To: Distribution

From: M. Asherman

Date: 12/15/76

Subject: New File Type Built from Areas

Introduction

A new file type is proposed whose appearance is identical to that of blocked files, except for the absence of a characteristic maximum record length (besides the segment size limit). Such files, termed "variable," can be implemented efficiently within vfile_ using system area management routines.

Uses of variable files

The primary motivation is to support APL I/O interchangeably with other forms of sequential language I/O and structured files.

Another reason for having these files is that they provice a standard interface for extensible permanent areas. This would be of use in indexed files, for example, as a replacement for the dynamic file space management logic.

Implications for indexed files

Besides improving performance and centralization of code, the use of extensible permanent areas can lead to some functional enhancements of indexed files.

Most notably, users could target independently synchronized allocations and frees at the msf component level, without locking the file as a whole.

The implementation of single segment indexed files would also be greatly facilitated by the use of a single, general space management routine. This feature will probably require a new allocation entry and/or area header bit which forces page alignment (for index modes).

Permanent areas

An initial implementation of variable files is possible with minimal changes to area routines. Ultimately, however, it may be desirable to introduce area-level synchronization/recovery logic for permanent areas (as distinguished by a bit in the area-header). Once this has been done, a similar consistency mechanism can be incorporated into variable files at the vfile_ level. Such permanent areas would be a useful extension in themselves, independent of their relation to vfile_, because they can permit efficient sharing among processes. Furthermore, the user could rely upon automatic recovery from internal area inconsistencies in the event of interrupted allocate or free operations.

Distinctive features of variable files

As required by APL, the first and last non-zero record positions would be maintained, and made available through the "file-status" control order, et.al. Records are located via an array of descriptors, each consisting of an msf component number and word offset.

The delete operation, which is currently not supported on blocked files, could be implemented in both variable and blocked files with a modified specification of the operation for these cases. Specifically, if the file is blocked or variable, a delete operation does not cause the current record position to disappear, unless the current record is the last in the file. Otherwise, the record is just replaced with one of zero length.

MPM documentation

Revised MPM documentation is attached describing the changes proposed for a first implementation of variable files. It has not yet been decided whether any recovery logic will be initially incorporated.

Descriptions of proposed extensions to system area routines, and a revised set_file_lock control order will be provided in a future MTB.

Table 5-1. Opening Modes and Allowed Input/Output Operations

Opening Mode No.	Name	get_line	get_chars	put_chars	read_record	rewrite_record	delete_record	read_length	position	seek_key	read_key	close	write_record	control modes	
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	x	1	1
6	sequential_input_output				x			x	x			x	x	1	1
7	sequential_update				x	x		x	x			x		1	1
8	keyed_sequential_input				x			x	x	x	x	x		1	1
9	keyed_sequential_output									x		x	x	1	1
10	keyed_sequential_update				x	x	x	x	x	x	x	x	x	1	1
11	direct_input				x			x		x		x		1	1
12	direct_output									x		x	x	1	1
13	direct_update				x	x	x	x		x		x	x	1	1

1. Depends on the attachment.
2. Allowed if attached to a file in the storage system.
3. ~~Allowed unless file is blocked.~~
4. Allowed for ~~blocked and sequential~~ non-indexed files in the storage system.

Interrupted Input/Output Operations

It may happen that an I/O operation being performed on a particular I/O switch, switch_1, is interrupted, e.g., by a quit signal or an access violation signal. In general, until the interrupted operation is completed, or until switch_1 is closed, it is an error (with unpredictable consequences) to perform any I/O operation except close on switch_1. However, some I/O modules (tty_ in particular) allow other operations on switch_1 in this situation. (See the module descriptions in Section III of the MPM Subroutines for details.) If the switch switch_1 is closed while the operation is interrupted, control must not be returned to the interrupted operation.

PROGRAMMING LANGUAGE INPUT/OUTPUT FACILITIES

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). The following statements about this sort of sharing of switches apply in most cases:

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 close (detach) a switch only if they opened (attached) it.

2. A switch opened for stream_input may be used both directly and through language 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 likely to arise after issuing a quit signal (pressing the appropriate key on the terminal, e.g., ATTN, BRK, etc.).

