

TO: Multics Repository List
FROM: M. L. Goudy
SUBJ: Interim I/O Document
DATE: December 20, 1968

Because of rapid fluctuation in I/O design, several attempts to publish a comprehensive document on I/O have failed. This document is an approximation of Multics I/O as it was conceived in early 1968. The document discusses concepts which have been deleted entirely (e.g., transaction blocks), concepts which will not be implemented at least for some time to come (e.g., reservation and resource management), and concepts that are undergoing redesign (e.g., the GIM). However, it is believed that the document might have some value as a historical record and a source of ideas for future system design.

The following is an attempt to give the reader some informative tips about current use of the document.

Section V on reservation of devices will not be implemented for some time, and it is likely that many of the concepts will change.

The concepts of ioname, iopath, attachment, and the I/O switch, discussed in Sections VI and VII, are basic to understanding the Multics I/O system.

Section IX discusses I/O transactions. All these concepts have been deleted from the system.

The discussion of the GIM in Section XII is valid through 1968; however, the GIM is being redesigned. The data bases discussed will probably not change drastically.

The information in Section XIII on the GIOC is a useful summary of the GIOC and its control words, especially for programmers who must write some sort of device interface module. It is not expected that this information will change.

I/O SYSTEM INTERMEDIATE DOCUMENT

TABLE OF CONTENTS

I.	Purpose	1
II.	I/O From the User's Viewpoint	1
III.	I/O From the System Point of View	1
IV.	Internal Structure of the I/O System	4
V.	Reservation of Devices	6
	A. Relation of I/O and Reservation	6
	B. The Reservation Procedure (Resource Management)	7
VI.	Attachment and Detachment of Devices	8
	A. Attachment	8
	B. Detachment	11
VII.	Relations Structure of I/O Attachment Modules	11
	A. Function Description	11
	B. The Iopath	14
	C. The I/O Switch	14
VIII.	Internal Structure of the I/O System	16
IX.	I/O Transactions	16
	General Statement on Transactions	16
	A. Relative Pointer Use	18
	B. The IS Header	19
	C. The Per-Ioname Base (PIB)	21
	D. PIB Extensions	25
	E. The Interprocess Communications Block	26
	F. Introduction to Transaction Block Discipline	26
	G. The Transaction Block Segment	28
	H. Transaction Block Allocation	30
	I. Transaction Block Holding	31
	J. Chaining of Related Blocks	34
	K. Outer Module Chaining Responsibilities	38
	L. Calls to Set Transaction Block Items	38
	M. Transaction Block Segment Switching	39
	N. Buffer Discipline	39
	O. Example of Transaction Block Use	40
	P. DSM Interface	42

X.	Processing of Data Within the I/O System	45
A.	Data Representation	45
1.	Linear Representation	45
2.	Sectional Representation	45
3.	Physical Representation	46
B.	Logical Divisions of Data	46
1.	Linear and Sectional Frames	47
2.	Random Frames	48
C.	Data Delimiting	48
1.	Establishing Delimiters	48
2.	Finding Bounds of a Frame	48
D.	Data Access	49
E.	Reading and Writing Data	49
F.	Iopath Modifications	50
XI.	Functions of Outer I/O Modules	60
A.	DSM	60
B.	DCM	60
C.	File System Interface Module	61
XII.	Hardcore I/O (The GIM).	62
A.	GIM Functions	62
B.	DCM/GIM Interface	62
C.	Status Information and the GIM	65
D.	Hardcore I/O System Data Bases	66
1.	Static Storage	66
2.	Channel Assignment and Status Table CATCST	66
3.	GIOC Mailbox Areas	66
4.	GIOC DCW Areas	67
5.	GIOC Data Area	67
6.	Channel Copy Table	68
XIII.	GIOC	68
A.	General Information	68
B.	Channels and Control Words	70
1.	Types of Channels	70
2.	Control Words	71
3.	Direct and Indirect Channels	71
4.	Control Word Functions	72
a.	Connect Operand Word (COW).	72
b.	Instruction Pointer Word (IPW).	72
c.	Channel Instruction Word (CIW).	73
d.	List Pointer Word (LPW)	73
e.	Data Transfer Control Word (DCW).	73
C.	I/O Hardware/Software Interface	77

I/O SYSTEM
INTERMEDIATE DOCUMENT

I. PURPOSE

The I/O system contains the procedures by which processes communicate with external devices.

II. I/O FROM THE USER'S VIEWPOINT

When the user logs into the Multics system (via the command subsystem), the aspects of getting information to and from peripheral and terminal devices are handled by the I/O modules of the Multics system and by the 645 hardware. The user's view of I/O is conditioned by the making of requests at the console to read, write and otherwise manipulate files or data which have been given symbolic names. The I/O system interprets these requests and transforms them from the symbolic language level down to the bit-picking machine language and physical hardware level.

Users of Multics whose I/O is confined to a single console and whose programming is confined to the manipulation of files are not concerned with the I/O system directly. The interfaces of console/device-to-process are handled within Multics, and the drum is not accessed through I/O. Users with needs requiring other I/O devices (card readers, printers, magnetic tapes, etc.) will have direct interest in the I/O system. For users of these devices, some information on requests to read, write, and delete information from files on these devices is required.

III. I/O FROM THE SYSTEM POINT OF VIEW

Systems programming also entails the use of logical concepts rather than physical

concepts. The systems programmer is generally cognizant of I/O in terms of symbolic attach and detach calls to associate (and disassociate) symbolic names (called ionames) with the names of specific devices.

The I/O system works in conjunction with the GIOC (Generalized Input Output Controller) to effect the transfer of information to and from peripheral and terminal devices to and from storage. A general illustration of the working relationship between the 645 processor, storage, the GIOC, and the peripheral and terminal devices is given in Figure 1. The working relationship is significant because the manner of data input/output transfer is markedly different from that of most other computing systems. Instead of the central processing unit of the machine accessing the data on the peripherals (as it does in most other machine designs), a discrete unit of hardware, the GIOC, has access to the data on the peripheral devices. The ramifications of this design are shown in Figure 8-1.

Of additional significance is the close coordination of hardware (the GIOC) and software (the I/O system modules). Upon initial examination of Multics I/O, it is difficult to determine any interfaces between the user, the software, and the hardware because I/O appears to be a monolithic logical unit, when actually I/O consists of many modules which function as parts of the user's working process-groups in Multics (and system process groups). For literary convenience, the user aspects of I/O, the discrete software modules of the I/O system and the discrete units of hardware, are often discussed in an isolated manner. The reader should always be aware that any portion of I/O about which he is reading functions as a part of the whole system.

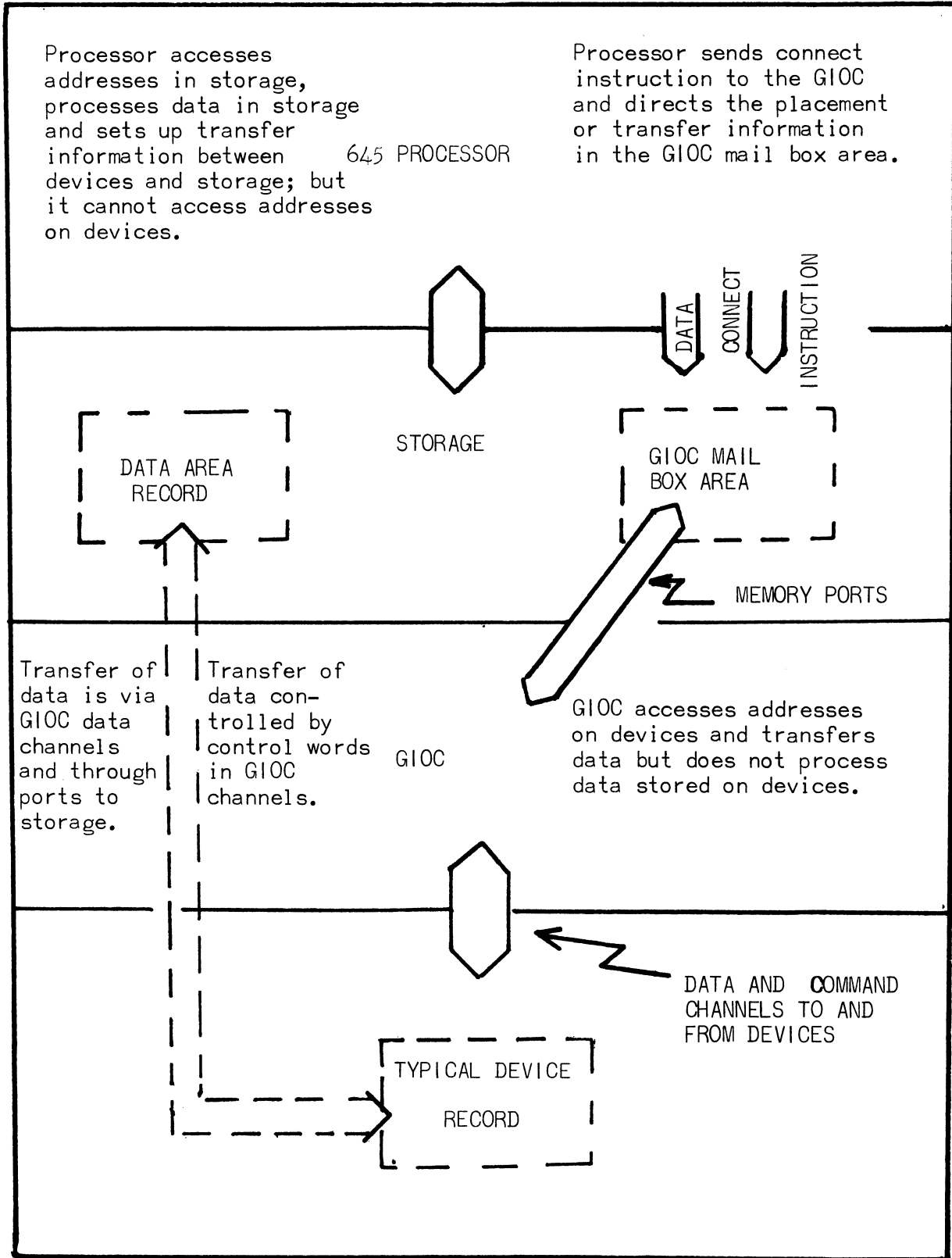


Figure 1. Schematic of I/O Processing

IV. INTERNAL STRUCTURE OF THE I/O SYSTEM

The Multics I/O system may be divided into two general categories, hardcore I/O and outer I/O. This division is made on the basis of the protection rings which the modules of the I/O system occupy. Hardcore I/O consists of the GIOC interface module (GIM), which is located in the hardcore ring (ring 0) of the Multics system. The outer I/O modules consist of the rest of the I/O system, and these modules are in rings outside of the hardcore ring. The outer I/O modules may be further differentiated by the outer rings they are in and by the process group to which they belong. The I/O modules which interface with the GIM belong to the universal device manager process group (UDMPG), and are in ring 1 (the administrative ring) of the Multics system. The modules which interface most directly with the user processes are a part of the user's process group and are in the applicable user ring (ring 32 or greater). The I/O module which coordinates the interaction between all of the outer modules of the I/O system, the I/O switch, is in both the administrative ring with the UDMPG and the user ring, with the other I/O outer modules. Figure 2 shows the major portion of the I/O system modules in schematic form.

