To: Distribution

From: Larry Johnson, Ross Klinger,
Janice Phillips, and Bill Silver

Date: August 27, 1976

Subject: Tape Interface to IOI

INTRODUCTION

This memorandum describes a proposed new internal tape interface, tape_ioi_ (TAPE_IOI). An interface like TAPE_IOI was first proposed in MTB-051, "New Tape DCM". TAPE_IOI will provide the interface between tape I/O modules and the I/O Interfacer (IOI).

This memorandum has been written for readers with varying levels of interest in TAPE_IOI. Each section becomes successively more detailed. Reading just one or two sections should give a reader a general overview of TAPE_IOI. The memorandum as a whole is intended to serve as a complete functional specification of TAPE_IOI. An outline of the sections contained in this memorandum is given below:

OVERVIEW
- Implementation Plans
- What TAPE_IOI Does
- Why TAPE_IOI is Needed
- Design Criteria

TAPE_IOI CONCEPTS
- Managing the IOI Workspace
- Tape I/O Operations and Primitives
- Buffer States
- Block Modes
- Data Lengths and Special Length Processing
- Error Recovery
- Status Information
- Interface to RCP

SUMMARY
- List of Capabilities
- List of Entry Points

APPENDICES
- Appendix A: Subroutine Interface Documentation
- Appendix B: Tape Order Commands
- Appendix C: TAPE_IOI Modes
- Appendix D: Result Index Summary
- Appendix E: Sample Scenarios

Multics Project internal working documentation. Not to be reproduced or distributed outside the Multics Project.

OVERVIEW

This section discusses plans for implementing TAPE_IOI, what TAPE_IOI does, why we need it, and how it was designed. In general, what TAPE_IOI does is to replace the interface between tape I/O modules and IOI that is now provided by the program tdcm_ (TDCM). The TDCM interface should be replaced because it has many deficiencies. TAPE_IOI was designed to correct those deficiencies.

Implementation Plans

TAPE_IOI will play an important part in the improvement of the Multics tape facility. See MTB-109 for an overall view of the future Multics tape facility. The implementation of TAPE_IOI is a necessary first step in the eventual implementation of a true raw tape I/O module. Other standard Multics tape I/O modules (tape_mult_, tape_ansi_, tape_ibm_, ntape_) will be improved by rewriting them to call TAPE_IOI rather than TDCM. None of the "ios_" tape dims (tape_, nstd_) will be converted to use TAPE_IOI.

The immediate plans for TAPE_IOI are to approve the TAPE_IOI interface design presented in this memorandum. Then implementation design and actual implementation of TAPE_IOI will begin. The reimplementation of tape I/O modules using TAPE_IOI may also be done in parallel with the TAPE_IOI implementation. One tape I/O module, probably tape_mult_, will definitely be converted to call TAPE_IOI.

What TAPE_IOI Does

The primary users of TAPE_IOI will be tape I/O modules. (TAPE_IOI will also be used by RCP in ring 1 to process tape labels.) In order to understand the role of TAPE_IOI in the overall scheme of tape processing, consider what happens when a user executes a command that involves tape processing. Several levels of I/O subsystems are involved. The command procedure calls IOX. IOX calls the specified tape I/O module. The tape I/O module will call TAPE_IOI. TAPE_IOI will call IOI. IOI (and other programs in ring 0) will perform the actual tape I/O. Figure 1 shows the relationship of TAPE_IOI to the other I/O subsystems involved in tape processing.

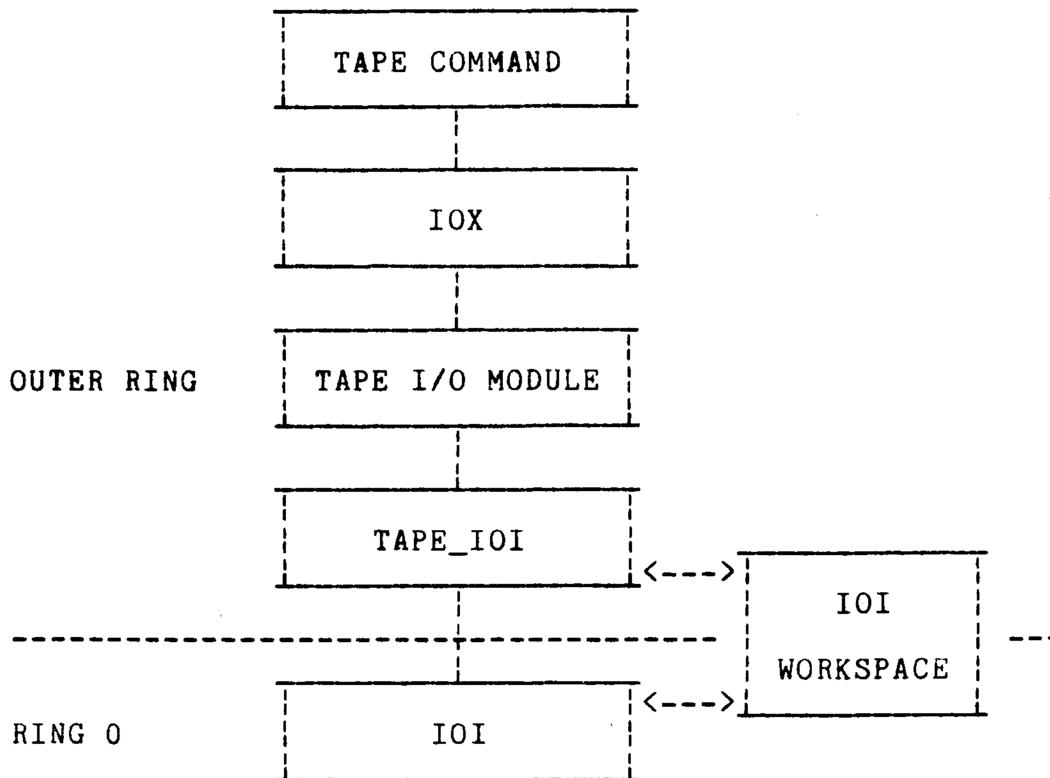

Figure 1: Tape Processing Subsystems

IOI is the Multics supervisor interface for all user peripheral I/O. It is a primitive, low level interface that allows user programs to have complete control over physical devices. Programs that call IOI must understand and deal with devices at the hardware level.

Interfacing with IOI involves more than just calling IOI. It also involves managing the IOI workspace. The IOI workspace is a segment that contains all of the IOI status queues, channel programs, and I/O buffers needed to perform tape I/O.

The tape processing functions described above are currently performed by TDCM. Before IOI was developed, TDCM resided in ring 0 and was the Multics supervisor interface for tape I/O. All tape I/O modules had to interface with TDCM. When IOI became available, TDCM was moved into the user ring and was rewritten to interface with IOI. The interface to TDCM, however, remained basically unchanged.

Why TAPE_IOI is Needed

An interface like TAPE_IOI is needed and is useful because the functions it provides are common to all tape I/O modules. The development and use of TAPE_IOI does not in any way prohibit a user from developing some other tape interface to IOI. For example, it is expected that all tape T&D programs will continue to interface directly with IOI. However, in order to make tape I/O modules interface directly with IOI, a large amount of complex and hardware-dependent code would have to be added to each one. It is hoped that the interface provided by TAPE_IOI will be so complete, efficient, and useful that it will not be necessary to bypass or replace it.

The question of whether or not we need an intermediate interface between tape I/O modules and IOI is really moot since we already have such an interface in TDCM. The important question is: "Why should TDCM be replaced?" The answer, simply, is that a better interface can be provided. TDCM has many deficiencies, none of which can be remedied without changing the TDCM interface. The deficiencies of TDCM are listed below in three groups. The first group consists of tape I/O capabilities that cannot be performed via TDCM. The second group consists of performance problems inherent with TDCM. The third group consist of facilities that TDCM does not provide, but could and should provide in order to make tape I/O modules simpler and easier to write and understand.

FACILITIES TDCM DOES NOT PROVIDE

1. TDCM does not allow a caller to issue tape order commands that use data. This means that writing special End-of-File records, reading and writing of device status, etc., cannot be performed via TDCM.

2. A caller of TDCM cannot obtain all of the hardware status information that is available. The right-hand 52 bits of IOM status is not available. TDCM does not return special status at all.

3. The element size supported by TDCM is one word (36 bits). It should be one character (9 bits). One problem with this word orientation is that TDCM cannot properly write records that have a length that is not an integral number of words. Also, TDCM cannot correctly return the length of a record that is not an integral number of words.

4. A caller of TDCM cannot specify the channel instruction field to be used in tape read operations. For certain models of tape drives, this field can be used to specify that automatic hardware error retry is to be performed.

5. TDCM cannot read or write records that are longer than 16,384 characters (4096 words).

TDCM PERFORMANCE PROBLEMS

1. The main TDCM performance problem is that it must copy all input and output data. All input and output data is processed by the tape hardware in the IOI workspace segment. All input and output data is processed by tape I/O modules in another work segment called a tseg (TSEG). The data copy performed by TDCM involves moving data between the IOI workspace segment and the TSEG.

2. TDCM cannot maintain continuous I/O on a tape drive. Once TDCM initiates I/O for a set of buffers, it must wait for the completion of I/O for all of these buffers before it can initiate any more I/O. This means that tape I/O must terminate at least once for each set of buffers. If TDCM initiates I/O for a set consisting of more than one buffer, it will not be able to process the first buffer of this set until the I/O has completed for the last buffer of the set. This means that the caller of TDCM must wait to process data that is already available.

3. When reading variable length records, the above performance problem is replaced by an even more serious performance problem. If a caller wants to know the length of an input record (and callers do want to know this when reading variable length records) then TDCM can process only one buffer at a time. This restriction is, in truth, forced on TDCM by restrictions in the ring 0 I/O facilities. These ring 0 restrictions are being removed in conjunction with the development of TAPE_IOI.