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 (e.g., by the issuing of a quit signal) is made while the language I/O routines are processing an I/O statement.

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.

A direct call to the I/O system has no effect on control blocks and buffers maintained by the language I/O routines and is likely to cause garbled input or output. The close_file command (described in the MPM Commands) closes PL/I and FORTRAN control blocks used by the language I/O routines. For details on the facilities of a particular language and for a discussion of the usage of related Multics commands, see the reference manual and/or user's guide for that language.

FILE INPUT/OUTPUT

The I/O system distinguishes ~~four~~ Five types of files: unstructured, sequential, blocked, and indexed. These types pertain to the logical structure of a file, not to the file's 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 multisegment file; but this does not affect the meaning of I/O operations on the file.

variable,

Blocked Files

A blocked file contains a sequence of records. Each record is a string of 9-bit bytes. The length of a record may range from zero to a preset maximum value associated with the file.

The following I/O operations apply to blocked files:

read_record	reads the next record
read_length	obtains the length of the next record
write_record	adds a record to the file or replaces a record
rewrite_record	replaces a record
position	positions to the beginning or end of the file, skips forward or backward over a specified number of records. Also, given its ordinal position, (0, 1, 2, ...) positions directly to a specified record

(para. on variable files)

Indexed Files

An indexed 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 0 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/I 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, however, write_record only adds records. In addition, the following two operations manipulate keys:

read_key	obtains the key and length of the next record
seek_key	positions to the record with a given key or defines the key to be associated with a record to be added (by a subsequent write operation)

Table 5-3 shows the I/O operations that are permitted with each type of file.

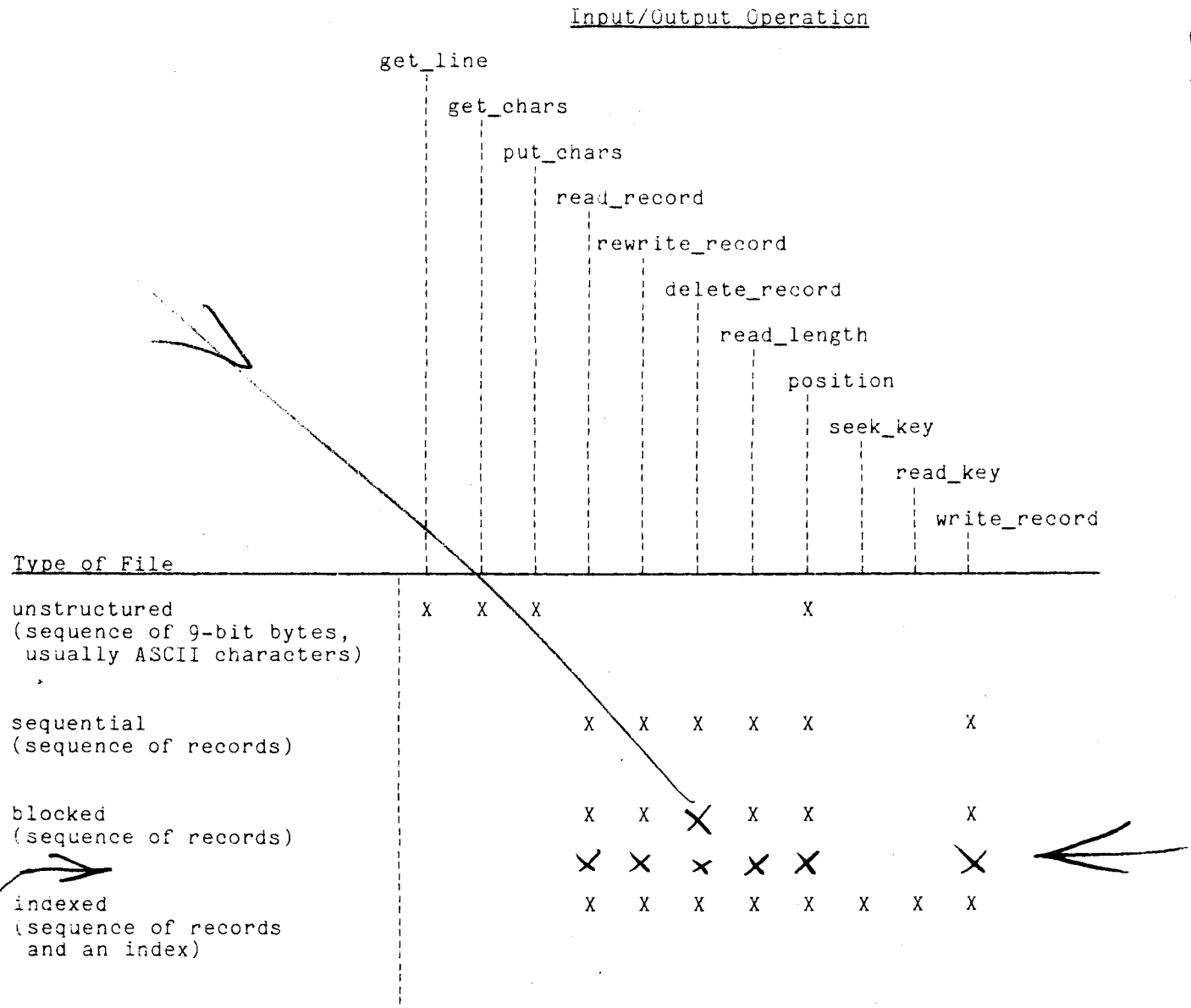
delete_record replaces a record with one of zero length. If the record is the last, the end of file position moves back to that of the deleted record.

