MULTICS TECHNICAL BULLETIN MTB-269

To: Distribution

From: M. Asherman

Date: 03/30/76

Subject: vfile_ changes for release 4.0

INTRODUCTION

This MTB summarizes changes to the vfile_ I/O module which are proposed for MR4.0. Aside from a slight alteration to the specs for iox_$delete_record, these are functional extensions of indexed files.

NEW OPERATIONS

The changes include the following new control orders:

"min_block_size"	record manipulation
"record_status"	
"get_key"	index manipulation
"add_key"	
"delete_key"	
"reassign_key"	
"set_file_lock	synchronization
"set_wait_time"	

NEW FEATURES

Some of the more important new features are:

record pointers
record locks
separate (unkeyed) records
separate (multiple) keys
duplicate keys
shared sequential operations
parallel access on passive shared operations

REASONS AND IMPLICATIONS

The purpose of these changes is to provide a more powerful file manipulation capability upon which to construct data base systems.

Specific applications which should benefit from these changes are:

selective access to portions and collections of records

construction of permanent list structures

sharing under almost all circumstances

establishing arbitrary many to many associations

DETAILED PROPOSAL

The revisions described in MCR's 1560, 1596, 1615 and 1616 are documented in the following pages, along with draft MPM documentation for the remaining vfile_ changes for MR4.0. Note that the control orders "min_block_size" and "set_file_lock" have been further revised since the publication of their respective MCR's (1596 and 1616).

FUTURE PLANS FOR INDEXED FILES

Most of the interface changes and new features which are planned have been included in the 4.0 release. Some other extensions under consideration are:

1. "key_range" order — causes subsequent operations to see only a selected portion of the file's index.

2. "read_position" order — (already supported with non-indexed files)

3. Integration of system area_ package with vfile_'s record management logic to provide a general permanent area capability and greater flexibility in targeting record allocations.

The bulk of the remaining modifications which are planned deal with various performance enhancements, not user interface changes. Substantial improvements can definitely be achieved in this area, at least for many common special-case situations. Some of these are described briefly in MTB-258.

Name: vfile_status, vfs

The vfile_status command prints the apparent type (unstructured, sequential, blocked, or indexed) and length of storage system 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 non-zero)
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.

Name: iox_

If the file is open for direct_update and the deletion takes place, the current and next record positions are set to null. For keyed_sequential_update, the current and next record positions are set to the record following the deleted record or to end of file (if there is no such record).

Usage

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

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

where:

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

2.   code   is an I/O system status code. (Output)

Entry: iox_$detach_iocb

This entry point 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);
```

where:

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

2.   code   is an I/O system status code. (Output)

Name: vfile_status_

```
dcl  1  blk_info           based (info_ptr),  /* structure for
     2  info_version       fixed,                blocked files */
     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  max_rec_len        fixed(21);