When a user at the console gives a command for a read or a write in the form:

write (name1,...) ,

the name used in the I/O command (name1) is identified by the I/O system as an "ioname." This ioname is used by the I/O switch (one of the outer I/O modules) to route calls within the I/O system. The general strategy of the I/O system is to attach (associate) one ioname with another ioname, which is a symbolic name for data or the name of a device. For any ioname, an associated ioname is specified by each attachment. An iopath is a chain of ionames implied by a

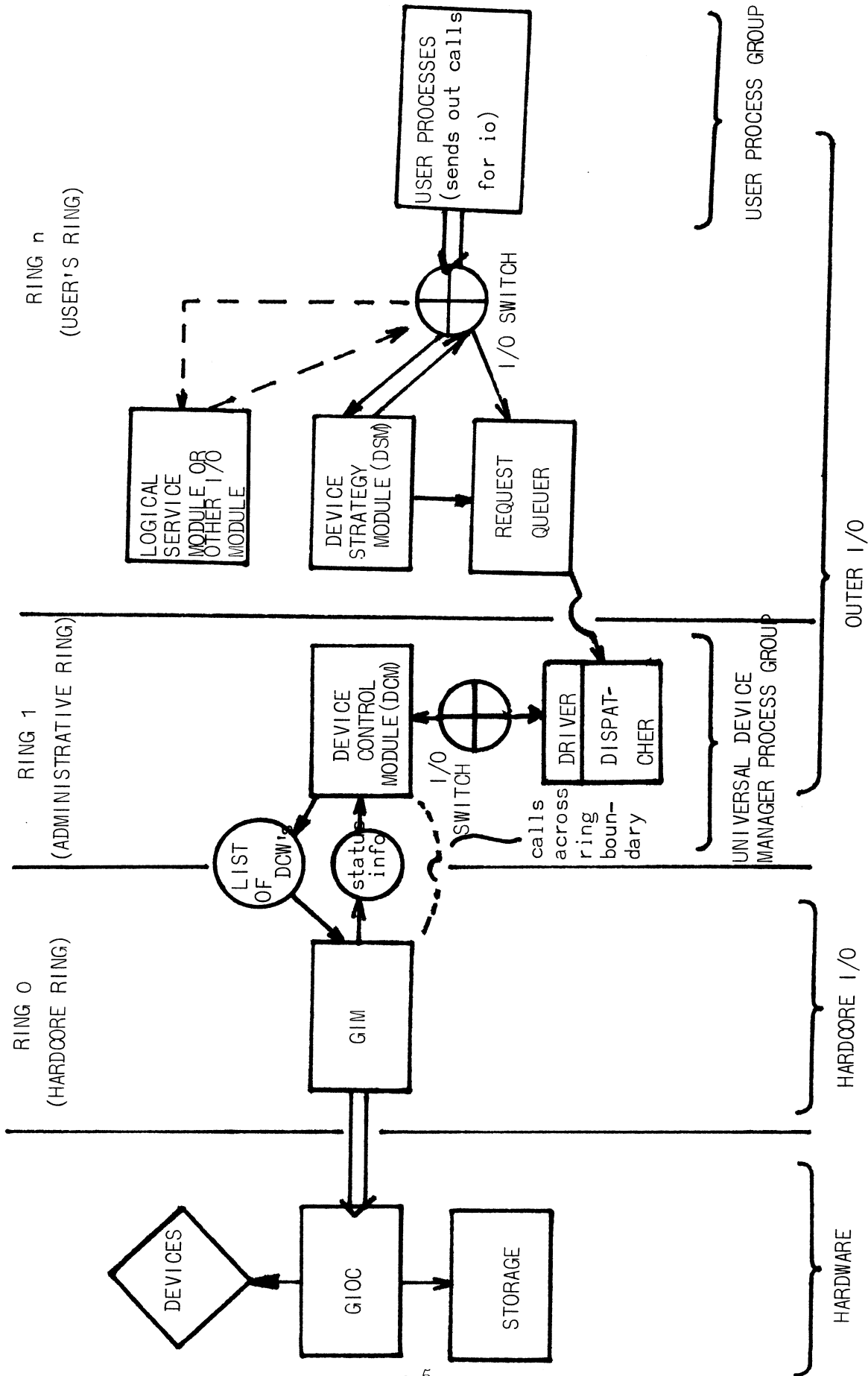


Figure 2. General View of I/O Structure

given chain of attachments followed by the GIM (the hardcore I/O module) or by the file system interface module (another outer I/O module). The establishment of an iopath from the symbolic command level to the physical device level or the file system is the object of the procedures within the I/O system.

The user's chief concern with I/O is to first reserve a device and/or the media that goes with the device, then assign an ioname to the data stream, frame or item to be processed by the I/O system, and then attach the ioname to an outer module of the I/O system. The ioname may be attached to an outer I/O module to perform strictly an input or output of the data designated by the ioname, such as attaching an ioname to the tape DSM (Device Strategy Module) which would eventually cause the data to be read or written from tape. On the other hand, the ioname may be attached to an outer I/O module which will cause the data to be processed in some manner such as attaching the ioname to a logical service module to superimpose a logical record structure on an input stream of data.

V. RESERVATION OF DEVICES

A. RELATION OF I/O AND RESERVATION

Almost any use of I/O requires advance reservation of devices. For those devices that require advanced reservation (e.g., tape drives, printers), the user must obtain advanced reservation before attempting to attach and run the device through the facilities provided in the I/O system. These reservations are not provided by the I/O system, but are obtained from the Multics system supervisor resource assignment module. Reservations are made through the resource management modules of Multics. These modules function to handle the dedicated assignment of system resources (devices, media, and time). While the resource management modules are not intrinsic parts of the Multics supervisor, they are logically considered as being a part of the supervisor.

Device types that require advance reservation are:

tape drives

line printers

card readers and punches

some typewriters

some communications lines

some directly referenced GIOC channels.

In addition to the type of devices reserved, media that are used on the devices, such as tape reels, can be reserved. (Thus, devices and/or media are termed system resources.) The resources most often reserved (dedicated) for exclusive use by a user are detachable-type storage media, such as tape reels and off-line devices (such as tape drives), because, characteristically, the random user demand for these devices exceeds the availability of the devices. In addition, resources which are normally shared may also be reserved. (For instance, a percentage of the capacity of a processor or a teletype could be reserved for a particular user).

B. THE RESERVATION PROCEDURE (RESOURCE MANAGEMENT)

Reservations are made and are released by calling the reserver module of resource management with the call: reserve. When reserving a device, the user specifies the device and the time during which the device is to be used. For instance, it may be desired to have a tape drive and three tapes available between two and three o'clock on a given date in order to run a sort. The user can call the I/O system and attach and detach devices without affecting a reservation. For example, detaching a tape does not release the reserved tape drive, and the tape detached could be re-attached at a later time, within the time limit of the reservation that was made for the tape. Reservations are released by calling the resource assignment module of the supervisor, by reaching expiration of the time allotted

for the reservation, or by the detection of certain fault or error conditions. The resource assignment module performs a protection function as well as a device reservation function. Thus a user cannot reserve a device for which he does not have authorized access. Figure 3 briefly outlines the functions of resource assignment at time of reservation and at time of use of some resource. It does not show the return paths. However, the user is notified whether or not he can reserve a device for a particular time, and is also notified at time of use when the device is actually available.

VI. ATTACHMENT AND DETACHMENT OF DEVICES

A. ATTACHMENT

To use a device once it has been reserved, the user issues some call that will cause an attach call to be generated. Usually the call will be a read or a write call, although in some cases the user himself will issue an attach call.

Attachment is the establishment by an attach call of an association between two symbolic names used as arguments in the call. The first is the symbolic name of the data to be processed (whether in the form of a data stream, a data item, or a frame of data), and the second is the symbolic name of an I/O system module. If the user is reading or writing data, the I/O system module named will be that device strategy module (DSM) that handles I/O for the particular type of device the user has reserved, e.g., a tape DSM for a tape drive. The symbolic name of a data stream can be associated by an attach call with other I/O modules also. For example, the user may wish to superimpose a logical record structure on an input stream of data. To do so, he would attach the symbolic name of his data stream to the I/O module called the logical record formatter. The symbolic names for data and I/O modules are called ionames.

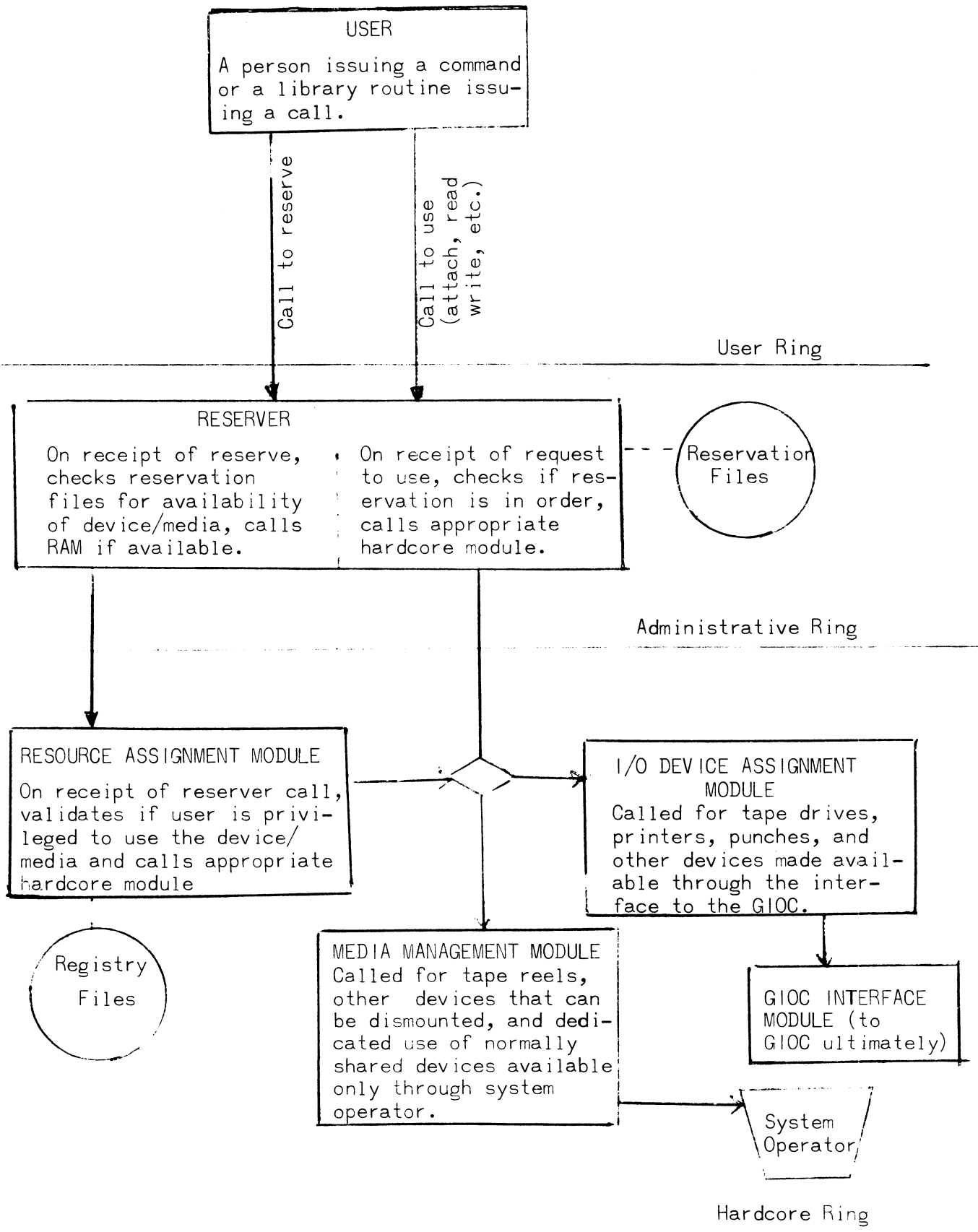


Figure 3. Resource Assignment

When a user at the console gives a read or write command of the form:

read (name1,...) or write (name1,...)

the name used in the command is identified by the I/O system as the ioname of the place from which data is read or into which it is written. The system then generates an attach call of the form:

call attach(ioname1,type,ioname2,mode,status)

The ioname1 argument of an attach call identifies the stream, frame or item of data. The ioname2 argument is the outer I/O module to be associated with the data designated in ioname1. The type argument specifies the type of device, and the mode argument specifies the media on the device or the selection of a number of logical services to apply to the data.