Variable Files

A variable file contains a sequence of records. Each record is a string of 9-bit bytes. A record may be of zero length. Deletions differ from those in sequential files, in that a record position is logically deleted only in the end-of-file case.

All the I/O operations applicable to blocked files apply to variable files.

Table 5-3. File Types and Allowed Input/Output Operations



Each record is a string of bytes; a record may be of zero length. A blocked file has a characteristic maximum record length that is initially set by the user. For an indexed file, a key is a string of 0 to 256 ASCII characters, with no trailing blanks.

variable
(sequence of records)

Table 5-4. Compatible File Attachments

Opening Mode		File Type				
No.	Name	unstructured	sequential	blocked	variable	indexed
1	stream_input	x	1	1		1
2	stream_output	x3				
3	stream_input_output	x3				
4	sequential_input		x	x	x	x
5	sequential_output		x3	x3	x3	
6	sequential_input_output		x3	x3	x3	
7	sequential_update		2,3	x3	x3	x
8	keyed_sequential_input					x
9	keyed_sequential_output					x3
10	keyed_sequential_update					x3
11	direct_input					x
12	direct_output					x3
13	direct_update					x3

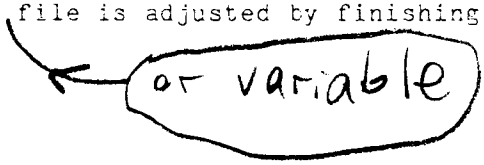
1. The structure of the file is ignored and everything in it is treated as data (including control words).

2. The file must be in the storage system.

3. This type of file is created by an output or update opening for the specified mode unless this feature is explicitly suppressed. Update openings never replace an existing file. (See the individual I/O module descriptions in Section III of the MPM Subroutines to see which control arguments are applicable.)

Name: vfile_adjust, vfa

The vfile_adjust command is used to adjust structured files left in an inconsistent state by an interrupted opening, or unstructured files in any state. For unstructured files a control argument must specify the desired adjustment. Otherwise, no control arguments are allowed. A sequential or blocked file is adjusted by truncation after the last complete record. An indexed file is adjusted by finishing the interrupted operation.



Usage

vfile_adjust path -control_arg-

where:

1.	path	is the pathname of the file to be adjusted.
2.	control_arg	must be specified only for unstructured files and is selected from the following:
	-set_nl	if the last nonzero byte in the file is not a newline character, a newline character is appended. The bit count of the file's last nonempty segment is then set to the file's last nonzero byte (which is now sure to be a newline character).
	-use_nl	the file is truncated after the last newline character.
	-set_bc	the bit count of the file's last nonempty segment is set to the last nonzero byte in that segment. Any components beyond it are deleted.
	-use_bc -n-	the file is truncated to the byte specified by the bit count of multisegment file component n. If n is not given, it is taken to be the last nonempty component.