dcl  1  indx_info          based (info_ptr),  /* structure for
     2  info_version       fixed,                indexed files */
     2  type               fixed,
     2  records            fixed(34),
     2  flags              aligned
        3  lock_status     bit(2) unal,
        3  pad             bit(34) unal,
     2  version            aligned,
        3  file_version    fixed(17) unal,
        3  program_version fixed(17) unal,
     2  action             fixed,
     2  non_null_recs      fixed(34),
     2  record_bytes       fixed(34),
     2  free_blocks        fixed,
     2  index_height       fixed,
     2  nodes              fixed,
     2  key_bytes          fixed(34),
     2  change_count       fixed(35),
     2  num_keys fixed(34),
     2  dup_keys fixed(34),
     2  dup_key_bytes fixed(34),
     2  reserved(1) fixed;
```

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

3. lock_status — if zero, indicates that the file's lock is not set; otherwise the file is busy

"01"b busy in caller's process
"10"b busy in another process
"11"b busy in a defunct process

4. records — is the number of records in the file, including those of zero length.

5. header_present — if set, indicates that an optional header is present.

6. header_id — contains the identification from the file's header, if present. Its meaning is user-defined.

7. bytes — gives the file's length, not including the header in bytes.

8. max_rec_len — is the maximum record length (in bytes) associated with the file.

9. version — identifies the version number of the file and its creating program.

10. action — if nonzero, indicates an operation in progress on the file:

-1 write in progress
-2 rewrite in progress
-3 delete in progress
+1 truncation in progress

11. record_bytes — is the total length of all records in the file in bytes.

12. free_blocks — is the number of blocks in the file's free space list for records.

13. index_height — is the height of the index tree (equal to zero if file is empty)

14. nodes — is the number of single page nodes in the index.

15. key_bytes — is the total length of all keys in the file in bytes.

16. non_null_recs — is a count, not including those of zero length, of the records in the file.

17. change_count — is the number of times the file has been modified.

18. num_keys — is the total number of index entries each associating a key with a record.

19. dup_keys — is the number of index entries with non-unique keys, not including the first instance of each key.

20. dup_key_bytes — is the total length of all duplicate keys in the file, as defined above.

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 tne vfile_ I/O module for further details.

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:

```
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.

-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.

-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 (See "Duplicate Keys" below). The file must be indexed.

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 control operations are supported by the vfile_ I/O module.

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

seek_head

The seek_head order is accepted when the I/O switch is open for keyed_sequential_input or keyed_sequential_update. For this order the info_ptr argument must point to a structure of the following form:

```
dcl  1  info            based (info_ptr),
     2  relation_type   fixed,
     2  n               fixed,
     2  search_key      char (0 refer (n));
```

The operation locates the first record with a key whose head has the specified relation with the given search_key. The next record position and (for keyed_sequential_update) the current record position are set to the record. If no such record exists, the code error_table_$no_record is returned.

The head of a record's key is the first n characters of the key, the key being extended by blanks if it has fewer than n characters. The allowed values for info.relation_type are:

0	head = search_key
1	head >= search_key
2	head > search_key

read_position

The read_position order is accepted when the I/O switch is open and attached to a nonindexed file. The operation returns the ordinal position (0, 1, 2, ...) of the next record (byte for unstructured files), and that of the end of file, relative to the file base. The file base is just beyond the header, if a header is present.

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

```
dcl 1 info            based (info_ptr)
      2 next_position  fixed(34),  /*output*/
      2 last_position  fixed(34);  /*output*/
```

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*/
```

control operation: "min_block_size"

This operation determines the minimum size for blocks of record space which are subsequently allocated by write_record or rewrite_record operations. 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(21),

          2  min_capacity fixed(21);
```

where:

1. min_residue (Input)

   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.

2. min_capacity (Input)

   specifies the minimum total record capacity (bytes), i.e. the maximum length which the record can attain without requiring reallocation.

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

control operation: "record_status"

This operation returns information about a specified record in an indexed file, and optionally permits the user to manipulate the record's lock and/or to allocate an empty record.

An argument is provided which permits one to entirely avoid using the index in accessing and creating records (see Note below).

The I/O switch must be open and attached to an indexed file. The next record position is not altered or used by this operation. The current record position is always set to the record referenced.

The I/O switch must be open for output or update in order to lock, unlock or create a record.

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

```
     dcl   1   rs_info based(info_ptr) aligned,
           2   version fixed,
           2   flags aligned,
                3 lock_sw bit(1) unal,
                3 unlock_sw bit(1) unal,
                3 create_sw bit(1) unal,
                3 locate_sw bit(1) unal,
                3 mbz1 bit(32) unal,

           2   record_len fixed(21),
           2   max_rec_len fixed(21),
           2   record_ptr ptr,
           2   descriptor fixed(35),
           2   mbz2 fixed;

     dcl   rs_info_version_1 static internal fixed init(1);
```

where:

1. version (Input)

   is provided for compatibility with possible future versions of this info structure. The user should set this argument to rs_info_version_1.