4. The method used by TDCM to define buffers in the TSEG results in the restriction that only one buffer may be processed at a time if that buffer is larger than 1040 words.

FACILITIES TDCM SHOULD PROVIDE

1. Most tape I/O modules implement read-ahead and write-behind. This allows the tape I/O module to overlap its processing of data with the actual hardware reading and writing of data. This common tape I/O module function should be provided by an interface like TDCM.

2. TDCM provides two levels of status information. The first, and highest level, just indicates whether or not an I/O operation has completed, and if so, whether it has completed successfully. The second, and lower level, consists of raw hardware status. Each tape I/O module must interpret this raw hardware status. An interface like TDCM should interpret this status and return it to its caller in a more useful form. It should also return all available raw hardware status.

3. Every tape I/O module performs its own error recovery. An interface like TDCM should be able to perform the simple and straightforward error recovery procedures that are common to many tape I/O modules.

4. An interface like TDCM should allow a caller to decide when to block. It should also do the blocking for a caller if the caller does not want to do it. TDCM does not give its callers a choice and always does the blocking. The ability of a tape I/O module to control blocking means that tape I/O modules and tape application programs could be developed that overlap the processing of more than one tape drive.

5. Tape I/O modules should have more control over their interface to the Resource Control Package (RCP). TDCM interfaces directly with RCP and effectively prohibits any tape I/O module from calling RCP. Interfacing with RCP should be done only as an option for those callers that do not want to do it themselves.

6. Tape I/O modules should be able to perform a tape order command (such as backspace record) several times with one call. TDCM limits its callers to 10. This is an unnecessary and annoying restriction.

7. Tape I/O modules are called by IOX which requires that all string and record lengths be expressed in characters. Tape I/O modules should not have to deal with an interface like TDCM in terms of words.

Design Criteria

The considerations that motivated the design of TAPE_IOI involved the deficiencies of TDCM listed above. The goal of the design was to develop an interface that is better than TDCM. It was clear that solving all TDCM problems required a totally new interface. The main design problem was what should this new interface be like. Intuition and experience with TDCM and tape I/O modules guided the preliminary specification. However, many technical decisions had to be made before the TAPE_IOI interface described in this memorandum was defined. In order to make these decisions, three design criteria, really guiding principles, were established. These design criteria are listed below in order of importance.

1. A caller of TAPE_IOI should be able to perform every non-privileged I/O function allowed by the tape controller. All physical status information should be available. TAPE_IOI should not perform any tape I/O operations unless told to do so.

2. TAPE_IOI should be efficient. Not only should each TAPE_IOI entry point be as efficient as possible, but a tape I/O module that calls TAPE_IOI must be able to efficiently perform its "critical path" functions.

3. The TAPE_IOI interface should be as simple and as useful as possible. Basic tape processing functions should be easy to perform. Functions which are common to several tape I/O modules should be performed by TAPE_IOI and thus not duplicated in each.

As was the case with the design, any analysis of the TAPE_IOI interface must be done with consideration of the above design criteria. The task now is to evaluate the design criteria themselves and to make sure that the proposed TAPE_IOI interface really meets them.

TAPE_IOI CONCEPTS

This section discusses some of the basic concepts of TAPE_IOI. These concepts represent the theoretical basis for the design of the TAPE_IOI interface.

Managing the IOI Workspace

A major part of the task of interfacing with IOI involves managing the IOI workspace. The IOI workspace is a segment that is created by IOI when the tape drive is attached. Only the attaching process has access to this segment.

In order to understand how the IOI workspace is used, it is necessary to know what data it contains. Since the implementation of TAPE_IOI has not yet been designed, it is not possible to present a detailed and accurate description of the data TAPE_IOI will keep in the IOI workspace. Such a description would not, in general, be interesting anyway. An important concept of TAPE_IOI is that a caller never has to know the structure and format of the data kept in the IOI workspace. What is interesting in terms of understanding TAPE_IOI, and what can be described now, are the types of data kept in the IOI workspace by TAPE_IOI. These types of data are listed below and are shown in Figure 2.

Channel Programs: The channel programs that actually perform the tape I/O must be in the IOI workspace. TAPE_IOI builds and updates these channel programs as needed depending upon the I/O requests of the caller.

IOI Status Queues: Status information detailing the result of I/O operations is returned to TAPE_IOI in status queues. These status queues must be in the IOI workspace. TAPE_IOI sets up and interrogates these status queues. IOI fills them in.

TAPE_IOI Information: Although this is an implementation design consideration, it is possible and quite probable that TAPE_IOI will keep most of its internal information in the IOI workspace.

I/O Buffers: Reading and writing physical records involves the use of buffers. These buffers are allocated in the IOI workspace by TAPE_IOI as requested by the caller. Each buffer may contain one and only one physical record at a time. Each buffer is identified by a pointer that references the beginning of the buffer. This feature of TAPE_IOI allows a caller to directly process input and output data in the IOI workspace.

Caller Work Area: TAPE_IOI will allow a caller to allocate a work area at the end of the IOI workspace. This work area may be allocated only after all I/O buffers have been allocated. The maximum size of this work area will be the number of unused words at the end of the last IOI workspace page containing allocated I/O buffers. No additional IOI workspace pages will be used for this area. This optional work area can be used to keep important data needed by the caller, for example, an IOX open data block. By placing this data in the IOI workspace (which is often wired) fewer page faults will be generated by references to this data.

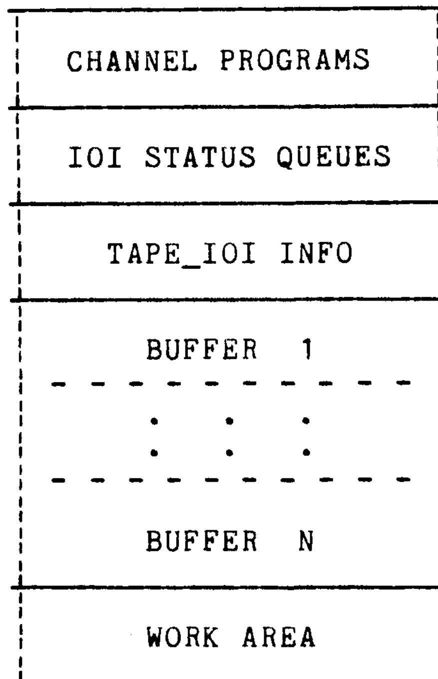

Figure 2: Possible Organization of IOI Workspace

TAPE_IOI provides entry points that allocate and deallocate I/O buffers and the caller work area. These allocation and deallocation entry points are simple and straightforward but not necessarily powerful. They are perfectly suited for the standard tape I/O modules. These I/O modules need to allocate, just once, a set of buffers all having the same length. Dynamic allocation and deallocation of buffers of different lengths can be performed by making a series of calls to these entry points.

TAPE_IOI is responsible for maintaining the size of the IOI workspace segment. As buffers are allocated, TAPE_IOI will increase the size of the IOI workspace segment up to the maximum size allowed by RCP. If buffers are deallocated, then TAPE_IOI will decrease the size of the IOI workspace segment accordingly. TAPE_IOI will always make sure that the IOI workspace segment consists of the fewest possible pages. This is important because all of the pages of the IOI workspace segment will be wired whenever any I/O is in progress.

Tape I/O Operations and Primitives

Designing the TAPE_IOI interface involved analysing all of the basic tape I/O operations and the primitive functions that comprise them. The three basic tape I/O operations are listed below:

reading
writing
order commands

Each of the basic tape I/O operations can be defined as a combination of primitive functions. These primitive functions are listed below in alphabetical order. Although the descriptions below are orientated primarily toward reading and writing, most of these primitive functions apply to order commands as well.

Allocate: Allocate a buffer to be used for I/O.

Check: Check to see if a buffer is ready for processing. In order to be ready for processing, I/O being performed on the buffer must be completed.

Deallocate: Deallocate a buffer. The buffer can no longer be used for I/O.

Processing: Process the data in the buffer. This is not a TAPE_IOI function, but rather a function to be performed by the caller. For reading, this means copying data out of the buffer. For writing, this means copying data into the buffer.

Queue: Queue I/O for this buffer.

Figure 3 shows how each of the three basic tape I/O operations are comprised of the primitive functions described above. Each example shows the primitive functions involved in performing that I/O operation once.

READING	WRITING	ORDERS
ALLOCATE	ALLOCATE	
QUEUE	PROCESS	QUEUE
CHECK	QUEUE	CHECK
PROCESS	CHECK	
DEALLOCATE	DEALLOCATE	

Figure 3: Primitive I/O Functions

Figure 3 shows the primitive functions performed in order to read or write one physical tape record. Tape I/O modules usually process many physical records during an attachment. Thus they perform these primitive functions over and over. The allocate and deallocate functions should not be part of the loop that performs this repeated I/O. Allocated buffers can be used over and over. The read and write loops are the critical paths of any tape I/O module and are the paths that must be optimized for efficiency. Figure 4 shows how the primitive I/O functions are performed within read and write loops.

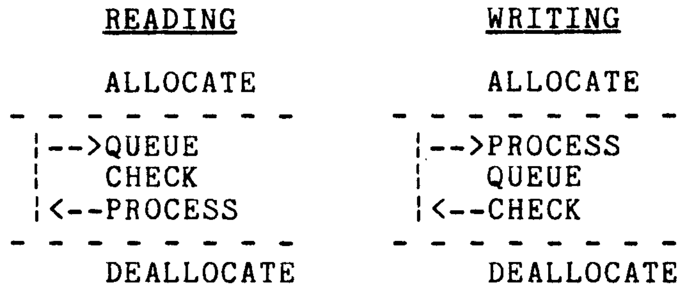

Figure 4: Critical Path Loops

Analysing the primitive functions that comprise reading and writing led to the realization of the following seemingly obvious but subtly important principle:

"Reading and writing are different."