An attach call causes an entry on the two ionames to be placed in the I/O data base known as the attach table. A general block diagram of initial I/O system action when an attach call is received is shown in Figure 4.

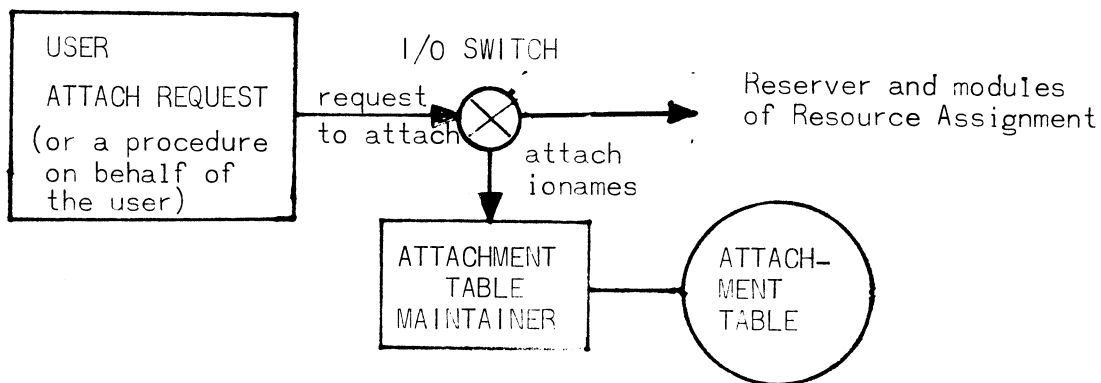


Figure 4 Initial I/O Action During Attachment

The I/O system establishes a chain of calls through the I/O system to the physical device that permits the requested I/O to be performed. The chain is called an iopath.

B. DETACHMENT

Detachment is the disassociation of the symbolic name of data and the symbolic name of an I/O system module. Detachment occurs when:

1. The time for which a device was reserved expires.
2. Certain fault or error conditions occur that preclude continuation of execution of the user's job.
3. A detach call is made by the user or by the I/O system following completion of the user's job.

Detachment does not necessarily release a reserved device. For example, a user can issue a detach call and later re-attach the device if the time for which the device was reserved has not expired.

VII. STRUCTURE AND RELATIONSHIP OF ATTACHMENT MODULES

A. FUNCTION DESCRIPTION

When an ioname is used for the first time, it is "unknown," and therefore not in the per process group attach table. An outer I/O module (the not_founder) temporarily appears as the first outer module of the iopath. Its function is to make an attach table entry for the "unknown ioname." While in the path, the not_founder determines the next outer module to call (via the I/O switch) from the type argument in the attach call. The next outer module negotiates to have itself replaced, or may splice other outer modules before or after itself in the iopath. Upon return to the user, the basic iopath for the particular ioname has been established, and a representation of the iopath is embedded in the attach table. A unique entry in the attach table corresponds to each appearance of the I/O switch in the basic software path. There is only one entry per ioname. Each entry associates an ioname with a unique outer module in the path. Each outer module "knows" the ioname associated with the next outer module in the path.

When the user issues an outer call, the I/O switch uses the specified ioname to reference the attach table. The I/O switch forwards the call to the outer module associated with the ioname. Outer modules drive control along the software path by issuing calls, which specify the ioname associated with the next outer module in the path, to the I/O switch.

Along with the I/O switch and the not_founder, three other outer I/O modules function to provide the user with facilities to attach ionames and establish an iopath. The ATM (attach table maintainer) services all requests to reference and/or update the attach table; this involves special action when the services of the not_founder are required. When an attempt is made to retrieve a pointer to an entry point vector (EPV) associated with an ioname which is not contained in the attach table, the ATM returns a pointer to the entry point vector for the not founder, which allows the not_founder to create an attach table entry for the ioname. Also, the ATM calls the type table maintainer (TTM) in order to force linking of the not founder.

The TTM services all requests to reference and/or update the type table (TT), which associates the type argument with the corresponding outer I/O module.

Figure 5 illustrates the internal I/O system processing during attachment for the call:

```
call attach (henry,tape,reel432,ws)
```

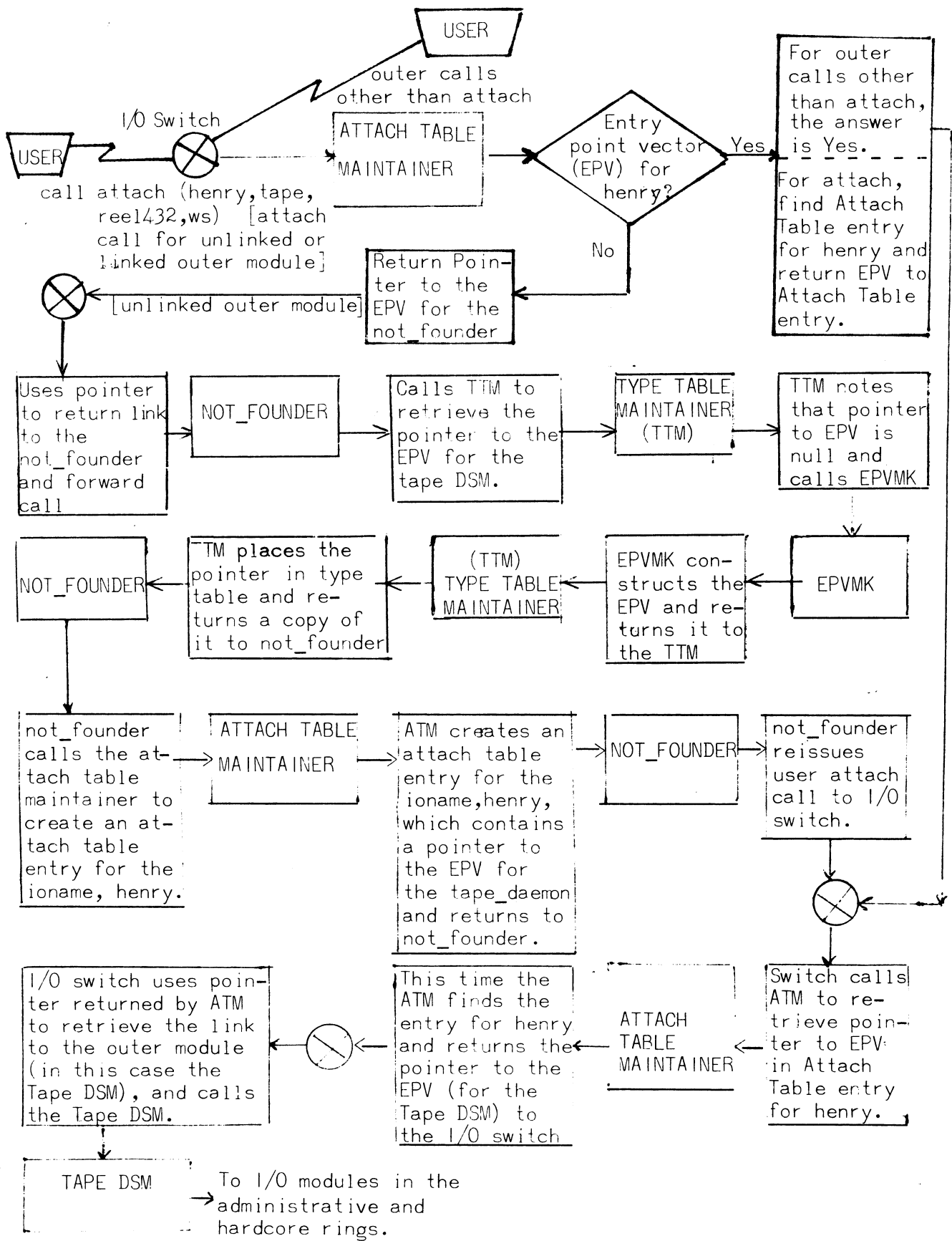


Figure 5. I/O System Internal Processing of Attach and Other Outer I/O Calls

B. THE IOPATH

By a series of attachments between the ioname of the user's stream, frame, or item of data and the ionames of outer I/O modules (as they are needed), the I/O system establishes a path of calls between those modules of the I/O system that are needed to perform the processing and input/output requested for the particular ioname. This path of attachments is called an iopath. The iopath is determined by the type and mode arguments of the attach call, and it can be subject to variation. The I/O switch module appears recursively throughout an iopath and gives calls to those I/O modules needed to associate the ioname in the initial attachment with other ionames which route the processing and/or transfer of data in the manner prescribed by the user and ultimately associate the penultimate ioname in the iopath with the ioname of a device strategy module (DSM) or the ioname of the file system interface module (FSIM). (The functions of both these outer modules are discussed later in this document.) A typical iopath is illustrated in Figure .6 .

C. THE I/O SWITCH

Whenever one I/O module calls another I/O module (within the same ring), the I/O switch performs its function, which is the routing of control from module to module in the I/O system as the individual modules are required. Besides the GIM, DCM, and DSM, the I/O system offers a number of logical services; these services are implemented by other outer modules. Structurally, each outer I/O module is a segment with an entry point corresponding to each call from other outer I/O modules which is meaningful to the segment. When the I/O switch receives a call referencing an ioname, it references a data base known as the per ioname segment (IS) to ascertain which I/O module to call next and repeats this process until all the necessary steps are taken to attach the appropriate device with the ioname (which initiates the setup of DCW's by the DCM, which in turn issues the DCW's to the GIM to perform I/O data transfer).

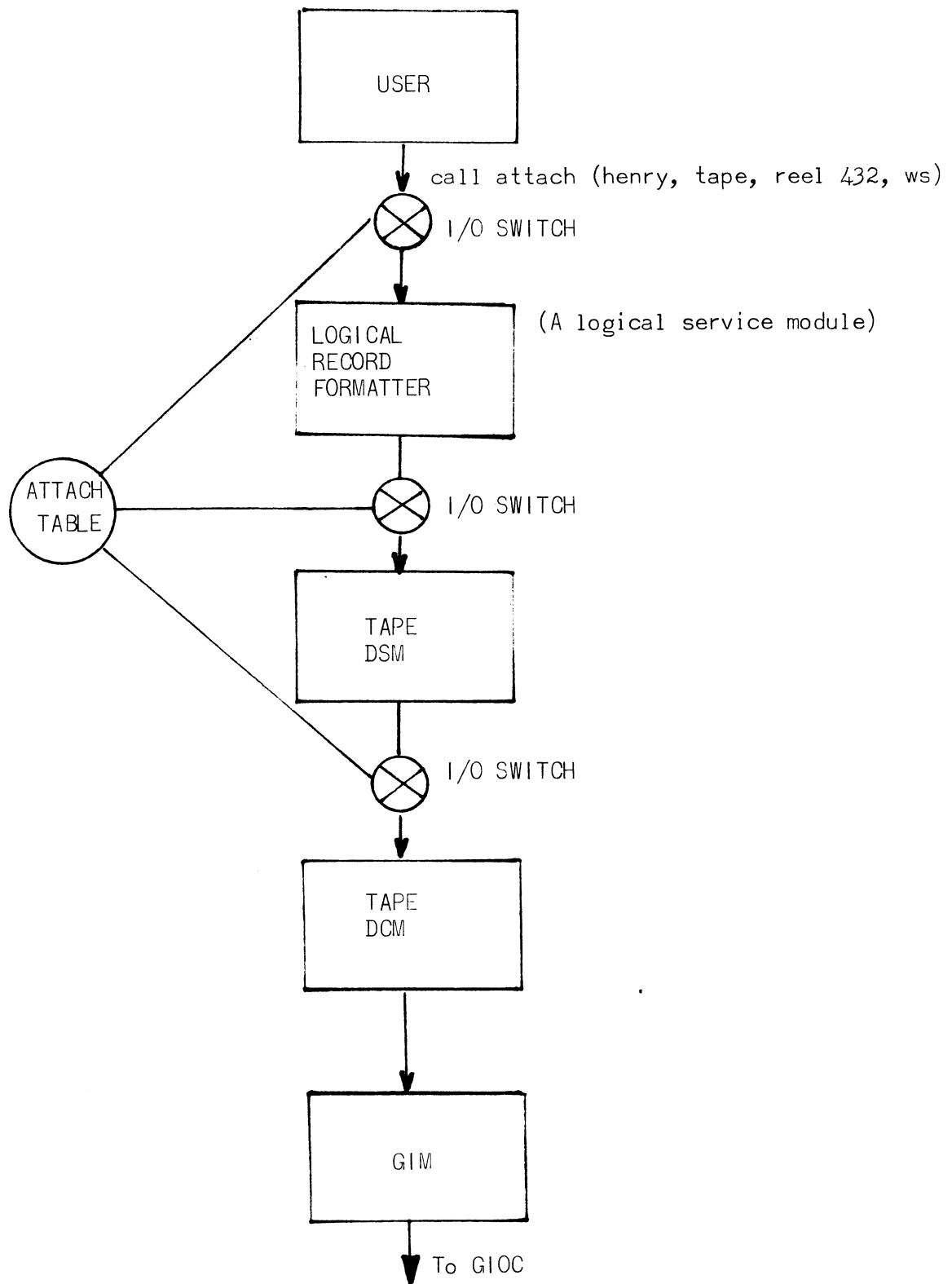


Figure 6. A Typical Iopath

In Figure 6 the I/O switch is shown in both the user ring and the administrative ring. Since the I/O switch is used to route I/O calls from one module to another in the establishment of an iopath, and since the I/O switch is used to interface to the user on calls to I/O, the I/O switch is called upon several times in any iopath.