2. lock_sw (Input)

   if set to "1"b an attempt is made to lock the specified record within the wait time limit given at attachment or subsequently set via the "set_wait_time" order. Possible error codes are those returned by set_lock_$lock , as well as the code error_table_$no_roon_for_lock, which is returned if the allocated record block is too small to contain a lock. (see section entitled "Records Locks").

3. unlock_sw (Input)

   if set to "1"b an attempt is made to unlock the record. Possible error codes are those returned by set_lock_$unlock and the code error_table_$no_roon_for_lock. If both lock_sw and unlock_sw are set to "1"b, the locking takes place first and determines the resultant error code. (This permits one to clear an invalid lock in a single operation.)

4. create_sw (Input)

   if set to "1"b causes a new record to be allocated using the record_len and max_rec_len arguments as input parameters. The contents of the record are set to zero, and its lock is set in the same operation if lock_sw = "1"b. Depending upon the setting of locate_sw, the new record may be entered into the index. If locate_sw = "0"b the current key for insertion is added to the index as a key for the new record. Otherwise, no index entry is created and the key for insertion becomes undefined.

5. locate_sw (Input)

   "0"b if create_sw also = "0"b, this indicates that the current record position defines the record of interest. Otherwise, the current key for insertion is used. If the relevant position designator is undefined, the code error_table_$no_record or error_table_$no_key is returned, whichever is appropriate.

   "1"b if create_sw = "0"b this indicates that the descriptor argument is an input parameter defining the location of the record of interest. If create_sw = "1"b this causes the new record to be created without a key.

6. mbz1 and mbz2 (Input) must be set to zero by the user

7. record_len (Output)

(if create_sw = "1"b this argument is input) gives the record's length in bytes.

8. max_rec_len (Output)

if create_sw = "1"b this argument is input and overrides any minimum block size specification which may currently be in effect (see "min_block_size" order). The returned value gives the maximum length which the record can attain (bytes) without requiring reallocation. In the current implementation, records are allocated in blocks whose record capacity is a multiple of four bytes greater than or equal to 24. when this argument is used as an input parameter, the resultant maximum record length is smallest number greater than or equal to max_rec_len which corresponds to an implemented (non-zero) block size.

9. record_ptr (Output)

points to the first byte of the allocated record, or is set to null if no allocated record exists.

10. descriptor (Output)

is a process independent locator for the specified record. This value is used as an input argument when locate_sw = "1"b and create_sw = "0"b. The actual structure of each descriptor is as follows:

```
dcl 1     descrip_struct based (addr(descriptor)) aligned,

      2    comp_num fixed(17) unal,

      2    word_offset bit(18) unal;
```

where:

a. comp_num is the msf component number of the segment containing the record.

b. word_offset is the word offset of the block of storage containing the allocated record, relative to the base of its file component.

A zero descriptor designates an unallocated (zero-length) record.

Descriptors may also be arguments to the orders "add_key", "delete_key", "reassign_key_, and get_key". Note that at any given time within a single file each record is uniquely located by its descriptor, which remains valid only for the life of a single allocation.

Note: If locate_sw is set to "1"b, the resultant current record position moves "outside" of the index in the sense that there is no key associated with the current record. This situation may also arise after using the "delete_key" operation.

When this is the case, a subsequent rewrite_record or delete_record operation behaves differently from the usual case. The difference is that no corresponding index entry is changed or deleted to reflect the change to the record.

Extreme caution must be exercised when using the control operations which take a descriptor as an input argument, especially in a shared environment. The user is responsible for insuring that previously obtained descriptors and pointers are still valid when they are used. Also, pains must be taken to maintain the index in a consistent state, i.e., each index entry should designate a valid record if a record reference may be attempted.

control operation: "get_key"

This operation returns both the key and the record descriptor for the next record in an indexed file.

The I/O 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_$end_of_info is returned. If the next record position is undefined, the code error_table_$no_record is returned. The next record position is unchanged, and the current record position is set to the next record if the operation is successful; otherwise, the current record position is set to null.