This principle is not so obvious given the hauntingly similar combination of primitive functions that comprise reading and writing. It was also not especially obvious to the designers of the TDCM interface since they decided to have one TDCM entry point perform both operations.

The differences between reading and writing outweigh the similarities. It is possible to play games, and rotate the order in which the primitive functions are performed, so that the order is the same for both operations. There is no way, however, to avoid the reality that, when reading, I/O must be performed before data can be processed. When writing, data must be processed before I/O is performed. Another important difference is the direction of data flow. When reading, data is passed from TAPE_IOI to the caller. When writing, data is passed from the caller to TAPE_IOI.

In accordance with the above principle, TAPE_IOI provides entry points to perform the queue and the check primitive functions for each of the three basic tape I/O operations. These entry points allow a caller to perform the exact sequence of I/O operations wanted. For example, a tape I/O module could use these entry points to implement its own special read-ahead or write-behind algorithms. The fact that there are a set of entry points for each basic tape operation means that it is more efficient and simpler to perform any one operation.

The design criterion that states that a caller be able to do everything that is physically possible with a tape drive is well served by the TAPE_IOI entry points described above. In order to to meet the other design criteria of efficiency and simplicity within the critical path of a tape I/O module, TAPE_IOI also provides entry points that perform multiple primitive functions. There are entry points that allow a caller to efficiently and simply perform read-ahead, write-behind, and order commands.

Buffer States

Except when performing order commands, the primitive functions described above operate on I/O buffers. Each primitive function changes the state of a buffer. An I/O buffer is always in one, and only one, of the following states:

Null: The buffer does not exist.

Ready: The buffer does exist and a caller has, or can get, a pointer to the buffer. The validity of data in the buffer is determined solely by the caller. A caller may perform any kind of processing it wishes on the data in the buffer.

Busy: The buffer is participating in a read or write I/O operation. A caller should not perform any processing of data in the buffer.

Suspended: The buffer had been busy for writing, but due to an error in another buffer, it was not written.

Figure 5 shows how each primitive I/O function changes the state of a buffer. It shows the state of the buffer before and after the primitive function is performed. Primitive functions performed on <u>suspended</u> buffers have the same result as if they were performed on ready buffers.

FUNCTIONS	STATES: BEFORE	STATES: AFTER
ALLOCATE	NULL	READY
DEALLOC	READY	NULL
QUEUE	READY	BUSY
CHECK	BUSY	READY
PROCESS	READY	READY

Figure 5: Buffer State Changes

<u>Block Modes</u>

The <u>check</u> primitive function checks to see if a buffer is ready for processing. If the I/O operation <u>queued</u> for this buffer has not completed, the buffer will still be busy. Until the buffer is in the ready state, processing of data in the buffer cannot begin. Therefore, someone has to wait until the I/O queued for this buffer has completed.

IOI, in ring 0, receives the hardware interrupt that signals the completion of the I/O operation queued for the buffer. IOI tells the outer ring of this event by sending a wakeup.

The block modes supported by TAPE_IOI determine who goes blocked waiting for this wakeup from IOI. There are two choices and therefore two block modes. They are listed below:

<u>Simplex</u>: In simplex block mode TAPE_IOI blocks. TAPE_IOI will automatically wait for any I/O that is needed to complete the check function. This is the default block mode.

<u>Multiplex</u>: In multiplex block mode the caller must block. TAPE_IOI will never block. If waiting for I/O is required in order to complete a check function, TAPE_IOI will inform the caller that it must wait.

Simplex block mode corresponds to the wait/block capabilities currently provided by TDCM. All standard tape I/O modules will operate in simplex mode. No additional complexity will be added to these tape I/O modules since TAPE_IOI will perform all of the blocking.

Multiplex block mode adds tape processing capabilities not provided by TDCM. By allowing the caller to block, and to decide when to block, new tape I/O modules and tape application programs can be developed that multiplex the processing of two or more tape drives. One planned user of multiplex block mode is RCP. Since RCP executes in an inner ring it cannot block. However, RCP can still use TAPE_IOI for label checking and other tape drive processing by operating in multiplex block mode. Multiplex block mode allows RCP to pass on the task of blocking to its caller in the user ring.

Data Lengths and Special Length Processing

The element size supported by TAPE_IOI is one character (9 bits). All data and buffer lengths are expressed in terms of characters. This is especially convenient for tape I/O modules since they interface with their caller, IOX, in terms of characters.

Treating all tape input and output data as character strings results in several problems. These problems are due to the way the tape controller works, and depend upon the tape controller data mode being used. It is beyond the scope of this memorandum to discuss in detail how the tape controller works. The following examples are presented in order to give the reader some idea of the issues involved.

1. When writing a record whose length is an odd number of words, if the tape controller is in binary data mode, then it will append 4 zero bits onto the output record. Normally, these bits should be stripped off by TAPE_IOI when reading this record.

2. When writing a record that is not an integral number of words, the tape controller must be in ASCII data mode. With model 500 tape controllers, the output data must be right aligned.

There are six read/write data modes supported by Honeywell model 500 9-track tape drives. TAPE_IOI provides a mode entry point to set the read/write data mode to any one of the six. Each of the read/write data modes supported by the tape drives interface with the tape controller in either binary or ASCII. Figure 6 lists the six tape drive read/write data modes and their associated tape controller data modes.

TAPE DRIVE	CONTROLLER	
BINARY	BINARY	
BCD	BINARY	
ASCII	BINARY	*
EBCDIC	BINARY	*
TAPE 9	ASCII	*
ASCII/EBCDIC	ASCII	*

* => 9 track only

Figure 6: Tape Drive and Controller Modes

For most callers, the standard way that TAPE_IOI deals with record lengths is correct and sufficient. However, some callers, for example a raw tape I/O module, need special length processing. This special length processing involves reading and writing records that do not end on a word boundary. The processing of such records is rather complex and is only done when a caller tells TAPE_IOI, via the set_mode entry point, that special length processing is needed.

The rules enforced by TAPE_IOI for data and buffer lengths are listed below. Some of these rules apply to all cases, others depend upon the length processing mode, the tape I/O operation, and the tape controller mode. The binary and ASCII modes referred to in the following rules are tape controller modes as described in Figure 6.

GENERAL

1. I/O buffers must be 0 mod 8 characters in length (2 words). This is required since, when in binary mode, the tape controller always transfers data in units of 2 words.

2. The minimum length of an output record is 4 characters. The tape controller is incapable of writing a record smaller than one word. Users of TAPE_IOI are cautioned that it is unwise to write records that are less than 64 characters in length since there is little chance of successfully reading back a record that is any smaller. This is because the tape controller treats short input records as noise.

NORMAL LENGTH MODE

1. Reading - Binary: Input record lengths are always 0 mod 4 characters in length (1 word). Any data contained in a partial last word will not be included in the record length.

2. Reading - ASCII: Input record lengths will reflect the actual number of characters read.

3. Writing: In both binary and ASCII modes, output records must be 0 mod 4 characters in length (1 word). Any attempt to write a record that is not an integral number of words will be rejected by TAPE_IOI.

SPECIAL LENGTH MODE

1. Reading - Binary: Input record lengths are returned the same way as in regular length mode. In addition, the bit count of the record, including any data read into a partial last word, will be saved. This bit count can be obtained from status kept by TAPE_IOI for each buffer. Reading with the controller in binary mode and TAPE_IOI in special length mode is the only way a caller can read the exact data in all tape records.

2. Reading - ASCII: Input record lengths are returned the same way as in regular length mode.

3. Writing - Binary: Record lengths that are not 0 mod 4 characters (1 word) will be allowed. If necessary, the length of the record actually processed by TAPE_IOI will be increased to make it 0 mod 4 characters. However, no right hand padding of the output record will be performed.

4. Writing - ASCII: Record lengths that are not 0 mod 4 characters (1 word) are allowed. TAPE_IOI will set the initial and terminate character position fields according to the specified record length. The data for this record must be right or left aligned by the caller depending upon the setting of the TAPE_IOI align mode. Writing with the controller in ASCII mode and TAPE_IOI in special length mode is the only way a caller can write a record that is not an integral number of words.

Error Recovery

Most tape I/O modules perform similar error recovery procedures. The Multics standard tape I/O module is a notable exception in that it will perform its own special error recovery procedures. If told to, TAPE_IOI will perform what are considered to be the standard tape error recovery procedures. By providing this common service, the complexity of tape I/O modules can be reduced and more development effort can be devoted to doing a better job of error recovery within TAPE_IOI.

In order to to help callers that must do their own read error recovery, TAPE_IOI allows them to specify the channel instruction field to be used in tape read operations. This allows the caller to specify the kind, if any, of hardware error retry to be performed.

The standard error recovery procedures performed by TAPE_IOI are listed below. These procedures will be initiated if, in the opinion of TAPE_IOI, an I/O operation failed in a way that is recoverable.

Reading: Every read operation will be initiated with automatic hardware error retry enabled and with normal deskew window and threshold. In the event of an error, TAPE_IOI will backspace over the record in error and retry the read operation. Each retry will be performed with automatic hardware error retry enabled and a different deskew window and threshold. TAPE_IOI will retry the read operation until all combinations (8) of deskew windows and thresholds have be tried.

Writing: When a write operation fails, TAPE_IOI will backspace over the record in error, erase, and try to rewrite the record. This sequence will be attempted up to 30 times. Performing this backspace erase sequence more than 30 times will result in a blank spot on the tape that cannot be read past.