VIII. INTERNAL STRUCTURE OF THE I/O SYSTEM

The iopath established during device attachment threads from I/O modules operating as part of the user's process group in the user ring to I/O modules operating as part of the universal device manager process group (UDMPG) in the Multics administrative ring and from there to the GIOC interface module (GIM) operating in the Multics hardcore ring. The GIM interfaces with the peripheral devices. Figure 2 on page 5 showed a highly simplified version of the modules and their relationships.

IX. I/O TRANSACTIONS

At the time an ioname is defined as a result of an attachment, the I/O switch creates a per-ioname segment (IS) and establishes data bases within it. The IS continues to exist until the outer module associated with the ioname is detached. The most significant components of the IS are the per-ioname base (PIB), the transaction block, and the interprocess communication block. The PIB is the principal data base of the outer module, which is reached via the ioname attachment. The interprocess communication block is the common data base for the attachment module, the DSM's request queuer and the device manager dispatcher. The latter two outer I/O modules function to permit communication between I/O modules in different process-groups (and different rings) as is discussed later in this chapter. The transaction blocks provide the basic mechanism for maintaining the necessary history of I/O transactions to preserve the relationship between these.

For every outer call reaching an outer I/O module, except the divert call, the I/O switch allocates a transaction block within the transaction block segment by calling a module known as the transaction block maintainer (TBM). Transaction blocks are chained together and indices (pointers) describing this chain are stored in the PIB. The contents of transaction blocks are accessible by calls provided by the TBM. Additional per transaction data is kept in transaction block extensions (TBE's). These are allocated within an area in the PIB by the outer module called to perform an I/O transaction. A relative pointer to the TBE is kept in the corresponding transaction block. The TBM provides for de-allocation of unwanted transaction blocks together with their TBE's.

Each outer module may create additional transaction block chains for internal purposes. For example, the buffering of data being processed by the I/O system is implemented by such a chain. The outer module allocates (by a call to the TBM) a transaction block to which the buffer is appended in the form of a transaction block extension.

A. RELATIVE POINTER USE

Once an ioname is known to a user process-group, a procedure in any of the processes in the group can issue an outer call directed at that ioname. Thus an iopath is exercisable within any process in the group.

As a consequence, pointer information cannot be conveniently stored in the various IS data bases and must be referred to by the use of relative pointers, which are bit strings of length 18 and are easily connected to and from pointers by use of the Multics procedure, relptr. Pointer variables which are not easily computed by an outer module are provided by the switching complex which determines and keeps the pointer variables for every process using an ioname. For example, a pointer to the PIB itself would be determined by the I/O switch.

The IS contains a number of separately-allocated data bases. All of these data bases are accessed as "based" structures. The data bases in the IS are:

1. Header; created and initialized by the switching complex.
2. Standard Per-Ioname Base (PIB); allocated by the switching complex.
3. Standard PIB Auxiliary Based Storage; allocated by the outer module.
4. PIB Extensions (PIBE's); allocated by the outer module.
5. Transaction Block Extensions (TBE's); allocated by the outer module.
6. Interprocess Communication Block (ICB); allocated by the attachment module in a Device Strategy Module (DSM)

All data bases allocated by the outer module are allocated within an area in the PIB. The header, PIB, the PIB's auxiliary based storage, and the ICB have standard declarations which are given later in this section. Although the PIB and TBE extensions are intended for outer-module-dependent data, a portion of their declarations is standardized to enforce standard chaining of these extensions. The standardized chaining of extensions permits an outside procedure -- e.g., a dump routine -- to determine where all the data in the IS is.

B. THE IS HEADER

The IS header contains process-independent per-ioname data of interest primarily to the I/O switch. In particular, an outer module need not be concerned with the header. The only exception to the latter statement occurs in a DSM in which the attachment module and the request queuer use information in the header.

The header is created and initialized by the switching complex at the time the IS is created. The declaration for the header follows:

```
dcl 1 ioseg based (p),          /*per-ioname segment header*/
  2 flag,
  3 delayed_detach bit (1),    /*if ON, IOSW deletes ioname */
  3 restart bit (1),          /*if ON, IOSW issues upstate*/
  2 relp,                     /*relative pointers*/
  3 pib bit (18),             /*relp to PIB*/
  3 icb bit (18),             /*relp to ICB*/
  2 lock_list,                /*lock-list for ioname node*/
  3 n fixed bin,              /*size of key list*/
  3 chan_keys (N),            /*channel key list, n=N*/
  4 proc bit (36),            /*process locking or queued*/
  4 event bit (36);           /*queued process wake-up event*/
```

The "delayed-detach" switch is used by the switching complex to implement delayed deletion of the ioname and the IS. The "restart" switch is used to implement a part of the restart strategy. The relative pointer to the PIB is used by the I/O switch to supply the outer module with a real pointer to the PIB (see discussion below). The relative pointer to the ICB is used by the attachment module and the request queuer in a DSM.

The "lock_list" is the standard Multics lock list used when calling the Locker. It is used by the I/O switch to lock the ioname. When the switch receives an outer call, it attempts to lock the attachment graph node corresponding to the ioname by calling the Locker; an argument in the call is a pointer to the lock list. The ioname remains locked until the switch is about to return to the original caller, at which time the lock is released by another call to the Locker. This ioname locking strategy prevents simultaneous use of an ioname and the corresponding per-ioname segment of an iopath by procedures in more than one process.

C. THE PER-IONAME BASE (PIB)

The PIB is the principal data base of an outer module. The PIB is allocated and initialized by the switching complex when the per-ioname segment (IS) is created. The PIB is accessed as based storage by a pointer supplied by the I/O switch with every call. Additional per-ioname data is allocated by the outer module in the form of PIB extensions (PIBE's), discussed later.

The pointer to the PIB (pibp) is supplied by the I/O switch to the outer module as an additional last argument on every outer call routed to the outer module. In addition, the I/O switch initializes two groups of PIB items prior to routing each call. One group consists of all pointer items; the second consists of certain items that are copied from the caller's PIB. Per-call initialization is discussed later. The contents of the standard PIB are chosen to meet the common needs of a majority of outer modules. The PL/1 declaration is:

```
dcl 1 pib based (p),
  2 sync_event bit (36),
  2 error_event bit (36),
  2 dtabp1 ptr,
  2 dtabp2 ptr,
  2 dtabp3 ptr,
  2 auxptr ptr,
  2 ioname1 char (32),
  2 typename char (32),
  2 ioname2 char (32),
  2 bmode bit (72),
  2 next_ioname char (32),
  2 nmore fixed bin,
  2 elsize fixed bin,
  2 readbit fixed bin,
  2 writebit fixed bin,
  2 lastbit fixed bin,
  2 boundbit fixed bin,
  2 nbrk fixed bin,
  2 ndelim fixed bin,
  2 relp,
  3 pibe bit (18),
  3 more bit (18),
  3 brk bit (18),
  3 delim bit (18),
  2 chain_base,
  3 t1index bit (18),
  3 t2index bit (18),
  3 b1index bit (18),
  3 b2index bit (18),
  3 a1index bit (18),
  3 a2index bit (18),
  2 loarea area ((MAX));

/*standard PIB*/
/*for sync management*/
/* " */
/*driving table ptr 1, mode control*/
/*driving table ptr 2*/
/*driving table ptr 3*/
/*auxillary outer module ptr*/
/*ioname1*/
/*type name*/
/*ioname2*/
/*mode bit string*/
/*next ioname*/
/*number of additional next ionames*/
/*element size*/
/*read pointer in bits*/
/*write pointer in bits*/
/*last pointer in bits*/
/*bound pointer in bits*/
/*number of break delimiters*/
/*number of read delimiters*/
/*relative pointers*/
/*relp to PIB extension*/
/*relp to additional ionames*/
/*relp to break list*/
/*relp to delimiter list*/
/*base of call transaction block chain*/
/*base of buffer chain*/
/*base of aux transaction block chain*/
/*outer module allocation area*/
```

The sync_event and the error_event are used for internal synchronization management; modules not concerned with such matters can ignore these items except for one requirement. These items, when present (not zero), must be passed along to the next outer module. The mechanism for doing this consists of the module adding its "pibp" as an extra argument on outer calls it issues. Upon receiving an outer call with a "pibp," the I/O switch copies these two items into the callee's PIB before passing on the call. Thus outer modules must always include their "pibp" as an extra argument on every call issued to the next outer module in the iopath. If the I/O switch does not receive a "pibp," it zeros these items in the callee's PIB. Thus the I/O switch provides automatic forwarding of these synchronization management items.

There are two exceptions to the rule that an outer module must include its "pibp" in its calls. First, the module in the iopath that is the ultimate recipient of these items (the DSM) does not forward them. Second, the "pibp" must not be included on calls representing "incidental" input/output, i.e., calls issued to an ioname which is not a next ioname along the module's iopath.

The driving-table pointers and the auxiliary pointer are initialized by the I/O switch on every call, so that they are appropriate to the process in which the call is being made. The driving table pointers point to the first word of the segments containing the driving tables. The pointers are determined by the switching complex from driving table names kept in the Type Table. This mechanism is provided to permit outer modules to be table-driven where possible. One kind of driving table used by all outer modules is the mode control structure; dtabp1 in the PIB is reserved for this table. The code conversion tables and the code conversion module are good examples of the table-driven module approach.

The auxiliary pointer, "auxptr," is used to reference an external segment known only to the outer module. Such use of an external segment is permitted only in approved cases. One case involves the use of the I/O registry files by the DSM and the DCM. Another example of an external segment is the common data base shared by DCM's which operate devices connected to shared channels. The File System Interface Module (FSIM) uses auxptr to access the File System file. Since the switching complex does not know the identity of the external segment, it is up to the outer module to compute auxptr. Upon a return from the outer module, the I/O switch saves the value of this pointer in a per-process entry of the Attach Table. On subsequent calls to the module, the switch restores the pointer to the value previously saved for that process. If upon return the I/O switch notes that a restored auxptr has been modified by the outer module, the values saved for the other processes are made null. If no previous value had ever been saved, a null pointer is used for the initial value. Outer modules using auxptr should always test for a null pointer before the first use during a call.

"ioname1", "typename", and "ioname2" correspond to the first three arguments of the original attach call received by the outer module. Their values are assigned by the outer module at attachment time. "bmode" is the mode string array returned by the Mode Handler.