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

```
dcl 1  get_key_info based (info_ptr),

    2  mbz fixed,

    2  descriptor fixed(35),

    2  key_length fixed,

    2  key_string char(0 refer(get_key_info.key_length));
```

where:

0. mbz (Input) must be set to zero

1. descriptor (Output)

   is the record locator for the next record. This value may be used as an input argument to the control operations "add_key" "delete_key", "reassign_key", and "record_status", (see Note below)

2. key_length (Output)

   is the length of the key at the next record position.

3. key_string (Output)

   is the next record's key

Note: The interpretation of the descriptor argument as a record locator is not mandatory, since the operations "add_key" and "reassign_key" permit the user to set the descriptor portion of an index entry to an arbitrary 36 bit value.

In such cases the descriptor itself may be thought of as a one-word record which is "read" by the "get_key" operation.

control operation: "add_key"

This operation creates a new index entry with a given key and record descriptor.

The I/O switch must be open for direct_output, direct_update, keyed_sequential_output, or keyed_sequential_update. Current and next record positions are unchanged.

Associations may be formed between any number of keys and a single record via this operation. Duplicate keys may be added if the file was attached with the -dup_ok option, or if the file already contains duplications; otherwise, the code error_table_$key_dup is returned. (See section entitled "Duplicate Keys").

Note that this operation, as well as the orders "delete_key", "reassign_key", and "get_key", do not reference the length or contents of a record. This permits one to avoid the use of actual records altogether in any given indexed file.

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

```
dcl 1     add_key_info based(info_ptr),
          2 flags aligned,
               3 input_key bit(1) unal,
               3 input_descrip bit(1) unal,
               3 mbz bit(34) unal, /* must be zero */
          2 descriptor fixed(35),
          2 key_len fixed,
          2 key_string char(0 refer(add_key_info.key_len));
```

where:

1. input_key (Input)

   "0"b indicates that the current key for insertion is the new key. If this value is undefined, the code error_table_$no_key is returned.

   "1"b indicates that the key to be added is the key_string contained in this info structure.

2. input_descrip (Input)

   "0"b indicates that the current record defines the new descriptor. If the current record is undefined, the code error_table_$no_record is returned.

   "1"b indicates that the user supplied descriptor in this info structure is the new descriptor.

3. descriptor (Input)

This argument is used only if the variable input_descrip is set to "1"b. The descriptor is stored into the index together with its associated key. Any 36 bit quantity may be supplied, although in general this number will have been previously obtained via the "record_status" or "get_key" control operations. Descriptors are used by operations which reference the contents or length of a record, in order to obtain the record's address.

4. key_len (Input)

is the length of the key_string. Keys must be between 0 and 256 chars, inclusive.

5. key_string (Input)

is used only if add_key_info.input_key is set to "1"b. It defines the key to be added to the index with the appropriate record descriptor.

/

control operation: "delete_key"

This operation deletes a specified index entry.

The I/O switch must be open for direct_update or keyed_sequential_update. The current and next file positions are left unchanged, with the following exception: if the deleted index entry is at the next record position, then the next record position is advanced to the following index entry, or becomes undefined in direct openings.

For this order the info_ptr argument may be null, or may point to a structure of the following form:

```
dcl 1     delete_key_info like add_key_info based (info_ptr);
```

where:

1. input_key (Input)

   "0"b indicates that the key associated with the current file position defines the key of the index entry which is to be deleted. If current position is undefined or outside the index (e.g., after deleting the current key of the current record), the code error_table_$no_key is returned.

   "1"b indicates that the user_supplied key_string defines the key of the entry to be deleted. If no such key is found, the code error_table_$no_key is returned.

2. input_descrip (Input)

   "0"b indicates that the index entry to be deleted 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 entry to be deleted is associated with the user_supplied descriptor. If no such entry exists, the code error_table_$no_record is returned.

