_$no_operation
error_table_$no_record
error_table_$no_iocb
error_table_$not_attached
error_table_$not_closed
error_table_$not_open
error_table_$old_dim
error_table_$short_record
error_table_$tape_error
```

APPENDIX C - PL/I I/O

File State Blocks

A PL/I i/o statement references a file state block. See the Multics PL/I Language Manual, 11.2. There is one file state block for each distinct file constant. An i/o statement may reference the file state block through its associated file constant through a file variable or (for sysin and sysprint) implicitly.

```
declare sysprint file constant, f file variable;
f=sysprint;
put file (sysprint) list (x);
put file (f) list (x);
put list (x);
```

All three put statements reference the file state block associated with the external file constant sysprint.

Each PL/I file state block is associated with an i/o control block, which depends on the associated file constant as follows.

1) For the external file constant sysin, the i/o control block is named user_input.

2) For the external file constant sysprint, the i/o control block is named user_output.

3) For any other external file constant, the i/o control block has the same name as the constant.

4) For an internal file constant, the i/o control block has a unique name. Distinct internal file constants are thus associated with distinct i/o control blocks.

Opening PL/I Files

The effect of a PL/I open statement (or implicit opening) depends on the state of the i/o control block associated with the PL/I file state block.

1) If the i/o control block is open, its opening mode must be the same as the opening mode specified in the open statement. If it is not, undefinedfile is signaled.

2) If the i/o control block is attached but closed, it is opened for the mode specified in the open statement. In this case the open statement must not contain a title option.

3) If the i/o control block is not attached, it is attached in accordance with the title option in the open statement. If there is no title option, it is attached to a file whose

pathname is the same as the file constant's name.

Stream I/O

When a PL/I file state block is opened for stream i/o operations, a buffer is associated with it. The individual PL/I i/o operations transmit data to/from this buffer. The points at which data is transmitted to/from the i/o system are therefore of interest, especially in cases where an i/o control block is being used in both PL/I and non-PL/I programs. The rules are as follows.

1) During execution of a put statement, the buffer contents are written whenever the buffer is filled up and when the output data list has been completely processed.

2) During execution of a get statement, data is read into the buffer whenever the buffer is empty and the input data list is not exhausted. The operation get_line is used to read the data. Hence a complete line will be read unless the line length exceeds the buffer size.

The following conclusions can be drawn from these rules.

1) Don't switch from PL/I output to foreign output in the middle of a put statement.

2) Don't switch from PL/I input to foreign input in the middle of a line.

Record I/O

All forms of record i/o openings are available in PL/I except sequential input output. (Note that a PL/I opening for "direct" corresponds to an i/o system opening for "keyed nonsequential").

The i/o system only handles records containing an integral number of 9-bit bytes and stored on a 9-bit byte boundry. Therefore PL/I i/o has to use special techniques when a PL/I record contains unaligned bit strings.

1) On output, the record must first be copied to an intermediate byte-aligned buffer unless the compiler can tell that the record is already byte-aligned.

2) On input, the record must be read into an intermediate byte aligned buffer unless the compiler can tell that the target variable is byte aligned and that it occupies an integral number of bytes.

Note that in some cases the record in the file will contain a few garbage bits at the end (to pad it to an integral number of bytes), but this doesn't confuse PL/I.

The simple forms of record I/O statements can be implemented very efficiently. Consider, for example,

```
read file (f) into (x);
```

The compiled code evaluatas the file expression f, obtaining a pointer pf to a file state block and the code evaluates the target reference x obtaining a pointer px to x and a length n. The read operation can then be performed by

```
if pf->fsb.funny_flag
then call pl1io_ (...);
else do;
   call pf->fsb.iocb_ptr->iocb.read(pf->fsb._ptr, px, n, status);
   if status^=0
   then do;
      /*analyze error*/
      end;
   end;
```

The test on funny_flag detects oddities relating to the file state block, e.g. this is first use, or there is a buffer to dispose of (from read file (f) set (p);). Normally funny_flag is "0"b, and the read is performed by a direct call to the I/O system. If the file is in the storage system this will be the only subroutine call.

Closing PL/I Files

The normal system action for the finish condition closes all PL/I file state blocks, all Fortran files, all Basic files, and all I/O control blocks not otherwise covered. Thus any data in buffers will be written out, file bit counts will be properly set, tape labels written, etc.

If a PL/I file state block is open when its associated I/O control block is closed by a direct call to the I/O system, data in PL/I buffers may be lost.

APPENDIX D - FORTRAN I/O

Logical Units

Each Fortran logical unit nn (nn=01, 02,...99) is associated with an i/o control block as follows.

1) Logical unit 05 is associated with the i/o control block named user_input.

2) Logical unit 06 is associated with the i/o control block named user_output.

3) Logical unit nn, nn not equal 05 or 06, is associated with the i/o control block named filenn.

File Attachment and Opening

If a Fortran i/o statement is executed and the associated i/o control block is not open, it is opened as follows.

1) If the i/o control block is not attached, it is attached to the file whose pathname is filenn where nn is the logical unit number.

2) If the i/o statement is a formatted read, the i/o control block is opened for stream input.

3) If the i/o statement is a formatted write, Fortran attempts to open the switch block for stream input-output. If that fails, it opens it for stream output.

4) If the i/o statement is a binary read, Fortran opens the i/o control block for sequential input.

5) If the i/o statement is a binary write, Fortran attempts to open the i/o control block for sequential input-output. If that fails, it opens it for sequential output.

Fortran Records

If a binary read or write statement specifies more than one variable, the i/o system transmits the logical record to/from a buffer supplied by Fortran, and Fortran moves data between the buffer and the seperate variables. Note that nothing is left in the buffer when the i/o statement is finished.