Order Commands: Depending upon the error and the order command being executed, TAPE_IOI will retry the order command a limited number of times.

Status Information

TAPE_IOI returns status information in increasing levels of detail. Much of the complexity of all tape I/O modules involves the interpretation of status. An important feature of TAPE_IOI is its interpretation of status for its caller and its ability to return this interpreted status in a useful form. TAPE_IOI, however, does not limit its callers to using this interpreted status. Another important feature of TAPE_IOI is that it provides a caller with all available raw status. Interpreted and raw status are the highest and lowest levels, respectively, in the TAPE_IOI status hierarchy. Figure 7 shows all four levels of the TAPE_IOI status hierarchy.

```
interpreted
  descriptive
    reformatted
      raw
```

Figure 7: TAPE_IOI Status Hierarchy

INTERPRETED STATUS

TAPE_IOI returns interpreted status in the form of a result index. Most callers of TAPE_IOI can perform all of their tape processing with just the status interpretation provided by result indexes. Result indexes are fixed binary variables that are intended to be used in referencing PL/I label arrays. A result index is returned by all TAPE_IOI entry points that perform a check primitive function. The result index value is generated by TAPE_IOI by interpreting the hardware status associated with the tape I/O operation that was checked. The result index values returned by TAPE_IOI are listed below. See Appendix D for a complete list of the result index values returned for each tape I/O operation.

(-1) Block: This result index value indicates that the I/O operation being checked has not yet completed. This value will be returned only when TAPE_IOI is operating in the multiplex block mode. A wakeup will be sent to the caller when the I/O operation has completed. The caller should go blocked waiting for this wakeup. When waked up, the caller should call TAPE_IOI to again check the I/O operation. Before going blocked the caller may perform other processing.

(0) Success: The I/O operation being checked has completed successfully. If the I/O operation was a read or a write, the caller may now begin processing data in the buffer associated with this I/O operation.

(1) Program Error: The caller has violated some requirement of TAPE_IOI. The requested action was not performed. The status code returned contains an error_table_ value that indicates the particular error.

(2) Unrecoverable I/O Error: The I/O operation being checked has failed in such a way that (almost certainly) precludes it from ever being performed successfully. It is also unlikely that subsequent I/O operations will succeed. Such an error probably indicates an event requiring manual intervention by the operator or a hardware malfunction. The status code returned contains an error_table_ value that indicates the particular error.

(3) I/O Error: The I/O operation being checked has failed. However, in the opinion of TAPE_IOI, retrying the operation may succeed. For I/O operations that read or write, this implies that the I/O operation succeeded in at least moving the tape. Such an error probably resulted either from a defective section of tape, or a spurious hardware error condition. This result index value will be returned only when TAPE_IOI is not performing error recovery. If TAPE_IOI is performing error recovery, the occurrence of such an error will cause TAPE_IOI to retry the I/O operation. The check function will not be completed until either the I/O operation is successful or all error recovery procedures have been tried and have failed. If, after trying all error recovery procedures, TAPE_IOI cannot successfully complete the I/O operation, it will return a result index value (2) that indicates an unrecoverable I/O error. The status code returned contains an error_table_ value that indicates the particular error.

(4, 5, 6) Special Event: The I/O operation being checked has completed and some special event has occurred that is normal for this I/O operation. These result index values are operation dependent. Examples of special events are: reading an EOF record, writing past the end of tape reflector, backspacing when at BOT, etc. The status code returned contains an error_table_ value that indicates the particular event.

DESCRIPTIVE STATUS

TAPE_IOI provides an entry point that will return a character string containing an English language description of the hardware status resulting from the last I/O operation checked. See the description of the hardware_status entry point in Appendix A. Tape I/O modules can provide an order call that returns this status string. The caller of the tape I/O module can then display this status string to the user.

REFORMATTED STATUS

Some callers of TAPE_IOI may need to perform their own interpretation of the hardware status resulting from an I/O operation. The hardware_status entry point also returns hardware status in a reformatted form. The status information returned is the major and substatus for the last I/O operation that was checked.

The purpose of returning reformatted major and substatus is to make it easier for the caller to interpret this status. It also means that the calling programs will be interpreting logical status (as generated by TAPE_IOI). This will make them more independent of future changes in the real hardware status.

RAW STATUS

The TAPE_IOI hardware_status entry point also returns all raw hardware status available from the last I/O operation checked. This status information includes all 72 bits of IOM status.

Interface to RCP

TAPE_IOI allows its callers to interface directly with RCP. This is possible because TAPE_IOI itself does not *have* to interface with RCP. TAPE_IOI does, however, provide entry points that perform the common interfaces between tape I/O modules and RCP.

These entry points will attach and detach a tape drive. They allow the caller to specify all input information needed by RCP. They also return all output information returned by RCP.

SUMMARY

This section summarizes and lists the major capabilities and entry points provided by TAPE_IOI.

List of Capabilities

Below is a list summarizing the major capabilities of TAPE_IOI. The reader is urged to read this list and then compare it with the list of TDCM problems presented in the overview section.

1. Complete management of the IOI workspace.

2. The ability to allocate the exact number and size of I/O buffers needed, provided there is room in the IOI workspace. Also the ability to read and write records that are longer than 16,384 characters.

3. Allows a caller to directly process input and output data in the IOI workspace.

4. Provides a character orientated interface that is especially useful for IOX tape I/O modules.

5. Provides an entry point for each combination of primitive function and basic I/O operation.

6. Provides entry points that perform read-ahead and write-behind. These entry points perform the combinations of primitive functions that are found within the critical paths of read and write loops.

7. Will modify channel programs in execution in order to queue a buffer for reading or writing. This feature will allow a user, who is receiving sufficient processing time from the system, to perform continuous I/O.

8. Allows a caller to issue order commands that read or write data.

9. Can read all data contained in records that are not an integral number of words in length.

10. Can write records that are not an integral number of words in length.

11. Performs all blocking, or optionally allows the caller to perform all blocking.

12. Performs standard error recovery procedures.

13. Allows a caller to specify the channel instruction field to be used in any read operation.

14. Interprets hardware status for the caller and returns it in a useful form.

15. Provides a caller with all available hardware status.

16. Lets the caller interface directly with RCP, or optionally will call RCP for the caller.

17. Support for 9-track and 7-track tape drives. Support for 400, 500, and 600 model tape drives. Provides an interface that is independent of the model tape drive being used.

List of Entry Points

Below is a list of the TAPE_IOI entry points. In this list the entry points are ordered according to function. Each entry point is accompanied by a brief description of what the entry point does. See Appendix A for a complete description of these entry points including their calling sequences.

INITIALIZATION ENTRY POINTS

attach: Calls RCP to attach a tape drive.

activate: Initiates a TAPE_IOI activation.

deactivate: Terminates the TAPE_IOI activation.

detach: Calls RCP to detach the tape drive.

WORKSPACE ALLOCATION ENTRY POINTS

allocate_buffers: Allocates the specified number of I/O buffers, all of which must be the same length.

allocate_work_area: Allocates a caller work area in the IOI workspace.

deallocate: Deallocates <u>all</u> I/O buffers and any caller work area.

SPECIAL ENTRY POINTS

set_mode: Sets one of the TAPE_IOI modes.

get_mode: Gets one of the TAPE_IOI modes.

set_buffer_ready: Puts a specified buffer in the ready buffer state.

stop_tape: Stops all I/O operations currently in progress. All buffers are put into the ready buffer state.

STATUS ENTRY POINTS

list_buffers: Returns a list of all allocated buffers, or all buffers that are currently in a specified state.

buffer_status: Returns all information relevant to the specified buffer.

hardware_status: Returns all available hardware status obtained from the last I/O operation that was checked.

READ ENTRY POINTS

queue_read: Queues a read operation for the specified I/O buffer.

check_read: Performs a check of the read operation that has been queued the longest.

read: Queues read operations for all ready I/O buffers. It then performs a check of the read operation that has been queued the longest.

WRITE ENTRY POINTS

queue_write: Queues a write operation for the specified I/O buffer.

check_write: Performs a check of the write operation that has been queued the longest.

write: Queues a write operation for the specified I/O buffer, or optionally for all I/O buffers currently in the suspended buffer state. It also returns a pointer to a ready buffer. If there are no ready buffers, then it will perform a check of the write operation that has been queued the longest.

ORDER COMMAND ENTRY POINTS

queue_order: Queues the specified tape order command. No other I/O may be in progress.

check_order: Performs a check of the current order I/O operation.

order: Queues the specified tape order command. No other I/O may be in progress. It also performs a check of this order operation.

Entry: tape_ioi_$activate

This entry point initiates a TAPE_IOI activation for a tape drive. In order for the activation to succeed, the specified tape drive must be attached to the calling process. Only one TAPE_IOI activation is allowed at any one time for the same tape drive.

No TAPE_IOI entry points, except tape_ioi_$attach and tape_ioi_$detach, may be called unless TAPE_IOI is activated. This entry point returns an ID that must be used in calls to all other TAPE_IOI entry points, except tape_ioi_$attach and tape_ioi_$detach.