After the outer module determines by some algorithm what the ioname of the next module in the iopath is, this ioname is stored in "next_ioname." When more than one next ioname is involved (such as in the case of a broadcaster), the additional ionames are kept in auxiliary based storage to be described below. The number of additional ionames is kept in "nmore."

The current element size, measured in bits, is kept in "elsize." The various "pointers," the read, write, last and bound pointers, are kept as bit counts. The determination of these pointers as element counts is always accomplished by dividing the bit counts by the current element size.

"nbrk" and "ndelim" are the current numbers of break and read delimiter elements respectively. The actual strings of these elements are kept in auxiliary based storage to be described below.

The relative pointer "pibe" points to the first PIB extension; "more," "brk," and "delim" point to the auxiliary based storage for the additional ionames, the break element string, and the read delimiter string, respectively. The declarations for the auxiliary storage follow.

```
dc1 brklist (nbrk) bit (elsize) based (p), /*p related
      "                                             to relp.brk*/
  delmlist (ndelim) bit (elsize) based (q), /*q related
      "                                             to relp.delim*/
  more_ionames (nmore) char (32) based (r); /*r related
      "                                             to relp.more*/
```

This auxiliary storage is allocated within "ioarea" by the outer module whenever necessary. If any of this auxiliary storage is not needed, the corresponding relative pointer should be zero.

The various transaction block chain base indices are discussed later in this section.

All based storage allocated by the outer module is allocated within the PL/I area, "ioarea."

D. PIB EXTENSIONS

Per-ioname data not resident in the PIB are kept in what are known as PIB Extensions (PIBE's). There can be any number of PIBE's. Successive PIBE's are reached by relative pointers kept in immediately-preceding PIBE's. Each PIBE is required to contain an included measure of its size. The standard declaration for any PIBE follows:

```
dcl 1 pibe based (pn),          /*standard PIBE form*/
  2 relp,                     /*standard PIBE chaining*/
  3 next bit (18),            /*relp to next PIBE*/
  3 last bit (18),           /*relp to last item in this structure*/
  ...                         /*outer module's private dcls*/
  ;
```

The "next" is a relative pointer to the next PIBE in the chain. "last" is a relative pointer to the last item in the PIBE structure, and is included to permit size determination by outside procedures. The relative pointer to the first PIBE is (relp.pibe) in the PIB. The last PIBE must have (relp.next) zero. An outer module can avoid chasing a PIBE chain by keeping a copy of all PIBE relative pointers in the first PIBE.

Examples of PIB and PIBE Use

The real pointers corresponding to the relative pointers kept for accessing IS based storage must be computed each time a new call is made to the outer module. The real pointers therefore exist only in automatic storage. For example, to obtain the pointer p1 for use in accessing the first PIBE, one of the pointer manipulation procedures of BY.14 is used:

```
p1 = ptr$ptr(pibp,pibp->pib.relp.pibe);
```

The code to freshly allocate a second PIBE is:

```
allocate pibe in (pibp->pib.ioarea) set (p2);
p1->pibe.relp.next = ptr$rel(p2);
p2->pibe.relp.last = ptr$rel(addr(p2->pibe.last_item));
p2->pibe.relp.next = "0"b;
```

The code to allocate a new list of break elements following a setdelim call is:

```
brkp = ptr$ptr(pibp,pibp->pib.relp.brk);
free brkp->brklist;
allocate brklist in (pibp->pib.ioarea) set (brkp);
pibp->pib.relp.brk = ptr$rel (brkp);
```

E. THE INTERPROCESS COMMUNICATION BLOCK

The Interprocess Communication Block (ICB) is a common data base for the attachment module, the DSM's request queuer, and the device manager dispatcher. The ICB is allocated by the attachment module.

F. INTRODUCTION TO TRANSACTION BLOCK DISCIPLINE

The creation of Transaction Blocks (TB's) for outer calls (and for certain buffering functions and for non-switched iopath-directed calls) together with the ability to associate related blocks, is the basic mechanism for maintaining the necessary history of individual transactions and for preserving the relationships between these transactions.

A transaction block is a short fixed-length structure containing mostly information relating the block to other related blocks. The data in the block of direct interest to an outer module is a status bit string, some flags, and a relative pointer to a Transaction Block Extension (TBE). One such block is allocated by the I/O switch (by a call to the Transaction Block Maintainer) every time an outer call is made (except for the divert call); this block is automatically chained into the chain of blocks corresponding to calls issued to the same ioname. The base of this chain is anchored in the callee's PIB; specifically, "t1index" and "t2index" are actual indices into a transaction block array and point to the oldest and newest ends of the chain, respectively. Further, this chain essentially belongs to the callee; the only item in a block of interest to the caller is the status bit string.

Additional per-transaction data is kept in TBE's allocated by the outer module. The standard portion of the declaration of a TBE is exactly the same as that

described earlier for a PIBE. Any number of TBE's may be chained; the relative pointer to the first is kept in the parent TB.

Provision is made in the PIB for basing two additional TB chains. Modules retaining user data between calls must use a standard buffer chain. The module allocates (by a call to the Transaction Block Maintainer) a TB for every buffer needed; the actual data is kept in a corresponding TBE. Buffering is discussed in more detail later in this section. A third chain, called the auxiliary chain, is used by modules making outgoing calls but having no outgoing switch node. The auxiliary chain is used by DCM's to maintain the necessary per-transaction data for calls to the GIOC Interface Module (GIM), and is used by the DSM's Request Queuer to communicate with the Device Manager's Driver.

In addition to the chaining of the three "main" TB chains mentioned above, the Transaction Block Maintainer (TBM) will, upon request, create secondary chains of related blocks. The secondary chains are known as "down" chains and consist of existing blocks in main chains. The base of a down chain is anchored in some TB. A block can be included in the down chain based in some other block by a call to the TBM. Typical usage is to include blocks for outgoing calls in the down chains of blocks for corresponding incoming calls in such a way that status may be easily updated. An example of such use is given later. When buffer chains are in use, the incoming TB's' down chains include the corresponding buffer blocks, and each buffer block's down chain includes the corresponding outgoing TB's.

All the transaction blocks for a user are kept in a Transaction Block Segment (TBS). All use of these blocks by outer modules is by calls to the Transaction Block Maintainer (TBM).

The lifetime of a transaction block is controllable. A mechanism involving several hold bits representing various interests is used to delay the otherwise automatic deallocation of blocks. The holding and releasing of blocks is discussed later.

The Transaction Block Maintainer (TBM) services the allocation and deallocation of Transaction Blocks (TB's), the chaining of related blocks, the chasing of these chains, the storing and retrieving of the transaction status, outer module flags, and the TBE relative pointer. It also services the setting and resetting of the hold bits which control deallocation.

G. THE TRANSACTION BLOCK SEGMENT

The Transaction Block Segment (TBS) contains all the transaction blocks for a user process-group and is the principal data base of the TBM. The declaration for the TBS follows:

```
dcl 1 tbs based (p),
  2 lock_list,
  3 n fixed bin,
  3 chan_keys (N),
  4 proc bit (36),
  4 event bit (36),
  2 index,
  3 vacant1 bit (18),
  3 vacant2 bit (18),
  3 orphan1 bit (18),
  3 orphan2 bit (18),
  3 last bit (18),
  2 tb (MAX),
  3 status bit (144),
  3 hold bit (6),
  3 tbm_flags bit (6),
  3 om_valid bit (6),
  /*transaction block segment*/
  /*lock list for TBS*/
  /*size of key list*/
  /*channel key list, n=N*/
  /*process locking or queued*/
  /*queued process wake-up event*/
  /*indices*/
  /*head of vacant list*/
  /*tail of vacant list*/
  /*head of orphan list*/
  /*tail of orphan list*/
  /*highest block used*/
  /*transaction blocks*/
  /*outer call status*/
  /*hold bits*/
  /*flags for TBM*/
  /*outer module validation level*/
```

```

3 flags bit (18),          /*outer module flags*/
3 chain_cnt bit (18),     /*chain inclusion count*/
3 tberelp bit (18),       /*relp to TBE in per-ioname segment*/
3 xn1 bit (18),          /*main chain next index; from x1*/
3 xn2 bit (18),          /*main chain next index; from x2*/
3 dn1 bit (18),          /*down chain next index; from d1*/
3 dn2 bit (18),          /*down chain next index; from d2*/
3 d1 bit (18),           /*down1 index*/
3 d2 bit (18);           /*down2 index*/

```

The "lock_list" is used by the TBM to lock the TBS whenever necessary (by a call to the Locker); the TBM waits for the TBS to become free. The transaction blocks are members of a transaction block array; when the TBM is requested to "allocate" a block, it merely obtains a currently unused block from the vacant list. When a block is deallocated, it is returned to the vacant list. "vacant1" and "vacant2" are the indices to the head and tail respectively of the vacant list. "orphan1" and "orphan2" are the indices to the head and tail respectively of the "orphan" list. The orphan list contains blocks which have been removed from their original main chains but which do not yet meet the full conditions for deallocation; this mechanism is discussed later .

Each transaction block contains a standard status bit string, a string of hold bits, a string of flag bits private to the TBM, a string of flag bits usable by the outer module, the outer module's validation level, a chain inclusion count, a relative pointer to the transaction block extension, and indices of related transactions. Most of these items are discussed in detail later. The outer module's bit string (flags) is provided solely for the module's arbitrary use; for example, it may be used to conveniently differentiate between useful transaction categories. The TBM's bit string (tbm_flags) is used to indicate whether a block is in a main chain, in the orphan list, or in the vacant list.

The outer module validation level (`om_valid`) is used by the TBM to control what procedures can call to have certain items in the block set.

H. TRANSACTION BLOCK ALLOCATION

The following call to the TBM allocates a new block in a main chain:

```
call tbm$allocate(chain_base_ptr,holdn,tbindex,cstatus);  
    decl chain_base_ptr ptr, /*base of main chain*/  
          holdn fixed bin, /*hold bit indicator*/  
          tbindex bit (18), /*TB index of new block*/  
          cstatus bit (18); /*allocate call status*/
```

The chain base ptr is a pointer to a pair of chain base indices in the outer module's PIB. If the indices `x1` and `x2` point to the oldest and newest block respectively in the `x` chain, the TBM allocates the new block in the `x2` end of the chain. Here "`x`" corresponds to the "`t`", "`b`", or "`a`" in the chain-base-indices' names in the PIB. The index for the newly-allocated block is returned in tbindex and can also be found in the location for `x2`. holdn permits the caller to set the `n`th hold bit; if holdn is zero, the hold bits are initialized to zero. See the discussion on holding later below.

Transaction blocks for outer calls are allocated by the I/O switch. When the outer module receives control, it can find the index to the block corresponding to the current call in (`chain_base.t2index`) in the PIB. However, the status bit string is initialized by the I/O switch and bits 127-144 of the status string also contain the new TB index (bits 1-126 are initialized zero). A convention of using the value of the TB index extracted from the status string removes the necessity of coping with the effect of recursive entry upon the value of "`t2index`." (Although the `ioname` is locked upon first entry, a recursive entry by the same process is permitted.)

Transaction blocks for buffer or auxiliary chains are allocated by tbm\$allocate calls issued by the outer module itself.

When the TBM freshly allocates a block, it initializes a data base (om_valid) to zero. Upon receipt of the first subsequent call which requires storing information in the block, the TBM stores the current validation level in om_valid. Further such calls are fulfilled only if the caller's validation level is equal to or less than that in om_valid.

1. TRANSACTION BLOCK HOLDING

The "hold" bit string in a transaction block contains six hold bits called hold1,...,hold6. Setting any of these bits non-zero prevents deallocation of the block. At present, only hold1, hold2, and hold3 are assigned. With respect to an ioname, hold1 is for the caller and hold2 is for the callee. In those cases where a block is known to only one module (e.g., buffer blocks), hold2 is used by that module. Hold bit hold3 is used by the I/O switch to guarantee a caller a chance to set hold1.