3. descriptor (Input)

   is used only if delete_key_info.input_descrip="1"b. The entry which is deleted is the first whose descriptor matches this value, among those entries with the specified key.

4. key_len (Input)

   same as in "add_key"

5. key_string (Input)

when delete_key_info.input_key="1"b, this defines the key for which the index entry with the specified record descriptor is to be deleted.

If the info_ptr argument is null, the index entry at the current file position is deleted, i.e., the effect is the same as that of setting both arguments, input_key and input_descrip, to "0"b.

control operation: "reassign_key"

This operation causes the descriptor portion of a specified index entry to be replaced with a given value.

The I/O switch must be open for direct_update or keyed_sequential_update. The file position designators are not changed.

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

```
dcl 1     reassign_key_info based(info_ptr),
          2    flags aligned,
               3     input_key bit(1) unal,
               3     input_old_descrip bit(1) unal,
               3     input_new_descrip bit(1) unal,
               3     mbz bit(33) unal,
          2    old_descrip fixed(35),
          2    new_descrip fixed(35),
          2    key_len fixed,
          2    key_string char(0 refer(reassign_key_info.key_len));
```

where:

1. input_key (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 (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 (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 (Input)

   is used only if reassign_key_info.input_old_descrip="1"b. The entry which is reassigned is the first whose descriptor matches this value, among those index entries with the specified key.

5. new_descrip (Input)

   is used only if reassign_key_info.input_new_descrip="1"b. This value replaces the old descriptor of the specified index entry.

6. key_len (Input)

   same as in "add_key"

7. key_string (Input)

   when reassign_key_info.input_key="1"b, this argument defines the key for which the index entry with the specified descriptor is to be reassigned.

control operation: "set_file_lock

The order "set_file_lock" 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 structure of the following form:

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

This operation causes the file to be locked (if possible within the wait-time limit) or unlocked, depending on the user's setting the first bit of info_ptr->set_lock_flag to "1"b or "0"b, respectively.

The possible error codes are those returned by set_lock_$lock and set_lock_$unlock, excepting the code error_table_$invalid_lock_reset, which is not treated as an error.

The second bit of set_lock_flag indicates the class of operations which are to be excluded by locking the file. If "0"b only operations which alter the file are excluded; passive operations do not detect this state. Otherwise, all index referencing operations are excluded. In any case, the exclusion only applies to operations outside the current opening.

control operation: "set_wait_time"

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

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

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

Multiple Openings

It is possible to have or attempt to have multiple openings of the same file, that is, to have two or more open I/O switches attached to the same file. These switches might be in the same process or in different processes. With respect to the effects of multiple openings, the various opening modes can be divided into four classes (explained below). Multiple openings in which the opening modes are in more than one class are invalid, as are multiple openings within certain classes. The vfile_ I/O module prevents some cases of multiple opening. If a multiple opening is detected, error_table_$file_busy is returned by the open operation. In cases where an invalid multiple opening does occur, I/O operations will cause unpredictable errors in the processes involved, and the contents of the files may be damaged.

The classes of multiple openings are:

1. Openings for input without the -share control argument.
Any number of openings in this class are allowed. The existence of an opening in this class never causes damage to the file. When this class of opening is attempted, the existence of all class 2 and 3 openings and some class 4 openings will be detected for structured files.

2. Openings for output or input_output without the -extend control argument.
Only one opening is allowed. The existence of another opening is never detected when this class of opening is attempted. The file is simply replaced by an empty file of the appropriate type. If the file was already open with an opening of any class except class 1, the contents of the new file will probably be damaged.

3. Openings for update without the -share control argument and for output or input_output without the -share control argument and with the -extend control argument.
Only one opening of this class is allowed. For structured files, multiple openings within the class are detected. An invalid multiple opening involving an opening of this class and other openings of class 4 may be detected. If not, the only effect is that the class 3 opening locks the file for the entire opening.