Usage

```
dcl  tape_ioi_$activate entry (ptr, bit(36) aligned, fixed
         bin(35));

call tape_ioi_$activate (tioi_info_ptr, tioi_id, code);
```

where:

1. tai_ptr — is a pointer to a structure containing information needed by TAPE_IOI for this activation. A description of this structure is given below. (Input)

2. tioi_id — is an identifier that uniquely identifies this TAPE_IOI activation. This ID must be used in subsequent calls to TAPE_IOI. (Output)

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

The tai_ptr pointer must point to a structure with the format shown below. All fields in this structure are considered to be input by tape_ioi_$activate. A declaration for this structure can be found in the include file: tioi_activate_info.incl.pl1.

```
dcl 1 tai  based(tai_ptr)  aligned,
        2 version         fixed bin,        /*  1. */
        2 actv_name       char(32),         /*  2. */
        2 drive_name      char(8),          /*  3. */
        2 volume_name     char(32),         /*  4. */
        2 write_flag      bit(1),           /*  5. */
        2 model           fixed bin,        /*  6. */
        2 tracks          fixed bin,        /*  7. */
        2 density         bit(36),          /*  8. */
        2 ioi_index       fixed bin,        /*  9. */
        2 workspace_max   fixed bin(19),    /* 10. */
        2 timeout_max     fixed bin(71),    /* 11. */
        2 event_id        fixed bin(71);    /* 12. */
```

where:

1. version — is the version number of this structure. The current value is defined in the include file.

2. actv_name — is a caller defined character string identifier for this activation. All standard tape I/O modules will set this field to be the IOX switch name for the this tape drive attachment.

3. drive_name — is the name of the attached tape drive for which this activation is being performed.

4. volume_name — is the name of the tape reel mounted on this attached tape drive.

5. write_flag — is a flag specifying whether or not write operations should be allowed. If this flag is OFF ("0"b) then only read type operations will be allowed. If it is ON ("1"b) then all I/O operations will be allowed.

6. model — is the model number of the tape drive (400, 500, or 600).

7. tracks — is the number of tracks (7 or 9).

8. density — specifies the density capabilities of the attached tape drive.

9. ioi_index — is the IOI index for this attachment.

10. workspace_max is the maximum size (in words) of the IOI workspace used for this tape drive attachment. This value must be less than or equal to the value returned by RCP.

11. timeout_max is the maximum time limit (in micro-seconds) allowed for any individual I/O operation. This value must be less than or equal to the value returned by RCP.

12. event_id This is the IPC event channel ID to be used for blocking when waiting due to a check operation.

Entry: tape_ioi_$allocate_buffers

This entry point allocates I/O buffers in the IOI workspace. Any number of buffers may be allocated, limited only by the buffers that are already allocated, the size of the buffers being allocated, and the size of the IOI workspace. All buffers allocated in any one call will be the same size. No I/O buffers may be allocated after a work area has been allocated. I/O buffers may subsequently be redefined in the IOI workspace by calling tape_ioi_$deallocate.

Usage

```
dcl  tape_ioi_$allocate_buffers entry (bit(36) aligned,
     fixed bin(21), fixed bin, fixed bin(21), fixed bin,
     dim(*) ptr, fixed bin(35));

call tape_io_$allocate_buffers (tioi_id, req_length,
     req_number, act_length, act_number, buffer_ptrs, code);
```

where:

1. tioi_id uniquely identifies this TAPE_IOI activation. (Input)

2. req_length is the requested length (in characters) of the buffers to be allocated. If this length is zero, then TAPE_IOI will allocate the requested number of buffers, each as long as possible. (Input)

3. req_number — is the requested number of I/O buffers to be allocated. If this number is zero, then TAPE_IOI will allocate as many buffers as possible, each with the requested length. A program error will occur if both the requested length and the requested number of buffers is zero. (Input)

4. act_length — is the actual length of the buffers allocated by this call. This length will always be 0 module 8 characters (2 words). (Output)

5. act_number — is the actual number of I/O buffers that were allocated by this call. (Output)

6. buffer_ptrs — is an array of pointers to the I/O buffers that were allocated. If hbound(buffer_ptrs,1) is less than the number of buffers allocated, then only that number of buffer pointers are returned. If hbound(buffer_ptrs,1) is zero, no buffer pointers are returned. (Output)

7. code — is a standard Multics system status code. (Output)

Entry: tape_ioi_$allocate_work_area

This entry point allocates a work area in the IOI workspace. The caller can use this work area to keep information needed to process this tape. The allocation will not succeed if a work area is already allocated. Work areas are always allocated at the end of the last IOI workspace page containing allocated I/O buffers. No additional IOI workspace pages will be used for this work area.

Usage

```
dcl  tape_ioi_$allocate_work_area entry  (bit(36) aligned,
     fixed bin(19), fixed bin(19), ptr, code);

call tape_ioi_$allocate_work_area (tioi_id, req_size,
     act_size, work_area_ptr, code);
```

where:

1. tioi_id uniquely identifies this TAPE_IOI activation. (Input)

2. req_size is the size (in words) of the work area to be allocated. The allocation will fail, and a program error will occur, if there is not enough space left in the last page of the IOI workspace for a work area of the specified size. If this size is zero, then all of the remaining space in the last page of the IOI workspace will be allocated as a work area. (Input)

4. act_size is the actual size of the work area allocated. (Output)

3. work_area_ptr is a pointer to the work area allocated in the IOI workspace. (Output)

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

<u>Entry</u>: tape_ioi_$attach

This entry point calls RCP to attach a tape drive. This entry point is provided for callers that do not want to interface directly with RCP. It should be called before activating TAPE_IOI for this tape drive.

This entry point allows the caller to specify all information needed by RCP in order to select and attach a tape drive. It also returns all information return by RCP for this tape drive attachment.

If this entry point returns without error, the caller can assume that the specified tape reel has been mounted and attached on the specified tape drive, is positioned at BOT, and is ready for processing.

Usage

```
dcl  tape_ioi_$attach entry (ptr, ptr, char(*), bit(36)
     aligned, fixed bin(35));

call tape_ioi_$attach (tape_info_ptr, tai_ptr, comment,
     rcp_id, code);
```

where:

1. tape_info_ptr is a pointer to a structure containing information needed by RCP in order to select and attach a tape drive. A description of this structure is given below. (Input)

2. tai_ptr is a pointer to a structure containing information needed by TAPE_IOI. This information is returned by TAPE_IOI. All fields in this structure, except the version and actv_name fields, will be set by this entry point. This allows the caller to pass this structure directly to the tape_ioi_$activate entry point. See the description of the tape_ioi_$activate entry point for a description of this structure. (Input)

3. comment is a string that will be displayed to the system operator after RCP has successfully completed the attachment. No comment will be displayed if this string is null or blank. This comment will be tested for illegal characters. RCP will consider a character to be illegal if it does not belong to the 95 character subset of ASCII characters (octal 040 - 176) that are usually considered printable. Any illegal characters found will be converted to blanks. This comment will be displayed in the form of an RCP note message. The format of this message is given below. (Input)

   "RCP: Note (drive name) - comment"

4. rcp_id is RCP's unique identifier for this attachment. This ID must be used in all subsequent calls to RCP or tape_ioi_$detach for this tape drive. (Output)

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

The tape_info_ptr must point to a structure with the format shown below. This structure is used to interface with RCP. All of the fields in this structure, except the version field, are input/output fields. The caller should set all of these fields so that RCP can select the desired tape drive. All of this input information will be sent to RCP. All of the output information will be returned by RCP. Fields that are duplicated in the tioi_activate_info structure will be set in both structures with the same values. A declaration for this structure can be found in the include file: rcp_tape_info.incl.pl1.

```
dcl 1 tape_info  based(tape_info_ptr)  aligned,
      2 version        fixed bin,       /*  1. */
      2 usage_time     fixed bin,       /*  2. */
      2 wait_time      fixed bin,       /*  3. */
      2 system_flag    bit(1),          /*  4. */
      2 device_name    char(8),         /*  5. */
      2 model          fixed bin,       /*  6. */
      2 tracks         fixed bin,       /*  7. */
      2 density        bit(36),         /*  8. */
      2 volume_name    char(32),        /*  9. */
      2 write_flag     bit(1),          /* 10. */
      2 position_index fixed bin;       /* 11. */
```

where:

1. version is the version number of this structure.

2. usage_time currently must be zero.

3. wait_time currently must be zero.

4. system_flag — is used to tell RCP whether or not RCP should consider the calling process to be a system process for this attachment. A value of "1"b implies yes, "0"b implies no. In addition to asking to be treated as a system process, the calling process must have "E" access to the gate rcp_sys_. On output, this field will be set to "1"b if RCP is actually treating this process as a system process for this attachment.

5. device_name — specifies whether or not a specific tape drive is to be attached. If this field is not blank, RCP will assume that it specifies the name of the tape drive to be attached. In this case, RCP will attempt to attach only this tape drive. RCP will ignore any device characteristics found in fields in this structure. However, if this field is blank, RCP will attempt to attach a tape drive based upon these other device characteristics. On output, this field will contain the device name of the attached tape drive.

6. model — specifies the model number of the tape drive that is to be attached. If the value of this field is 0, RCP will not consider the model characteristic in its selection of a tape drive to attach. Otherwise, RCP will select only a tape drive that has the specified model number. The acceptable values are: 400, 500, and 600. On output, this field will contain the model number of the attached tape drive.

7. tracks — specifies the track type of the tape drive to be attached. If the value of this field is 0, RCP will not consider the track type characteristic in its selection of a tape drive to attach. Otherwise, RCP will select only a tape drive that has the specified track type. The acceptable values are: 7 and 9. On output, this field will contain the track type of the attached tape drive.

8. density — specifies the density capabilities of the tape drive to be attached. On output, this field will contain the density capabilities of the attached tape drive. If this field contains all zeros, RCP will not consider density capabilities in its selection of a tape drive to attach. Otherwise, RCP will select only a tape drive that has the specified density capabilities. This field does not deal with the actual density setting of the tape drive. It deals with the <u>possible</u> density setting that the tape drive is capable of. In this field, one bit is used for each of the four currently supported density settings. All unused bits in this field must be set to zero. Counting from left to right, and numbering from 1 to 4, the bits in this field correspond to the following density settings:

1 - 200 BPI
2 - 556 BPI
3 - 800 BPI
4 - 1600 BPI

8. volume_name — specifies the volume name of the tape reel to be used during this attachment. Unless a specific tape drive is requested, RCP will attempt to select the tape drive on which this tape reel is already mounted. Currently, this field is not changed on output.