The caller may wish to hold certain blocks in order to later examine an updated version of the status bit string. The status originally returned to a caller contains the then current status in bits 1-126 and contains a transaction block index in bits 127-144. During later calls, the callee updates the status of earlier calls when appropriate and if their blocks still exist. Further, it is the status string in the block which is updated, not the original caller's status string. The following calls are used by the caller to set and reset hold1:

```

call hold(status, cstatus)
call release(status, cstatus)

    dcl status bit (144),          /*returned outer call status*/
        cstatus bit (18);        /*hold/release call status*/

```

The TB index in the status argument is used by TBM to identify the correct block. A related TBM call is the following:

```
call getstatus(status, cstatus)
```

This call is used to replace an old status string by a new, possibly updated one. The caller provides status equal to the status bit string of a previous transaction which is being held; the TBM uses the TB index provided in status to identify the block and returns status equal to that currently in that block. The argument declarations are the same as the previous call. The following calls permit the callee (the outer module receiving the call) to set and reset the hold bits.

```

call tbm$set_hold(tbindex, holdn, cstatus);
call tbm$reset_hold(tbindex, holdn, cstatus);

    dcl tbindex bit (18),          /* TB index */
        holdn fixed bin,         /* hold bit indicator */
        cstatus bit (18);        /* call status */

```

holdn is an integer from 1 to 6 indicating which hold bit is to be set or reset in the block whose index is tbindex. If holdn is zero, neither call has any effect. It may be noted that when holdn is one, these calls duplicate the functions of the hold and release calls; the latter are designed to be safer and more convenient for the caller to use. If the holdn argument of a tbm\$allocate call is zero, the TBM initializes the entire hold bit string to zero.

A block whose hold bits are all zero and whose chain inclusion count is zero is a candidate for deallocation. When tbm\$allocate is called, other blocks in the

concerned chain are considered for deallocation. When a block is deallocated, its TBE's are freed and the blocks, if any, included in its down chain have their chain inclusion count (see later below) reduced by one.

Once an outer module is finished with a call block and has had hold2 set to zero, the block will continue to exist in the call chain until the full conditions for deallocation are realized. Inasmuch as an outer module frequently chases this call chain (see below), it would be convenient if only significant blocks were present. The following call is provided to remove

nondeallocatable blocks:

```
call tbm$remove(chain_base_ptr,tbindex,cstatus)
```

tbindex is the index of a block located in the main chain whose base indices are pointed to by chain base ptr. The TBM undertakes the following steps:

(1) hold2 is set to zero; (2) the block is deallocated if deallocation conditions are met; (3) if not, the TBE's are freed, the tberelp is set to zero, and the block is removed from the main chain and placed in the orphan chain. Blocks in the orphan chain can be expected to be eventually deallocated. Whenever an orphan block has any hold bits reset or its chain inclusion count reduced, the TBM considers deallocation.

When an outer module is detached, it is necessary to remove all the TB's in main chains based in the module's PIB. Upon receiving a return from a detach call, the I/O switch issues the following call on behalf of any such chain whose base indices are not zero.

```
call tbm$delete_chain(chain_base_ptr,cstatus)
```

The TBM performs the functions described for the tbm\$remove call for each block in the chain whose base indices are pointed to by chain base ptr.

J. THE CHAINING OF RELATED BLOCKS

Each transaction block contains the indices of the next oldest and next newest block in the same main chain; these are the xn1 and xn2 indices respectively in the block declaration. The oldest block has xn1 zero, and the newest block has xn2 zero. The use of bidirectional next indices permits the main chain to be chased in either direction beginning at any block. We speak, for example, of the xn1 next index as pointing away from the x1 end of the chain or toward the x2 end of the chain, where x1 and x2 are the chain base indices. The top row of blocks in Figure 7 shows the chaining of blocks in a main chain.

A second kind of chaining which is used to associate related blocks is available upon request. Suppose, for example, an incoming call to a module results in three outgoing calls. The incoming call has a transaction block in the module's call chain, and the outgoing calls have blocks in the next module's call chain. It is convenient for future status updating to save the relationship between these blocks. A mechanism is provided which permits threading the blocks for the outgoing calls into a "down" chain based in the block for the incoming call.

The following call threads an existing block into another block's down chain:

```
call tbm$thread(tbindex1,tbindex2,cstatus);  
    dc1 tbindex1 bit (18), /* index of TB to be threaded */  
    tbindex2 bit (18), /* index of TB basing the down chain  
    cstatus bit (18); /* thread call status */
```

tindex1 is the index of the block to be threaded; the threading will result in its dn1 or dn2 being set nonzero. tindex2 is the index of the block in which the down chain is or is to be based; the threading will affect the values of its d2 and possibly its d1. Prior to the existence of a down chain based in a block, both the d1 and d2 down indices are zero; when the down chain exists, d1 is the index of the oldest block threaded and d2 is the index of the newest one. If there is only one block in the down chain, d1 = d2 in the base block, and dn1 = dn2 = 0 in the threaded block. The blocks in the down chain are threaded to each other using the dn1 and dn2 down-chain-next indices; except for blocks at either end of the down chain, dn1 is the index of the next oldest block threaded and dn2 is the index of the next newest block threaded. Except for the blocks at either end of the down chain, a block can be in only one down chain. The block at the d1 end of the chain has dn2 zero if it is in only that chain; if it is also at the d2 end of another down chain, dn2 applies to that chain. A similar situation is true with respect to dn1 for a block at the d2 end of a **down chain**. A block which is at the d2 end of one down chain and at the d1 end of a second down chain can be in additional down chains provided it is the only block in those chains. No confusion exists in the interpretation of dn1 and dn2, since the down chains are chased by comparing successive dn1's with d2 (or dn2's with d1). The chain inclusion count (chain_cnt in the TB) is increased by one every time a block is threaded into another down chain.

The second row of blocks in Figure 7 shows a main chain whose blocks are included in down chains based in blocks in the top row. Blocks B1 and B2 are included in the down chain based in block A1; similarly, B2, B3, and B4 are included in the down chain based in block A3. Block B2 is in three down chains, those based in A1, A2, and A3.

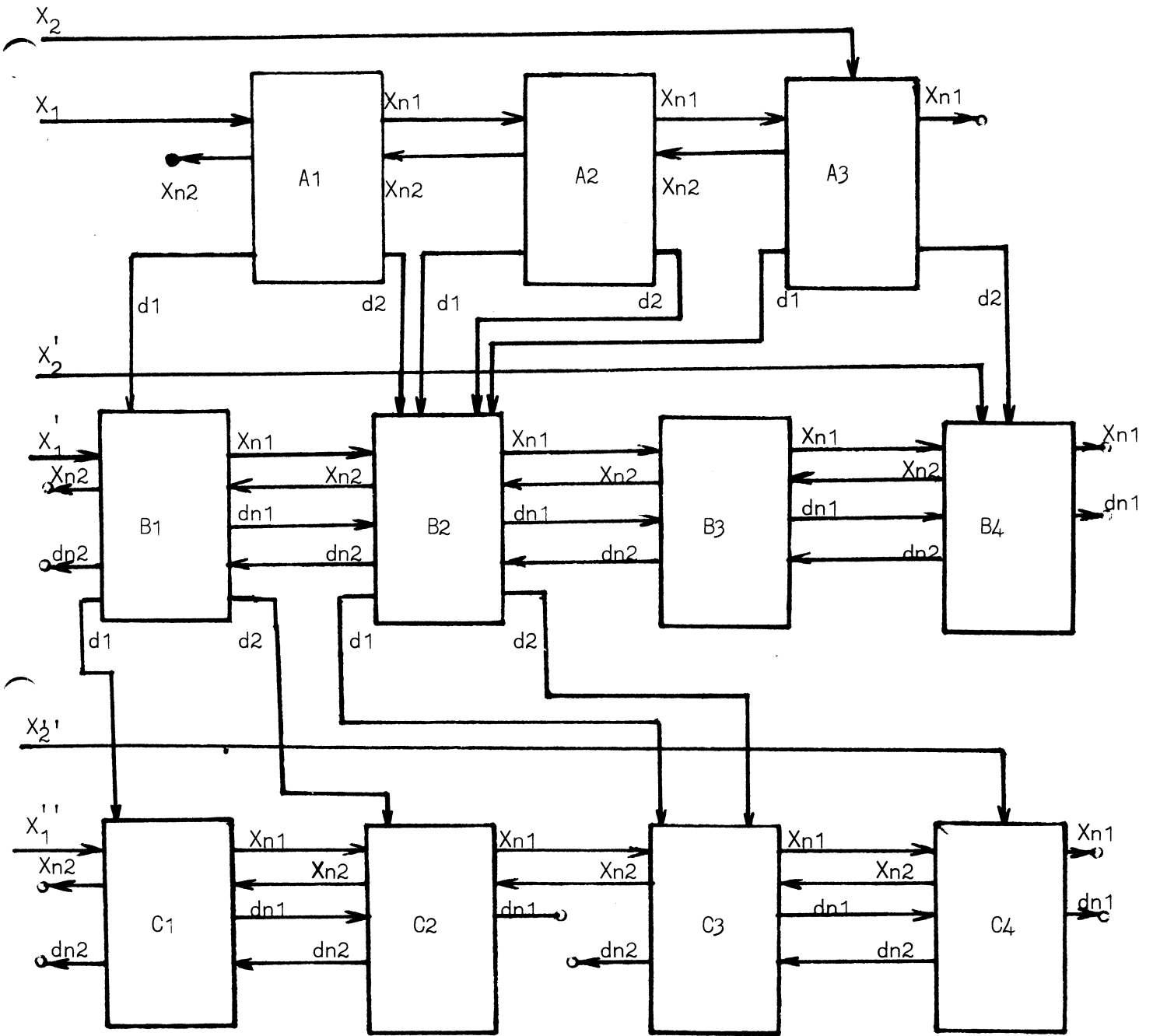



Figure 7. Example of Transaction Block Chaining

NOTES:  represents 0 zero index.

An outer module does not need to be concerned with the effort of chasing main or down chains. The following call is provided for chasing both main and down chains:

```
call tbm$get_chain(tbindx, type, orig, cnt, listptr, cstatus);

dcl tbindx bit (18), /*see below*/
    type fixed bin, /*1=down1, 2=down2, 3=main1, 4=main2*/
    orig fixed bin, /*offset, see below*/
    cnt fixed bin, /*size of return list*/
    listptr ptr, /*ptr to list*/
    1 list (cnt), /*return list*/
    2 tbindx1 bit (18), /*TB index*/
    2 flags bit (18), /*outer module flags*/
    2 status bit (144), /* transaction status*/
    2 tberelp bit (18), /*TBE relp*/
    : (18); /*get_chain call status*/
```

For type = 1 or 2, tbindx is the TB index to the base block containing the down chain to be chased. The down chain is chased from either the d1 end or the d2 end, according to whether type is 1 or 2. For type = 3 or 4, tbindx is an index of a block in a main chain which is to be chased from that point. The main chain is chased from either the x1 end or the x2 end, according to whether type is 3 or 4. orig is the offset from the basic block in numbers of blocks. cnt is the number of blocks whose tbindx (tbindx1), flags, status bit string (status), and TBE relp (tberelp) are wanted. A bit in cstatus indicates whether or not there are additional blocks in the chain. If orig = 1, the first block whose data is returned is the first down or next main block; if orig = N (greater than 1), the first data reported is from the Nth down or (next+N-1)th main block; if orig = 0, the first data reported is from the base block itself.

The get_chain call is the only call provided for fetching items in a transaction block. If data for only one block is wanted, an orig = 0 and a cnt = 1 are used.

K. OUTER MODULE CHAINING RESPONSIBILITIES

For every outer call received by an outer module for which the returned status indicates incomplete status reporting (status bit 5 equal to zero), the module must arrange for adequate holding and down-chain-inclusion of all other related blocks required for future status updating.