Notes

See the description of the vfile_ I/O module (described in the MPM Subroutines) for further details. The adjust_bit_count command used with the character control argument is equivalent to vfile_adjust used with the -set_bc control argument, except that the latter only operates on a file that appears to be unstructured.

variable,

Name: vfile_status, vfs

The vfile_status command prints the apparent type (unstructured, sequential, blocked, or indexed) and length of files. For structured files, information about the state of the file (if busy) and the file version (unless current) is printed. The maximum record length is printed for blocked files. For indexed files, the following statistics are printed:

1. The number of records in the file, including zero length records
2. The number of nonnull records in the file, if different from the above
3. The total length of the records (bytes)
4. The number of blocks in the free space list for records
5. The height of the index tree (equal to zero for empty files)
6. The number of nodes (each 1K words, page aligned) in the index tree
7. The total length of all keys (bytes)
8. The number of keys (if different from record count)
9. The number of duplicate keys (if nonzero)
10. The total length of duplicate keys (if any)

Usage

vfile_status path

where path is the pathname of the segment or multisegment file of interest. If the entryname portion of the pathname denotes a directory, it is ignored. If no files are found for the given pathname, a message to that effect is printed. If the entry is a link, the information returned pertains to the entry to which the link points. The star convention is permitted.

Notes

Additional information may be obtained through the status command.

For variable files, the first & last allocated record positions are printed (if different from ~~zero~~ & eof-1, respectively).

Examples

Assume that the file foo is in the user's working directory. The command:

```
vfile_status  foo
```

might produce the following output:

```
     type:   unstructured
     bytes:   4993
```

if the file is unstructured,

or

```
     type:   sequential
     records:   603
```

if the file is sequential,

or

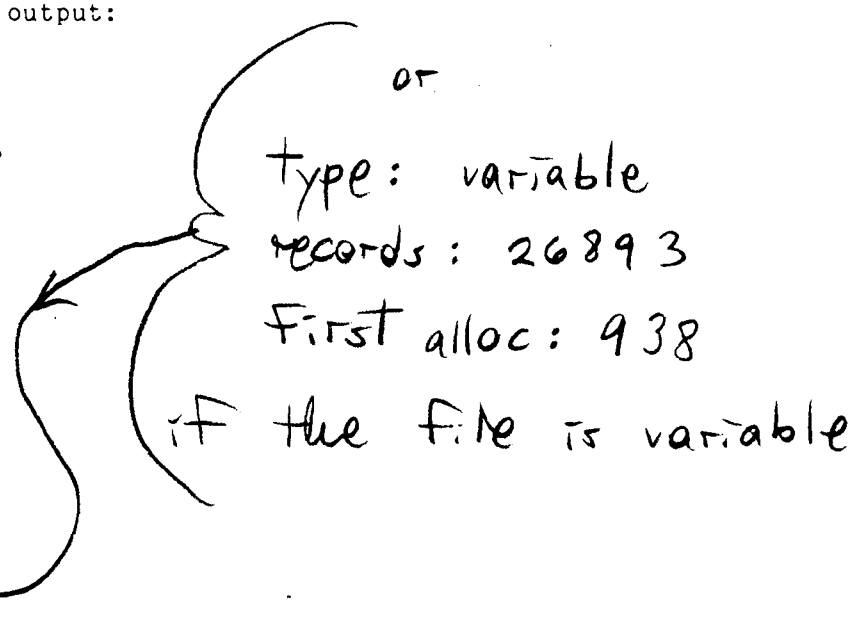
```
     type:   blocked
     records:   1200
     max recl:   7 bytes
```

if the file is blocked,

or

```
     type:   indexed
     records:   397
     state:   locked by this process
     action:   write in progress
     record bytes:   3970
     free blocks:   1
     index height:   2
     nodes:   3
     key bytes:   3176
```

if the file is indexed and a write operation has been interrupted in the user's process.



Name: vfile_status_

The vfile_status_ subroutine returns various items of information about a file supported by the vfile_ I/O module.