10. write_flag — is a flag that specifies whether or not the tape reel to be used for this attachment should be mounted with a write ring. If this flag is OFF ("0"b) then the tape reel will be mounted without a write ring. If it is ON ("1"b) then the tape reel will be mounted with a write ring. Currently, this field is not changed on output.

11. position_index — currently must be zero.

Entry: tape_ioi_$buffer_status

This entry point returns the status of the specified I/O buffer.

Usage

```
dcl  tape_ioi_$buffer_status entry (bit(36) aligned, ptr,
     ptr, fixed bin(35));

call tape_ioi_$buffer_status (tioi_id, buffer_ptr, tbs_ptr,
     code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — is a pointer to the buffer whose status is requested. (Input)

3. tbs_ptr — is a pointer to a TAPE_IOI buffer status structure. A description of this structure is given below. (Input)

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

The tbs_ptr must point to a structure with the format shown below. All fields in this structure, except the version number field, are output fields whose values will be set by TAPE_IOI. A declaration for this structure can be found in the include file: tioi_buffer_status.incl.pl1.

```
dcl 1 tbs based(tbs_ptr) aligned,
      2 version          fixed bin,          /*  1. */
      2 state            fixed bin,          /*  2. */
      2 buffer_len       fixed bin(21),      /*  3. */
      2 data_len         fixed bin(21),      /*  4. */
      2 bit_len          fixed bin(24),      /*  5. */
      2 modes            aligned,
        3 cif            bit(6),             /*  6. */
        3 data           char(4),            /*  7. */
       (3 align          bit(1),             /*  8. */
        3 length         bit(1),             /*  9. */
        3 recovery       bit(1),             /* 10. */
        3 pad            bit(33)) unaligned;
```

where:

1. version — is the version number of this structure.

2. state — is the current state of the buffer, as follows:
   - 1 => ready
   - 2 => busy
   - 3 => suspended

3. buffer_len — is the allocated length of this buffer, in characters.

4. data_len — is the length (in characters) of the actual data in this buffer.

5. bit_len — is the length (in bits) of the actual data in this buffer. This field is valid only when the buffer has been used for a read operation while TAPE_IOI was in special length mode.

6. modes.cif — is the setting of the channel instruction field use to process this buffer.

7. modes.data — is the setting of the data mode for this buffer at the time the buffer was queued. See Appendix C for a list of the TAPE_IOI data modes.

8. modes.align — is the setting of the align mode at the time this buffer was queued. This field may contain the following values:
   - "0"b => Left Aligned
   - "1"b => Right Aligned

9. modes.length — is the setting of the length mode at the time this buffer was queued. This field may contain the following values:
   - "0"b => Normal Length Mode
   - "1"b => Special Length Mode

10. modes.recovery — is the setting of the error recovery mode at the time this buffer was queued. This field may contain the following values:
    - "0"b => No Recovery
    - "1"b => Error Recovery

Entry: tape_ioi_$check_order

This entry point performs a check of the order I/O operation currently queued. A program error (error_table_$device_not_active) will occur if there is no order operation queued.

For order operations that involve special interrupts (rewind, rewind and unload, etc.) the check operation will be completed when the order operation terminates. In order to check the special interrupt itself, the caller must use the "ready" order command. See Appendix B for more information about this order command.

Usage

```
dcl  tape_ioi_$check_order entry (bit(36) aligned, fixed
          bin, fixed bin, fixed bin(35));

call tape_ioi_$check_order (tioi_id, ocount, rx, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. ocount — is the number of times the order command was actually performed. (Output)

3. rx — is the result index generated by interpreting the status obtained from the order operation being checked. (Output)

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

Entry: tape_ioi_$check_read

This entry point performs a check of the read operation that has been queued the longest. The successful checking of a read operation means that the buffer used in this read operation will be placed in the ready state. The caller may begin processing the data read into this buffer. A program error (error_table_$device_not_active) will occur if there is no read operation queued.

Usage

```
dcl tape_ioi_$check_read entry (bit(36) aligned, ptr, fixed
    bin(21), fixed bin, fixed bin(35));

call tape_ioi_$check_read (tioi_id, buffer_ptr, data_len,
    rx, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)
2. buffer_ptr — is a pointer to the buffer used in the read operation being checked. (Output)
3. data_len — is the length (in characters) of the data actually read into the buffer. (Output)
4. rx — is the result index generated by interpreting the status obtained from the read operation being checked. (Output)
5. code — is a standard Multics system status code. (Output)

Entry: tape_ioi_$check_write

This entry point performs a check of the write operation that has been queued the longest. The successful checking of a write operation means that the buffer used in this write operation will be placed in the ready state. The caller may begin copying output data into this buffer. A program error (error_table_$device_not_active) will occur if there is no write operation queued.

If other write operations are queued, and any kind of I/O error or special event occurs (result index value is > 1), then these other write operations will be suspended and the buffers used for these write operations will be placed in the suspended state.

Usage

```
dcl  tape_ioi_$check_write entry (bit(36) aligned, ptr,
          fixed bin, fixed bin(35));

call tape_ioi_$check_write (tioi_id, buffer_ptr, rx, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — is a pointer to the buffer used for the write operation being checked. (Output)

3. rx — is the result index generated by interpreting the status obtained from the write operation being checked. (Output)

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

Entry: tape_ioi_$deactivate

This entry point will terminate the current TAPE_IOI activation for a tape drive. As a result of deactivation, all I/O buffers and any work area will be deallocated.

If any I/O operations are currently queued for this tape drive, then a program error will occur. Therefore, if it is not known whether or not I/O operations are currently queued, a call should be made to tape_ioi_$stop_tape before calling this entry point.

Usage

```
dcl  tape_ioi_$deactivate entry (bit(36) aligned, fixed
          bin(35), fixed bin(35));

call tape_ioi_$deactivate (tioi_id, error_count, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. After deactivation, this TAPE_IOI ID is no longer valid. Any subsequent calls to TAPE_IOI using this TAPE_IOI ID will result in a program error. (Input)

2. error_count — is a count of I/O errors detected by TAPE_IOI during this activation. (Output)

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

Entry: tape_ioi_$deallocate

This entry point will deallocate all I/O buffers and any work area. A program error occurs unless all I/O buffers are in the ready or suspended state. The caller may allocate new I/O buffers and a new work area by calling the TAPE_IOI allocate entry points.

Usage

```
dcl  tape_ioi_$deallocate entry (bit(36) aligned, fixed
          bin(35));

call tape_ioi_$deallocate (tioi_id, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

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

Entry: tape_ioi_$detach

This entry point calls RCP to detach a tape drive. This entry point is provided for callers that do not want to interface directly with RCP.

Detaching the tape drive may involve demounting the tape reel mounted on this tape drive. The detachment may also involve unassigning a tape drive. These action are performed by RCP and depend upon the "disposition" argument and whether are not the tape drive and tape reel were assigned and mounted by explicit command.

Usage

```
dcl  tape_ioi_$detach entry (bit(36) aligned, bit(*), fixed
         bin(35), char(*), fixed bin(35));

call tape_ioi_$detach (rcp_id, disposition, error_count,
         comment, code);
```

where:

1. rcp_id — is the RCP ID that was returned by RCP when the tape drive was attached. (Input)

2. disposition — specifies the action to be performed by RCP with regard to the assignment disposition of the tape drive being detached. The disposition of the tape drive involves the possible retention of the tape drive assignment even though the tape drive is being detached. The acceptable values which this argument currently may have are: (Input)
   "0"b => unspecified
   "1"b => retain the assignment

3. error_count — specifies the number of error detected by the caller during the attachment. RCP will keep a cumulative total of all errors reported. (Input)

4. comment — is a string that will be displayed to the system operator after the tape drive has been detached. See the description of the tape_ioi_$attach entry point for details about valid comment strings. (Input)

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

Entry: tape_ioi_$get_mode

This entry point will return the current value of any one of the TAPE_IOI modes. See Appendix C for more information about the TAPE_IOI modes.

Usage

```
dcl  tape_ioi_$get_mode entry (bit(36) aligned, char(8),
     ptr, fixed bin(35));

call tape_ioi_$get_mode (tioi_id, mode, data_ptr, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. mode — specifies the name of the mode to be returned. (Input)

3. data_ptr — is a pointer to a location where the current value of the mode is to be stored. (Input)

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

Entry: tape_ioi_$hardware_status

This entry point returns all available status information obtained from the last I/O operation for which a check has been completed.

Usage

```
dcl tape_ioi_$hardware_status entry (bit(36) aligned, ptr,
    fixed bin(35));

call tape_ioi_$hardware_status (tioi_id, ths_ptr, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. ths_ptr — is a pointer to a TAPE_IOI hardware status structure. A description of this structure is given below. (Output)

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

The ths_ptr must point to a structure with the format shown below. All fields in this structure, except the version field, are output fields whose values will be set by TAPE_IOI. A declaration for this structure can be found in the include file: tioi_hardware_status.incl.pl1.

```
dcl 1 ths based(ths_ptr) aligned,
      2 version     fixed bin,               /* 1. */
      2 description char(128) varying,       /* 2. */
      2 major       fixed bin,               /* 3. */
      2 substatus   bit(36),                 /* 4. */
      2 iom         bit(72);                 /* 5. */
```

where:

1. version — is the version number of this structure.

2. description — is an English language description of this hardware status.

3. major is reformatted major status. A description is given below.

4. substatus is reformatted substatus. A description is given below.

5. iom is the raw IOM hardware status.