L. CALLS TO SET TRANSACTION BLOCK ITEMS

The following calls permit an outer module to set the flags, the TBE relative pointer, and the transaction status respectively in its transaction blocks:

```
call tbm$set_flags(tbindex, flags, cstatus);
call tbm$set_tbe(tbindex, tbeptr, cstatus);
call tbm$set_status(tbindex, status, cstatus);

dcl tbindex bit (18), /*TB index*/
    flags bit (18), /*outer module flags*/
    tbeptr ptr, /*TBE pointer*/
    status bit (144), /*transaction status*/
    cstatus bit (18); /*call status*/
```

flags is a bit string to be kept in the block for any use an outer module may desire. tbeptr is a pointer to the transaction block extension in the per-ioname segment; the TBM stores the corresponding relative pointer. status is the transaction status bit string. When an outer module returns to the I/O switch, the switch always calls the TBM to store the status parameter in the corresponding block. Thus, the outer module itself need not do so; it uses the set-status call only for updating old status strings.

M. TRANSACTION BLOCK SEGMENT SWITCHING

The following call is used by the Device Manager Process Driver and Dispatcher to switch the TBS:

```
call tbn$tbs(tbsp,event,cstatus);  
  
    decl tbsp ptr,      /*pointer to special TBS*/  
        event bit (36), /*event name for locker*/  
        cstatus bit (18); /*call status*/
```

The TBM normally uses the TBS created for the process-group in which it is called. The tbn\$tbs call causes the TBM to use the segment pointed to by tbsp as a special TBS. If tbsp is null, the TBM will revert to using the group's regular TBS. The event is an event name to be used by the TBM in its calls to the Locker. When operating using the special TBS, TBM calls to the Locker provide event as the event to be signalled when the TBS is free; the TBM does not wait for the signal but returns to its caller and indicates in cstatus that the TBS was not available.

N. BUFFER DISCIPLINE

Outer modules which need to hold user data between incoming calls must conform to a buffering discipline. It should be recalled that only DSM's buffer data to implement read-ahead and write-behind and that other modules are entitled to keep only unavoidably read-ahead data. However, certain modules may need to keep copies of processed output data as a precaution against an error occurring prior to physical completion of the output. For example, a code conversion module should keep processed output data until physical completion is indicated.

A data buffer takes the form of a transaction block extension of a block allocated into the buffer chain based in the PIB. The outer module allocates a new block by

calling tbn\$allocate with holdn = 2 to hold the block. When the buffer is no longer needed, the block is released by calling tbn\$reset hold with holdn = 2. The block along with the buffer (TBE's) will eventually be deallocated automatically. The declaration of the buffer TBE has the form of a standard TBE (and PIBE).

The standard buffer discipline includes the following standard down chaining. All buffer blocks whose TBE's hold data for a given incoming call are included in the down chain of the transaction block for that call. The data transmission involved in an outgoing call (to the next module) can be concerned with only a part or all of one buffer. All the blocks for outgoing calls corresponding to a buffer are included in the down chain of that buffer block. This is shown in Figure 7, if the top, middle, and bottom rows are considered to be incoming call blocks, buffer blocks, and outgoing call blocks respectively. For example, the TBE attached to block B2 contains data provided by incoming calls corresponding to blocks A1, A2, and A3; this data was passed on with outgoing calls corresponding to blocks C3 and C4.

This chaining enables straightforward status updating by starting with the module's call chain and chasing down to the buffer chain and then down to the outgoing call blocks. DSM's engaging in read-ahead create buffer blocks prior to the corresponding read calls; under these circumstances the down chaining from the call chain is not used.

0. EXAMPLES OF TBM USE

The examples included herein are intentionally concise to focus on the steps being

demonstrated. In particular, irrelevant but usually necessary intervening code is simply omitted.

The following is an example of outer module code involved in receiving an outer call, relaying it, holding, chaining the two call blocks, and returning; minimum status handling is shown:

```
tbin = substr(in_status,127,18);
call write(pibp->pib.next_ioname,...,status1,pibp);
call hold(status1,cstatus);
tbout = substr(status1,127,18);
call tbm$thread(tbout,tbin,cstatus);
substr(in_status,1,126) = substr(status1,1,126);
return;
```

The following example shows the allocation of a buffer block, the threading of the buffer block into the call block's down chain, and the allocation of the TBE (buffer). Writing out the buffer and threading the outgoing call block into the buffer block's down chain would be similar to the previous example.

```
tbin = substr(in_status,127,18);
bcbp = addr(pibp->pib.chain_base.blindex);
call tbm$allocate(bcbp,2,tbx,cstatus);
call tbm$thread(tbx,tbin,cstatus);
/*compute any variable lengths for bufthe*/
allocate buftbe in (pibp->pib.ioarea) set (tbep);
call tbm$set_tbe(tbx,tbep,cstatus);
/*copy user's data into buffer*/
```

The next example shows a general method of updating status. The following is a declaration for two structure arrays to be used when calling tbm\$get_chain:

```
dcl 1 mlist (N),
    2 (tbx,flags) bit (18),
    2 status bit (144),
    2 tberelp bit (18),
    1 list (M),
    2 (tbx,flags) bit (18),
    2 status bit (144),
    2 tberelp bit (18);
```

The first N blocks in the call chain are chased from the t1 end by issuing the following call:

```
call tbm$get_chain(pibp->pib.chain_base.t1index,3,0,N,  
                  addr(mlist),cstatus1);
```

If there are less than N blocks in the chain, the redundant mlist(i) are set to zero; cstatus indicates if there are more than N blocks. To chase the down chain in the i-th block from the d1 end, the following call is issued:

```
call tbm$get_chain(mlist(i).tbx,1,1,M,addr(list),cstatus2);
```

The list(j).status are examined to update mlist(i).status. Such updating for arbitrary numbers of blocks in these chains is accomplished by using program loops to repeat the calls when more than N and/or M blocks are present respectively.

P. DSM INTERFACE

Typically, an iopath includes a DSM which calls a DCM which calls the GIM.

Synchronization between the user process and the devices (e.g., read-ahead and write-behind) is implemented by the DSM. On the other hand, the DCM performs device oriented and device dependent functions. The functions which must straddle the boundary between the DSM and the DCM are the queueing of calls to the DCM and the forwarding of calls to the DCM. All DSM functions, except the queueing and forwarding of calls, are incorporated in the DSM which is a part of the user's working process-group. Calls to the DCM are queued in a data base known as the per-ioname-segment (IS). This segment is a per-ioname data base which contains pertinent information about I/O transactions on that ioname to the transmission of information across the process boundary between the user working process and the UDMPG; calls to the DCM from the DSM are queued and updated by another I/O system module, the request queuer. The DSM's per-ioname segment is the common data base between the user's working process and the universal device manager process group. All calls between the DSM

(in the user ring) and the DCM (in ring 1) can only involve data in the IS.

The information, and space used by the DSM to record the progress of the request, is placed in a transaction list in the IS. At this point, the list contains the following information:

Per Transaction:

Status of transaction

Requests issued to carry out this transaction

Associated logical buffers (if any)

Per Device (a transaction may involve more than one device):

Status of device

Physical information

Read pointer

Write pointer

Element size

Next I/O name (if any)

When a transaction is completed, final status is reported. The user process originating the request may ask to be informed of the completion.

As indicated, the DSM is concerned with read-ahead and write-behind buffering, which results in two DSM functions. First is the transferring of data between user workspace and the I/O buffers. Second is keeping the read buffer full and the write buffer empty. The second function involves adding information to the transaction list when transactions are complete.

The DSM uses information stored in the transaction list to determine if the indicated task is a result of a specific request or the result of a read-ahead or a write-behind strategy. If the former, the DSM updates the status of the specific request and, if necessary, may initiate or update outstanding transactions. This allows a large request to be broken into smaller tasks for the DCM. As transactions are finished, more are initiated (either logically or physically) until the original request is satisfied.

If the indication is of a task resulting from a read-ahead or a write-behind strategy, the DSM updates the pointers to the buffers (which are used in the PIB (per-ioname base) portion of the IS), and if buffer space permits, places additional information in the transaction list. Filling and emptying of buffers is not dependent on user calls to the I/O system.

Finally, the forwarding of calls from the DCM (via the request queuer and the IS) is performed by the dispatcher and the driver, which are I/O modules in the UDMPG. The more elaborate passing of information between the DSM and the DCM is necessitated because the information must be passed across process boundaries and across rings; therefore, interprocess communication and ring crossing and protection must be taken into account. The overall effect of the DCM/DSM interface procedures is the passing of requests for I/O to the DCM from the DSM.

X. PROCESSING OF DATA WITHIN THE I/O SYSTEM

Besides supporting a variety of devices, the I/O system offers a number of logical services. These services are implemented by logical service modules, which are outer modules of the I/O system. An iopath may include calls to one or several of these logical service modules. Many of these logical services involve the representation and handling of data.

A. DATA REPRESENTATION

The I/O system logical service modules allow the user to specify his mode of data representation which may be:

1. Linear
2. Sectional
3. Physical

1. Linear Representation

The basic representation for most users is linear. Every device and medium supported by Multics I/O can be used for linear data. In the linear data mode, the input from a medium such as tape or cards is buffered by the I/O system to appear as a steady stream of characters, words, or bits. For example, data read from cards can be read 80 characters at a time, one character at a time, 119 characters at a time or at any set buffer length regardless of the physical medium's limitation of 80 characters.

2. Sectional Representation

Sectional representation of data superimposes logical record structure on linear representation. Logical records may be of arbitrary length and may not be correlated with the physical structure of the external medium. For example,

logical bits ranging from one bit to ten or more can be punched into cards and, when read back as sectional data, can have the physical record suppressed and logical divisions retained. Logical data may be further divided into hierarchical groupings. The I/O system provides for treating a single record as separable into several independent sets of **data**.

3. Physical Representation

Data in physical form reflects the physical characteristics of the recording medium. For example, card data in the physical mode occurs in records which are all the same length as on a card. The same data read from a typewriter console arrives as a steady stream without record boundaries. Because of limitations inherent in handling data in the physical mode, use of this mode should be avoided wherever possible.

B. LOGICAL DIVISIONS OF DATA

In the data structures (physical, linear, sectional), external data is logically divided into frames. Frames are further subdivided. I/O operations such as read and write manipulate data aggregates and media which are external to the read and write procedures. A read call must specify explicitly or implicitly where the data is to be read from; and, likewise, a write call must specify where the data is to be put. In Multics, read and write calls specify external data by a two-component address. The first component is a data string specifying a frame of data; this component may be referred to as the "framename." The second component is the "item number." These terms are further explained in the ensuing discussion.

1. Linear and Sectional Frames

A frame of data is either linear or sectional. A linear frame is a single sequence of bits. Its only characteristic is the fact that it is of a given definite length. A linear frame has no intrinsic structure; however, a structure may be superimposed upon it to allow for easier handling of data within the frame. This is done by dividing the frame into a convenient number of fixed-length elements. An element size (in bits) is declared. I/O calls to linear data specify how much data is to be read, written, etc., in terms of the number of elements and the declared element size. Note that linear frames are fixed in length and contain fixed-length elements.

A sectional frame is a collection of data units called records. Each record has a number, which is a positive integer. Within the same frame, no two records have the same number. A sectional frame may have a variable number of variable length records. Sectional frames are also referred to as "logical record frames." The word "item" is used to denote an element of a linear frame or a record of a sectional frame. In the two-component external data address used in an I/O call, the second component is the item number. This item number determines where within the frame the activity (such as reading or writing) should be done. The place to be read from or written to may be either relative to the beginning of the frame (random accessing) or relative to the current item number (sequential accessing).

Every item in a frame of data has an item number, which is a positive integer. From the time a frame is attached to the time a frame is detached, the frame has a current item number.

2. Random Frames

The logical division of data is the same for frames in random access; the method of access is different. Frames are either random, linear, or sectional and their structure is as pictured in Figure 8.