Usage

```
declare vfile_status_ entry (char(*), char(*), ptr, fixed bin(35));

call vfile_status_ (dir_name, entryname, info_ptr, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)
2. entryname — is the entryname of the file of interest. If the entry is a link, the information returned pertains to the entry to which it points. (Input)
3. info_ptr — is a pointer to the structure in which information is to be returned. (See "File Information" below.) (Input)
4. code — is a storage system status code. (Output)

File Information

The info_ptr argument points to one of the following self-describing structures, as determined by the type of the file (see "type" below):

```
dcl  1  uns_info           based (info_ptr),  /* structure for
     2  info_version       fixed bin,            unstructured files */
     2  type               fixed bin,
     2  bytes              fixed bin(34),
     2  flags              aligned,,
        3  pad1            bit(2) unal,
        3  header_present  bit(1) unal,
        3  pad2            bit(33) unal,
     2  header_id          fixed bin(35);
```

where:

1. info_version — identifies the version of the info structure; this must be set to 1 by the user. (Input)
2. type — identifies the file type and the info structure returned:
   1 unstructured
   2 sequential
   3 blocked
   4 indexed

   5 variable
3. bytes — gives the file's length, not including the header in bytes.

12.	nodes	is the number of single page nodes in the index.
13.	key_bytes	is the total length of all keys in the file in bytes.
14.	change_count	is the number of times the file has been modified.
15.	num_keys	is the total number of index entries, each associating a key with a record.
16.	dup_keys	is the number of index entries with nonunique keys, not including the first instance of each key.
17.	dup_key_bytes	is the total length of all duplicate keys in the file, as defined above.

(insert vbl_info stuff here)

Notes

The user must provide the storage space required by the above structures. Normally, space should be allocated for the largest info structure that might be returned, namely, the one for indexed files.

See the description of the vfile_ I/O module for further details.

```
dcl  1  vbl_info                 based (info_ptr),
                                 /*structure for variable files*/

     2  info_version             fixed bin,
     2  type                     fixed,
     2  records                  fixed (34),
     2  flags                    aligned,
        3  lock_status           bit (2) unal,
        3  pad                   bit (34) unal,
     2  version                  fixed,
     2  action                   fixed,
     2  first_nz                 fixed (34),
     2  last_nz                  fixed (34),
     2  change_count             fixed (35);
```

where: 1.-6. are the same as in the blk_info structure above.

7. first_nz is the position of the first allocated record (same as eof if none).

8. last_nz is the position of the last allocated record (-1 if none).

9. change_count is the same as in the indx_info structure above.

Name: vfile_

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

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system. See "Multics Input/Output System" and "File Input/Output" in Section V of the MPM Reference Guide for a general description of the I/O system and a discussion of files, respectively.

Attach Description

The attach description has the following form:

or variable

vfile_ path -control_args-

where:

1. path — is the absolute or relative pathname of the file.

2. control_args — may be chosen from the following:

   -extend — specifies extension of the file if it already exists. This control argument is only meaningful with openings for output or input_output; otherwise, it is ignored.

   -share -wtime- — allows an indexed file to be open in more than one process at the same time, even though not all openings are for input. (See "Multiple Openings" below.) The wtime, if specified, is the maximum time in seconds that this process will wait to perform an operation on the file. A value of -1 means the process may wait indefinitely. If no wtime is given, a default value of 1 is used.

   -blocked -n- — specifies attachment to a blocked file. If a nonempty file exists, n is ignored and may be omitted. Otherwise, n is used to set the maximum record size (bytes).