The include file tioi_hardware_status.incl.pl1 also contains constants that can be used to reference the fields of reformatted major and substatus. The values that represent each major status, and the names of the constants that should be used to reference each major status value are given below:

```
dcl  subsystem_ready        fixed bin  init (0)  static;
dcl  device_busy            fixed bin  init (1)  static;
dcl  device_attention       fixed bin  init (2)  static;
dcl  device_data_alert      fixed bin  init (3)  static;
dcl  end_of_file            fixed bin  init (4)  static;
dcl  command_reject         fixed bin  init (5)  static;
dcl  mpc_device_attention   fixed bin  init(10)  static;
dcl  mpc_device_data_alert  fixed bin  init(11)  static;
dcl  mpc_command_reject     fixed bin  init(13)  static;
dcl  power_off              fixed bin  init(16)  static;
dcl  system_fault           fixed bin  init(17)  static;
dcl  iom_central            fixed bin  init(18)  static;
dcl  iom_channel            fixed bin  init(19)  static;
dcl  time_out               fixed bin  init(20)  static;
```

Each substatus, for a given major status, is represented by one bit in the ths.substatus field. Whenever more than one substatus occurs at the same time, the bits representing each will be set. For each major status, there is a set of constants representing the values for all substatuses possible for that major status. As an example, listed below are the values and the names of the constants used to represent the substatuses associated with the subsystem_ready major status.

```
dcl  device_ready     bit(36)  init ("10000000"b)  static;
dcl  write_protected  bit(36)  init ("01000000"b)  static;
dcl  at_bot           bit(36)  init ("00100000"b)  static;
dcl  nine_track       bit(36)  init ("00010000"b)  static;
dcl  two_bit_fill     bit(36)  init ("00001000"b)  static;
dcl  four_bit_fill    bit(36)  init ("00000100"b)  static;
dcl  six_bit_fill     bit(36)  init ("00000010"b)  static;
dcl  ascii_alert      bit(36)  init ("00000001"b)  static;
```

Entry: tape_ioi_$list_buffers

This entry point will return a list of pointers to buffers in the IOI workspace. All buffers, or all buffers in a particular state can be listed. When all buffers are listed, they will be listed in the order in which they were allocated. When all buffers in a particular state are listed, they will be listed in the order in which they were put into that state.

Usage

```
dcl  tape_ioi_$list_buffers entry (bit(36) aligned, fixed
         bin, dim(*) ptr, fixed bin, fixed bin(35));

call tape_ioi_$list_buffers entry (tioi_id, state,
         buffer_ptrs, num_buffers, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. state — specifies the state of the buffers to be returned. The acceptable values are listed below: (Input)
   0 => all buffers
   1 => all ready buffers
   2 => all busy buffers
   3 => all suspended buffers

3. buffer_ptrs — is an array of pointers to the buffers. Only the number of buffer pointers that will fit into this array, hbound(buffer_ptrs,1), will be returned. (Output)

4. num_buffers — is the number of buffers in the state requested. (Output)

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

Entry: tape_ioi_$order

This entry point is called to queue and check an order. Calling this entry is equivalent to calling, in succession, tape_ioi_$queue_order and tape_ioi_$check_order.

Usage

```
dcl  tape_ioi_$order entry (bit(36) aligned, char(4), fixed
         bin, ptr, fixed bin, fixed bin, fixed bin(35));

call tape_ioi_$order (tioi_id, order, count, data_ptr,
         ocount, rx, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)
2. order — see tape_ioi_$queue_order. (Input)
3. count — see tape_ioi_$queue_order. (Input)
4. data_ptr — see tape_ioi_$queue_order. (Input)
5. ocount — see tape_ioi_$check_order. (Output)
6. rx — see tape_ioi_$check_order. (Output)
7. code — is a standard Multics system status code. (Output)

Entry: tape_ioi_$queue_order

This entry point will queue one tape order command. A program error will occur if an order command is already queued or if any read or write operations are queued. All non-channel orders are supported. See Appendix B for a list of the mnemonics, counts, and data associated with these order commands.

Usage

```
dcl  tape_ioi_$queue_order entry (bit(36) aligned, char(4),
     fixed bin, ptr, fixed bin(35));

call tape_ioi_$queue_order (tioi_id, order, count, data_ptr,
     code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. order — is the mnemonic name of the order to be queued. The reason for using these mnemonics is to make the use of this entry point simple, and to present a logical rather than a physical interface wherever possible. (Input)

3. count — is the number of times the order is to be executed. For some orders this field is ignored and the order command is executed only once. (Input)

4. data_ptr — is a pointer to any data required/returned by the order command. For orders which do not involve data, this argument is ignored. Warning: For order commands that return data, TAPE_IOI will remember this pointer and will use it to return data when the order operation is checked. It is the callers responsibility to make sure that this pointer is valid when the order operation is checked. (Input)

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

Entry: tape_ioi_$queue_read

This entry point will queue a read operation for the specified I/O buffer. A program error will occur if this I/O buffer is already queued for reading, any order command or write operation is queued, or any buffer is in the suspended state.

Usage

```
dcl tape_ioi_$queue_read entry (bit(36) aligned, ptr, fixed
     bin(35));

call tape_ioi_$queue_read (tioi_id, buffer_ptr, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — is a pointer to the buffer for which the read operation is to be queued. (Input)

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

Entry: tape_ioi_$queue_write

This entry point will queue a write operation for the specified I/O buffer. A program error will occur is this I/O buffer is already queued for writing or if any order command or read operations are queued.

Usage

```
dcl tape_ioi_$queue_write entry (bit(36) aligned, ptr,
     fixed bin(21), fixed bin(35));

call tape_ioi_$queue_write (tioi_id, buffer_ptr, data_len,
     code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — is a pointer to the buffer for which the write operation is to be queued. (Input)

3. data_len — is the length (in characters) of the actual data to be written from this buffer. (Input)

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

Entry: tape_ioi_$read

This entry point will queue a read operation for every ready buffer. It will then perform a check of the read operation that has been queued the longest. This is equivalent to calling tape_ioi_$queue_read for all available buffers, and then calling tape_ioi$check_read. For normal tape reading, this is the only read entry point that the caller needs.

Usage

```
dcl  tape_ioi_$read entry (bit(36) aligned, ptr, fixed
         bin(21), fixed bin, fixed bin(35));

call tape_ioi_$read (tioi_id, buffer_ptr, data_len, rx,
         code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — see tape_ioi_$check_read. (Output)

3. data_len — see tape_ioi_$check_read. (Output)

4. rx — see tape_ioi_$check_read. (Output)

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

Entry: tape_ioi_$set_buffer_ready

This entry point changes the state of an I/O buffer from suspended to ready. A program error will occur if the buffer is not in the suspended state. If a buffer is in the busy state (I/O is queued for this buffer) it can be changed to the ready state only by a check operation. Buffers can be changed to the null state only via the tape_ioi_$deallocate entry point.

Usage

```
dcl tape_ioi_$set_buffer_ready entry (bit(36) aligned, ptr,
     fixed bin(35));

call tape_ioi_$set_buffer_ready (tioi_id, buffer_ptr, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. buffer_ptr — is a pointer to the buffer to be set ready. (Input)

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

Entry: tape_ioi_$set_mode

This entry point will set any one of the TAPE_IOI modes. See Appendix C for more information about the TAPE_IOI modes.

Usage

```
dcl tape_ioi_$set_mode entry (bit(36) aligned, char(8),
     ptr, fixed bin(35));

call tape_ioi_$set_mode (tioi_id, mode, data_ptr, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. mode specifies the name of the mode to be set. (Input)

3. data_ptr is a pointer to data representing the setting of the mode. (Input)

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

Entry: tape_ioi_$stop_tape

This entry point will stop any I/O currently in progress. The results of any queued I/O operations are undefined. All buffers will be set to the ready state. This entry point is intended for use within a cleanup handler or a close procedure in order to guarantee that all queued I/O operations are stopped. It can also be useful when positioning the tape reel.

Usage

```
dcl  tape_ioi_$stop_tape entry (bit(36) aligned, fixed bin,
     fixed bin, fixed bin(35));

call tape_ioi_$stop_tape (tioi_id, count, rx, code)
```

where:

1. tioi_id uniquely identifies this TAPE_IOI activation. (Input)

2. count is the number of I/O operations that completed before the tape was physically stopped. If the tape was already stopped this number will be zero. (Output)

3. rx is the result index of this operation. Only values of -1 through 1 may be returned by this entry point. (Output)

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

Entry: tape_ioi_$write

This entry point is a useful combination of the tape_ioi_$queue_write and tape_ioi_$check_write entry points Optionally, it will queue a write operation for a specified I/O buffer. A program error will occur if the specified I/O buffer is already queued for writing or if any order command or read operations are queued.

If there are any suspended buffers, then write operations will be queued for all of these suspended buffers. Any buffer specified in this call will be queued first. Then the suspended buffers will be queued in the same order in which they were suspended. The data length and other mode specifications used for each suspended buffer will be the same as when the buffer was originally queued.

This entry point will also return a pointer to a ready I/O buffer. If the call to this entry point is successful, the caller may begin to copy output data into this buffer.

If there are no ready buffers, then this entry point will obtain one by performing a check of the write operation that has been queued the longest. If the check operation is successful, it will result in the buffer used for that write operation becoming ready. A pointer to this checked and newly readied buffer will be returned.