4. Openings with the -share control argument.
[redacted] Any number of openings of this type are allowed. When a process performs an [redacted] on the file, the file is locked. Other processes attempting an operation while the file is locked will wait up to the limit specified by wtime in the -share control argument. If the operation is not carried out because of the wtime limit, the code error_table_$file_busy is returned.

update

There are two codes that pertain only to class 4 openings: error_table_$asynch_deletion and error_table_$asynch_insertion. The first is returned by the read_record, read_length, and rewrite_record operations when a record located by a seek_key operation has been deleted (by an operation in some other opening). The second is returned by write_record when a record with the key for insertion (defined by a seek_key operation) has already been inserted (by some other opening).

Interrupted Openings

If a process opens a file and terminates without closing the file, the file may be left in an intermediate state that prohibits normal I/O operations on the file. The exception is openings for input only. The details depend on the particular type of file as follows:

1. Unstructured file.
   In general, the bit count of the file's last segment will not be properly set. This condition is not detected at subsequent openings, and part of the file's contents may be overwritten or ignored.

2. Sequential file.
   In general, certain descriptors in the file and the bit count of the file's last segment will not be properly set. This condition is detected at a subsequent open, and the code error_table_$file_busy is returned.

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 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 without the -share control argument, the file is automatically adjusted. If this situation is detected by an opening for input without the -share control argument, the code error_table_$file_busy is returned. Opening the file for update will properly adjust the file.

   When an indexed file is adjusted, the interrupted operation (write_record, rewrite_record, or delete_record), 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 any 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).

Record Locks:

This feature pertains only to indexed files. Record locks provide a basis for synchronizing concurrent access at the individual record level. The setting and clearing of record locks is explicitly controlled by the user via the "record_status" order.

when the capacity of an allocated record block exceeds its contents by at least four bytes, the last word of the block is treated as a record lock. A non-zero lock identifies the process which set it. The user can insure that record allocations leave room for a lock by using the "min_block_size" order with a residue specification of at least four bytes.

All operations which reference the length or contents of an existing record (e.g., seek_key, but not "seek_head") also check the record's lock (if one exists). If the record is not locked, the operation proceeds normally. Otherwise, the returned error code reflects the state of the lock, indicating that the contents of the record may be in an inconsistent state. In this case, if the operation does not explicitly involve changing the file, it proceeds normally and the returned code is one of the following:

1. error_table_$record_busy

   if the record is locked by a live process.

2. error_table_$lock_is_invalid

   if the record's lock is set, but not by an existing process.

Attempting a rewrite_record or delete_record operation on a record locked by another process has not effect other than to return the code error_table_$record_busy (file is unchanged). If the lock is invalid, these operations return the code error_table_$invalid_lock_reset and zero the lock, or if the lock was set by the caller, the code returned is error_table_$locked_by_this_process; in either case the operation is otherwise successful.

when a record which is locked by the user's process is rewritten, its lock remains set, so long as the minimum block size specification currently in effect is such as to leave enough room for a record_lock.

Duplicate Keys:

By default vfile_ prevents the user from associating a single key with more than one record in the same indexed file. This restriction is removed when the -dup_ok attach option is used or if the file's statistics indicate that duplicate keys are already present.

Duplicate keys can be created via either the write_record operation or the "add_key" control order. When duplications are permitted, the key for insertion is defined as the key of the current record, if it exists.

With this extension, the notion of an "index entry" becomes more basic than that of a single key in the index. An index entry is an association between a string of characters (key) and a number (record descriptor).

Index entries are ordered by key. within multiple occurrences of the same key, the order is identical to the order in which the entries were created. A seek_key or "seek_head" operation locates the first instance of a set of duplicate keys. A write_record operation advances the file position beyond the last instance of the key for insertion, if the key already exists in the index.

The next record position is best thought of as corresponding to the next index entry. Operations which can advance the next record position (read_record, rewrite_record, position skip) permit one to locate intermediate instances of duplicate keys.