   -no_trunc — indicates that a put_chars operation into the middle of an unstructured file (stream_input_output) is permitted, and no truncation is to occur in such cases. Also prevents the truncation of an existing file at open and in stream_input_output openings causes the next byte position to be initially set to beginning of file.

   -append — in input_output openings, this causes put_chars and write_record operations to add to end of file instead of truncating when the file position is not at end of file. Also the position is initially set to beginning of file, and an existing file is not truncated at open.

-variable, -vbl specifies attachment to a variable file.

-header -n- for use with unstructured files, this control argument indicates that a header is expected in an existing file, or is to be created for a new file. If a header is specified, it contains an optional identifying number, which effectively permits user-defined file types. If n is given and the file exists, the file identifier must be equal to n; a new file takes the value of n, if given, as its identifier. The header is maintained and becomes invisible only with the explicit use of this control argument.

-old indicates that a new file is not to be created if an attempt is made to open a nonexisting file for output, input_output, or update.

-ssf restricts the file to a single segment. If specified, an attempt to open a multisegment file or to expand a file beyond a single segment is treated as an error. The file must not be indexed.

-dup_ok indicates that the creation of duplicate keys is to be permitted. The file must be indexed. (See "Duplicate Keys" below.)

The -extend, -append, and -no_trunc control arguments conflict; only one may be specified.

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

Opening and Access Requirements

All opening modes are supported. For an existing file, the mode must be compatible with the file type. (See "File Input/Output" in Section V of the MPM Reference Guide.) The mode must be compatible with any control arguments given in the attach description.

An existing file is not truncated at open if its safety switch is on and its bit count is nonzero.

If the opening is for input only, only read access is required on the file. In all other cases, rw access is required on the file.

Position Operation

An additional type of positioning is available with unstructured and blocked files that are open for input, input_output, or update. When the type argument of the iox_$position entry point is 2, this specifies direct positioning to the record or byte whose ordinal position (0, 1, 2, ...) is given. The zero position is just beyond the file header, if a header is present.

Write Operation

In blocked and sequential files open for update, this operation is supported. Its effect is to append a record to the file or replace the next record, depending on the next record position.

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.

In a blocked file, no record may be rewritten with a record whose length exceeds the maximum record length of the file. Attempting to do so causes the code, error_table_$long_record, to be returned.

Delete Operation

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

Deletions are not supported in blocked files. If the user attempts to delete a record in a blocked file, the code, error_table_$no_operation is returned.

Modes Operation

This operation is not supported.

Control Operation

The following orders are supported by the vfile_ I/O module.

read_position	add_key
seek_head	delete_key
set_wait_time	get_key
truncate	min_block_size
max_rec_len	reassign_key
	record_status
	set_file_lock

The five orders in the first column are described below. The remaining orders, documented in the vfile_ I/O module in the MPM Subsystem Writers' Guide, implement various features of indexed files that require somewhat more knowledge of internal file structure than is expected of most users.

or variable

set_wait_time

The set_wait_time order is accepted when the I/O switch is open and attached to an indexed file with the -share control argument. For this order the info_ptr argument must point to a structure of the following form:

```
dcl new_wait_time float based(info_ptr);
```

This order specifies a limit on the time that the user's process will wait to perform an order when the file is locked by another process. The interpretation of new_wait_time is the same as that described earlier for the wtime limit used with the -share control argument.

truncate

The truncate order is accepted when the I/O switch is attached to a nonindexed file open for input_output or update. The operation truncates the file at the next record (byte for unstructured files). If the next position is undefined, the code error_table_$no_record is returned.

- No info structure is required for this order.

max_rec_len

The max_rec_len order is accepted when the I/O switch is open and attached to a blocked file. The operation returns the maximum record length (bytes) of the file. A new maximum length can be set by specifying a nonzero value for the second argument. In this case the file must empty and open for modification, or the code error_table_$no_operation is returned.

For this order the info_ptr argument must point to a structure of the following form:

```
dcl 1 info          based (info_ptr),
    2 old_max_recl  fixed(21),   /*output*/
    2 new_max_recl  fixed(21);  /*input*/
```

3. Blocked File.
In general, the file's bit count and record count will not be correct. This condition is detected at a subsequent open, and either the file is automatically adjusted or (if the opening is input only) the code error_table_$file_busy is returned.

4. Indexed file.
In general, the bit counts of the file's segments will not be properly set, and the file contents will be in a complex intermediate state (e.g., a record, but not its key in the index, will be deleted). This situation is detected at a subsequent open or at the beginning of the next operation, if the file is already open with the -share control argument. Unless the opening is for input only, the file is automatically adjusted; otherwise, the code error_table_$file_busy is returned.

When an indexed file is adjusted, the interrupted operation (write_record, rewrite_record, delete_record, etc.), if any, is completed. For rewrite_record, however, the bytes of the record may be incorrect. (Everything else will be correct.) In this case, an error message is printed on the terminal. The user can rewrite or delete the record as required. The completion of an interrupted write operation may also produce an incorrect record, in which case the defective record and its key are automatically deleted from the file.

Any type of file may be properly adjusted with the vfile_adjust command (described in the MPM Commands), if an interrupted opening has occurred.

Inconsistent Files

The code error_table_$bad_file (terminal 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. A program accidentally modified some words in the file.

Obtaining File Information

The type and various statistics of any of the four vfile_ supported file structures may be obtained with the vfile_status command or vfile_status_ subroutine (described in the MPM Commands and Subroutines respectively).

5. Variable File.
Same as for indexed files.

Name: vfile_

The majority of the vfile_ I/O module documentation is in Section III of the MPM Subroutines. The information given here describes additional order calls for users of indexed files. These orders allow a greater degree of control in the areas of synchronization and separate record/index manipulation. They implement various features of indexed files that require somewhat more knowledge of internal file structure than is expected of most users.

min_block_size

The min_block_size operation determines the minimum size for blocks of record space that are subsequently allocated by write_record or rewrite_record operations (documented in the iox_ subroutine). The specification remains in effect for the duration of the current opening or until another call to this order is issued. The I/O switch must be attached to an indexed file open for output or update.

For this order, the info_ptr argument must point to a structure of the following form:

```
dcl 1 min_blksz_info based(info_ptr),
      2 min_residue    fixed bin(21),
      2 min_capacity   fixed bin(21);
```

where:

1. min_residue specifies the minimum unused capacity of a record block (bytes); i.e., the difference between the record's length and the maximum length it can attain without requiring reallocation. (Input)

2. min_capacity specifies the minimum total record capacity (bytes); i.e., the maximum length that the record can attain without requiring reallocation. (Input)

When the I/O switch is initially opened, both these parameters are set to zero.

The current implementation imposes the following constraints on allocated record blocks:

1. The minimum allocation is eight full words, including two header words for the block length and record length. The minimum nonnull record capacity is, therefore, 24 bytes.

2. The size of an allocated block is always an integral number of full words, i.e., a multiple of four bytes.

where:

1. input_key — indicates whether the key is given in the info structure. (Input)
   "0"b indicates that the index entry to be reassigned has as its key the current key for insertion. If undefined, the code error_table_$no_key is returned.
   "1"b indicates that the key_string argument defines the key portion of the index entry to be reassigned. If the key_string is not found in the index, the code error_table_$no_key is returned.

2. input_old_descrip — indicates whether the old descriptor is given in the info structure. (Input)
   "0"b indicates that the entry to be changed is associated with the current record. If the current record is undefined, the code error_table_$no_record is returned.
   "1"b indicates that the old_descrip argument defines the descriptor portion of the index entry to be changed.

3. input_new_descrip — indicates whether the new descriptor is given in the info structure. (Input)
   "0"b indicates that the specified index entry is to be reassigned to the current record. If the current record is undefined, the code error_table_$no_record is returned.
   "1"b indicates that the argument new_descrip is to supply the new value for the descriptor portion of the specified index entry.

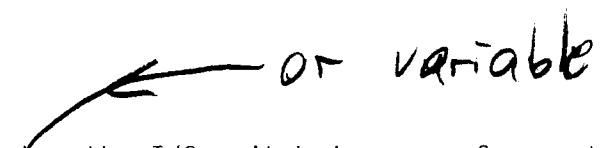
4. old_descrip — is used only if reassign_key_info.input_old_descrip equals "1"b. The entry that is reassigned is the first whose descriptor matches this value, among those index entries with the specified key. (Input)

5. new_descrip — is used only if reassign_key_info.input_new_descrip equals "1"b. This value replaces the old descriptor of the specified index entry. (Input)

6. key_length — same as in the add_key_info structure above. (Input)

7. key_string — if reassign_key_info.input_key equals "1"b, this argument defines the key for which the index entry with the specified descriptor is to be reassigned. (Input)

set_file_lock



The set_file_lock order is accepted when the I/O switch is open for output or update and attached to an indexed file with the -share control argument. For this order, the info_ptr argument must point to a variable of the following form:

```
dcl  set_lock_flag bit(2) aligned based(info_ptr);
```