Usage

```
dcl  tape_ioi_$write entry (bit(36), ptr, fixed bin(21),
          ptr, fixed bin, fixed bin(35));

call tape_ioi_$write (tioi_id, qbuffer_ptr, data_len,
          rbuffer_ptr, rx, code);
```

where:

1. tioi_id — uniquely identifies this TAPE_IOI activation. (Input)

2. qbuffer_ptr — is a pointer to a buffer for which a write operation is to be queued. This value may be null. (Input)

3. data_len — see tape_ioi_$queue_write. (Input)

4. rbuffer_ptr — is a pointer to a ready buffer. If the result index indicates that a write operation was checked and an I/O error occurred, then this will be a pointer to the buffer in error. (Output)

5. rx — see tape_ioi_$check_write. (Output)

6. code — is a standard Multics system status code. (Output)

Appendix B
Tape Order Commands

TAPE ORDER COMMAND	MNEMONIC	COUNT	COMMAND DATA
Ready	rdy		1. Set by TAPE_IOI
Backspace One File	bsf	*	
Backspace One Record	bsr	*	
Forward Space One File	fsf	*	
Forward Space One Record	fsr	*	
Write End-of-File Record	eof	*	2. Set by Caller
Erase	ers	*	
Rewind	rew		
Rewind/Unload	run		
Tape Load	lod		
Request Status	rqs		3. Set by TAPE_IOI
Reset Status	rss		
Request Device Status	rqd		4. Set by TAPE_IOI
Reset Device Status	rsd		
Set 200 bpi density	den		5. Set by Caller
Set 556 bpi density	den		" "
Set 800 bpi density	den		" "
Set 1600 bpi density	den		" "
Set File Permit	per		
Set File Protect	pro		
Reserve Device	rsv		
Release Device	rel		
Read Control Registers	rcr		6. Set by TAPE_IOI
Write Control Registers	wcr		6. Set by Caller

* implies that "count" may be more than 1.

1. Data returned is 36 bits of special status. If no special system was generated by the previous I/O operation, then this field will be zero.
2. Optional, 6 bits that form a 1 character record.
3. Data returned is the TAPE_IOI hardware status structure.
4. Data returned is 24 packed 8 bit bytes.
5. Data specified is a fixed binary density setting.
6. Data is 8 2-byte counters (4 words).

Appendix C
TAPE_IOI Modes

MODE NAME	DECLARATION		VALUES	MEANING
align (1.)	bit(1)		"0"b	Left Aligned
			"1"b	Right Aligned
cif (2.)	bit(6)	*	"010000"b	No retry, high
			"010001"b	No retry, low
			"010010"b	No retry, high, +deskew
			"010011"b	No retry, low, +deskew
			"011000"b	Retry, high
			"011001"b	Retry, low
			"011010"b	Retry, high, +deskew
			"011011"b	Retry, low, +deskew
data	char(4)	*	"bin"	Specifies actual read or write commands used.
			"bcd"	
			"tap9"	
			"asc"	
			"ebc"	
			"a/e"	
event	fixed bin(71)			IPC Event Channel
length	bit(1)	*	"0"b	Normal
			"1"b	Special
recovery	bit(1)	*	"0"b	No Error Recovery
			"1"b	Error Recovery
wait	bit(1)	*	"0"b	Simplex
			"1"b	Multiplex

* implies default setting

1. Default alignment depends upon the model tape drive being used. Model 500 => right aligned, model 600 => left aligned.

2. This mode is ignored if error recovery is ON.

Appendix D
Result Index Summary
Special Events

The result indexes returned by TAPE_IOI are generated by interpreting the hardware status (IOM major and substatus) for the I/O operation last completed. The result index values are summarized below:

-1 => block (multiplex block mode)
0 => success
1 => program error
2 => unrecoverable I/O error
3 => recoverable I/O error
4 => special event
5 => special event
6 => special event

The special event result indexes are returned for different statuses depending upon the I/O operation. A summary of special events is given below:

Reads:	4 => EOF 5 => Blank Tape on Read 6 => Code Alert (Data modes asc, ebc, or a/e)
Writes:	4 => EOT 5 => recoverable error <u>and</u> EOT 6 => Code Alert (Data mode = tap9 or a/e)
Back Space:	4 => BOT 5 => EOF (backspace record only)
Forward Space:	4 => Blank Tape on Read 5 => EOF (forward space record only)
Request Status:	Only result index values (-1, 0, 1) are returned. Hardware status information returned in data structure.
Ready:	Only result index values (-1, 0, 1) are returned.
All Others:	No special events.

Appendix D
Result Index Summary
Status Classes

STATUS	STATUS CLASS
Subsystem Ready	
At BOT	AB (At Beginning)
ASCII Alert	CA (Code Alert)
All Others	OK (Result Index = 0)
Device Busy	
In Rewind	SI (Special Interrupt)
Device Loading	SI
All Others	UE (Unrecoverable Error)
Device Attention (All)	UE
Device Data Alert	
Blank Tape on Read	ET (End of Tape)
Timing Alert	DA (Data Alert)
All Parity Errors	DA
End of Tape	ET
ET and any DA	DE (Data, End)
End of File (All)	EF (End of File)
Command Reject	
Positioned at BOT	AB
All Others	UE
MPC Device Attention	
All	UE
MPC Device Data Alert	
PE-Burst, NRZI CCC	DA
Preamble, Postamble	DA
Multi-track, Marginal	DA
Code Alert	CA
All Others	UE
MPC Command Reject (All)	UE
Power Off	UE
System Fault	UE
IOM Central	
IOM -> PASI	IP (IOM -> PSIA)
PSIA -> IOM	DA
All Others	UE
IOM Channel (All)	UE

Appendix D
Result Index Summary
Table of Result Indexes

The table below shows the result indexes returned for each unique combination of I/O operation and status class.

OPERATION	STATUS CLASSES								
	AB	CA	DA	DE	EF	ET	IP	SI	UE
rbin	2	2	3	2	4	5	3	2	2
rbcd	2	2	3	2	4	5	3	2	2
rtp9	2	2	3	2	4	5	3	2	2
rasc	2	6	3	2	4	5	3	2	2
rebc	2	6	3	2	4	5	3	2	2
ra/e	2	6	3	2	4	5	3	2	2
wbin	2	2	3	5	2	4	3	2	2
wbcd	2	2	3	5	2	4	3	2	2
wtp9	2	6	3	5	2	4	6	2	2
wasc	2	2	3	5	2	4	3	2	2
webc	2	2	3	5	2	4	3	2	2
wa/e	2	6	3	5	2	4	3	2	2
eof	2	2	3	5	2	4	3	2	2
ers	2	2	3	5	2	4	3	2	2
bsr	4	2	2	2	5	2	2	2	2
bsf	4	2	2	2	0	2	2	2	2
fsr	2	2	2	2	5	4	2	2	2
fsf	2	2	2	2	0	4	2	2	2
rqs	0	0	0	0	0	0	0	0	0
rdy	0	1	1	1	1	1	1	-1	1
others	2	2	2	2	2	2	2	2	2

Appendix E
Sample Scenarios
Activation and Reading

```
/*   Process options to build rcp_tape_info. */

     call tape_ioi_$attach (tape_info_ptr, tai_ptr,
                            comment, rcp_id, cd);
     if   cd ^= 0 then goto ERROR;

/*   Data in tioi_activate_info may be changed or completed here. */

     tai.actv_name = "example"
     call tioi_$activate (tai_ptr, tioi_id, cd);
     if   cd ^= 0 then goto ERROR;

     recovery = "1"b;                   /* Error recovery ON. */
     call tioi_$set_mode (tioi_id, "recovery", addr(recovery), cd);
     if   cd ^= 0 then goto ERROR;

     call tape_ioi_$allocate_buffers (tioi_id, 4096, 0, act_len,
                                      act_num, buf_ptrs, cd);
     if   cd ^= 0 then goto ERROR;

READ_LOOP:
     call tape_ioi_$read (tioi_id, buf_ptr, data_len, rx, cd);
     goto READX (rx);                   /* RX => decision. */

READX(0):                               /* Read succeeded. */
     call PROCESS_INPUT (buf_ptr, data_len);
     goto READ_LOOP;

READX(1):                               /* Program error. */
READX(2):                               /* Unrecoverable error. */
READX(3):                               /* Recovery ON => error. */
     goto ERROR;

READX(4):                               /* End of File. */
READX(5):                               /* Blank tape => EOF. */
     goto END_OF_FILE;
```

Appendix E
Sample Scenarios
Writing and Deactivation

```
/*   Attach and activate, error recovery ON. */

     buf_ptr = null();                    /* Initialize. */
     rx = 0,  data_len = 1;

     do   while ((rx = 0) & (data_len > 0));
          call tape_ioi_$write (tioi_id, buf_ptr, data_len,
                                next_buf_ptr, rx, cd);
          buf_ptr = next_buf_ptr;         /* If OK fill buffer. */
          if   rx = 0 then call FILL_BUF (buf_ptr, data_len);
     end;

     goto WRITEX (rx);                    /* Out of loop, rx => why. */

WRITEX(0): STOPX(0):                      /* No more data, flush. */
     call tape_ioi_$check_write (tioi_id, buf_ptr, rx, cd);
     goto STOPX (rx);                     /* Check the check. */

STOPX(1):                                 /* Program error. */
     if   cd ^= error_table_$device_not_active
     then goto ERROR;

/* No more buffers queued, write EOF to close file. */

     call tape_ioi_$order (tioi_id, "eof", 1, null(), x, rx, cd);
     if   cd ^= 0 then goto ERROR;

     call tape_ioi_$deactivate (tioi_id, error_count, cd);
     call tape_ioi_$detach (rcp_id, "0"b, error_count, cd);
     return;

WRITEX(1):                                /* Program error. */
WRITEX(2): STOPX(2):                      /* Unrecoverable error. */
WRITEX(3): STOPX(3):                      /* Recovery ON => error. */
ERROR:
     [ - - - - - - - - - - - - - - - ]

WRITEX(4): STOPX(4):                      /* End of Tape */
WRITEX(5): STOPX(5):                      /* EOT and error. */
END_OF_VOLUME:
     [ - - - - - - - - - - - - - - - ]
```