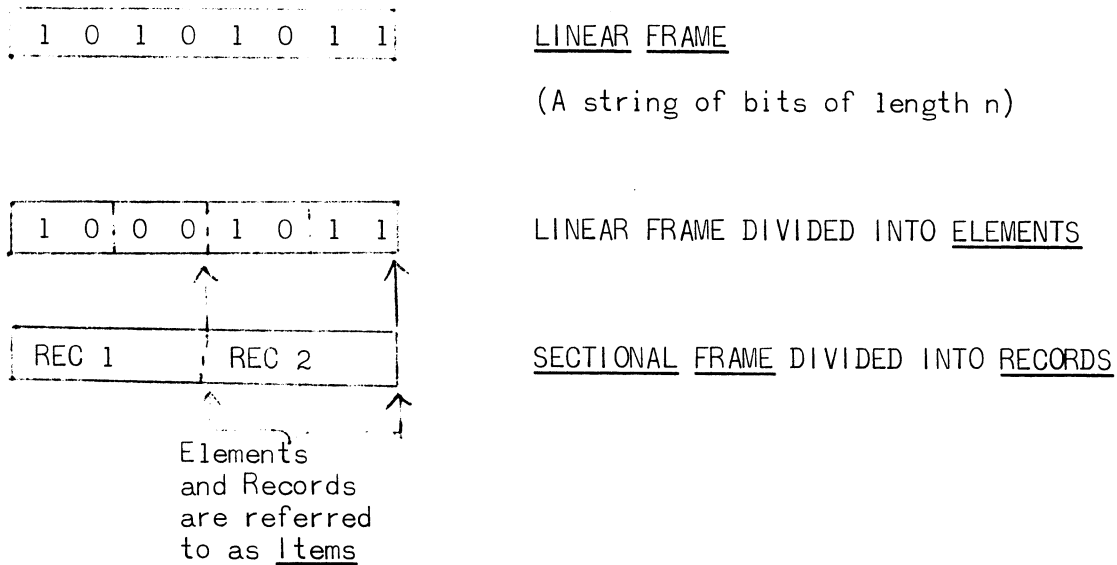


Figure 8 Pictorial Representation of Frames, Elements Records and Items

C. DATA DELIMITING

1. Establishing Delimiters

The user can set a specific element (or character) within a frame of data as the delimiter that defines the boundaries of the frame and the boundaries of items within the frame. It is also possible to define elements within a frame as ignore, erase, or kill indicators. These delimiters are set up by calls available to the user; in addition, the user may determine previous values of delimiters set for a frame associated with a particular ioname. The calls that perform these functions are setdelim and getdelim.

2. Finding Bounds of a Frame

Calls are also available to set a pointer to the bounds of a frame of data associ-

ated with a given ioname. This call is the set call. Conversely, a call, the tell call, enables the user to find pointers to the bounds of a frame when the pointers have been previously set by a set call.

D. DATA ACCESS

Random or sequential access is provided for both linear and sectional data. Random access is provided only to data on random access files or to data in file system files. Data on tape must be copied to a random access device before random access is feasible.

E. READING AND WRITING DATA

Through logical services, there are a number of ways in which data can be read or written. First the user can issue read/write calls to read a number of elements in a given frame or write a number of elements into a given frame which resides in a given work area. Further, the user can read and write in a synchronous or asynchronous manner. (Synchronous means control is not returned to the caller until reading, or writing, has been initiated or completed. Asynchronous denotes that the user wishes to perform read-ahead or write-behind processing.) The user can also issue calls to the I/O system to read and write records for devices concerned with physical records such as card readers, printers and punches. Housekeeping type calls enable the user to cause any unread or unwritten data in read-ahead/write-behind processing to be deleted or ignored. In addition, any read or write request can be cancelled. The calls that provide

these read/write facilities are:

<u>CALL</u>	<u>FUNCTION</u>
read/write	read/write the specified number of elements in a frame of data associated with the given ioname.
readsync / writesync	When asynchronous mode is specified, perform read-ahead or write-behind processing. With synchronous mode, reading or writing is initiated or completed before control is returned to the caller.
readrec / writerec	read and write records on unit record oriented devices or on magnetic tape.

figure 9 Summary of Read/Write Calls

Further modifications of methods of reading and writing of data is possible with respect to unique device characteristics. For instance, the I/O system can be informed about the location of tab stops, or tab stops can be set whenever this is a meaningful device attribute. Also the character of the printed output may be specified for those devices that perform printing. Printed output is specified in terms of:

1. total lines per page (effective page length)
2. total characters per line (effective page width)
3. number of lines of text per page (a function of text spacing and length as opposed to effective page length)
4. number of blank lines at the top of the page (origin of text)
5. number of characters to indent (left margin)

F. IOPATH MODIFICATIONS

The user can (1) discover information about the iopath created by the attachment of an ioname; (2) change the iopath, either temporarily or permanently, or (3) destroy the iopath. For example, it is possible to change the mode of an original attachment to provide a new diverted iopath or to replace one iopath with a new

iopath. It is also possible to suspend or resume current I/O along a particular iopath, or to restart I/O processing after a quit interrupt. By using a series of trace calls, it is possible for a user to find all the I/O module names to which the I/O switch would call after a call for a given ioname. The permutations of iopath tracing and modification lead one to conclude that there are at least as many ways of changing an iopath as there are positions in the Kama-Sutra.

A brief description of each call, its arguments and its functions, is contained in Figures 9 and 10. Generally, all calls are applicable to every outer I/O module, with exceptions due to peculiarities of each module. For example, the tabs and format calls are applicable only where printing is involved; similarly, calls to read or write are not meaningful where devices are write-only or read-only, respectively.

CALL	ARGUMENTS	FUNCTION
attach	(ioname1,type,ioname2,mode,status)	Associates ioname1 with a previously defined name or otherwise known device specified by ioname2.
detach	(ioname1,ioname2,disposal,status)	Removes from the given ioname(s) the association established by a former attach call. The disposal argument indicates how reserved resources (e.g., tape drives and tape reels) are to be treated
changemode	(ioname,mode,status)	The mode describes characteristics related to the attachment. The changemode call permits mode changes for the given ioname. The following is a partial list of the mode possibilities: readable; writable; appendable; random or sequential; if sequential, backspaceable or forward only; physical or logical; linear or sectional.

Figure 10. I/O Outer Calls

CALL	ARGUMENTS	FUNCTION
getmode	(ioname,bmode,status)	Returns a terse coding (bmode) of the mode of the attachment specified for the ioname.
noattach	(ioname1,type,ioname2,mode,status)	Used to prevent a subsequent attach call with the same ioname(s) and type from taking effect. This permits the specified subsequent call to be replaced by a different attach call.
readsync	(ioname,rsmode,limit,status)	For a given valid ioname which has been properly attached, the readsync call sets the synchronization mode (rsmode) of subsequent read calls. This mode is either synchronous or asynchronous. Synchrony means control is not returned to the caller until the read request is physically started or physically completed, depending on the workspace synchronization mode (see worksync below). Asynchrony means that read-ahead is possible to the extent permitted by the limit argument, which gives the desired maximum number of elements to be read ahead. The rsmode default is asynchronous.
writesync	(ioname,wsmode,limit,status)	Sets the write synchronization mode (wsmode) for a valid ioname. As in readsync, this mode is synchronous or asynchronous with default synchronous.
worksync	(ioname,wkmode,status)	For a given ioname, the worksync call sets the workspace synchronization mode. Synchrony implies that control is not returned to the user until the I/O system no longer requires the user's workspace (see read and write below). Asynchrony implies initiation of the call has taken place although the workspace is still in use. The default wkmode is synchronous.

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
read	(ioname, workspace, nelem, nelemt, status)	<p>The read call attempts to read into the specified workspace the number of elements (nelem) from the frame specified by the given ioname. The number of elements actually read is returned (nelemt). Reading begins with the current item of the frame. Thus for a linear frame reading begins with the element pointed to by a read pointer. Reading is then terminated by the occurrence of a read delimiter or by the reading of nelem elements, whichever comes first. The read pointer is moved to correspond to the next element, often the element last read. For a sectional frame (e.g., frame Y), reading begins with the first element of the subframe (e.g., frame X) pointed to by the read pointer for the current subframe (frame X). In this case, reading is terminated by the occurrence of the end of the subframe, by the occurrence of a read delimiter, or by the reading of nelem elements, whichever happens first. The current pointer for the frame (Y) and the read pointer for the subframe (X) are moved to correspond to the first element of the next subframe (X').</p>
write	(ioname, workspace, nelem, nelemt, status)	<p>The write call attempts to write from the specified workspace, the requested number of elements (nelem) onto the frame specified by the ioname. The number of elements actually written is returned (nelemt). The behavior of the write call with respect to the write pointer is similar to that described for the read call with respect to the read pointer, except that there is no write delimiter. Thus, writing begins with the current item of a frame and continues until nelem elements have been written.</p>

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
resetread	(ioname,status)	Deletes unused readahead data collected by the I/O system as a result of the read-ahead associated with the ioname.
resetwrite	(ioname,status)	Deletes unused write-behind data collected by the I/O system as a result of the write-behind associated with the given ioname.
iowait	(ioname,oldstatus,status)	For an ioname whose workspace synchronization mode is asynchronous, the iowait call defers the return of control as if the workspace synchronization were synchronous. This deferred synchronization applies to the most recent read or write call for a specified previous call. The oldstatus argument is the original status argument returned for the particular previous call and is used to identify the previous call uniquely. If oldstatus is missing, the most recent read or write call is implied.
abort	(ioname,oldstatus,status)	Cancels any physically incomplete read or write. The argument oldstatus has the same meaning as in the iowait call.
format	(ioname,ep1,epw,ts1,down,indent,status)	The format call is used to specify the characteristics of printed output formatting for the given ioname. The significance of the arguments in this call which refer to printing are as follows: ep1 - effective page length; epw - effective page width; ts1 - text space length; tsw - text space width; down - text space origin in lines; indent - number of characters to indent.
tabs	(ioname,tmode,hv,ntabs,tablist,status)	Used to inform the I/O system of the location of the tab stops on a device where this is a meaningful concept. The meaning of the arguments applicable to tabs is: tmode - indicates whether the call is providing tab locations, requesting that tabs be set, or requesting that the tab locations be returned to the caller; hv - indicates whether horizontal or vertical tabs involved; ntabs - indicates the number of tab stops; tablist - specifies each tab stop.

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
order	(ioname,request,argptr1,argptr2,status)	This call is used for communication among I/O system modules; it may also be used to set hardware device modules. The order call issues a request (request) to outer I/O modules. The third and fourth arguments point to additional request-dependent data; argptr1 points to a data structure containing forward-going arguments, while argptr2 points to a structure containing return arguments.
getsize	(ioname,elsize,status)	Returns the current element size (elsize) associated with read and write calls for the given ioname.
setsize	(ioname,elsize,status)	Sets the element size (elsize) for subsequent read and write calls for the given ioname.
setdelim	(ioname,nbreaks,breaklist,nreads,readlist,status)	Setdelim establishes elements which delimit data read by subsequent linear read calls with the given ioname. Argument breaklist points to a list of break characters (containing nbreaks elements). Each element in breaklist serves simultaneously as an interrupt canonicalization and erase/kill delimiter. Break characters are meaningful only on character-oriented devices. Argument readlist points to a list of read delimiters (containing nreads elements). The new delimiters established by this call are in effect until superseded by another setdelim call.

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION																																													
getdelim	(ioname,nbreaks,breaklist,nreads,readlist,status)	The getdelim call returns to the caller the delimiters established by the most recent setdelim call. The arguments have precisely the same meaning as those in the setdelim call.																																													
seek	(ioname,ptrname1,ptrname2,offset,status)	<p>sets the reference pointer specified by ptrname1 to the value of the pointer specified by ptrname2 added to the value of the signed offset (if the offset is present). Possible values of ptrname1 and ptrname2 are listed in the following table:</p> <table border="1" data-bbox="919 932 1451 1499"> <thead> <tr> <th data-bbox="919 932 1094 963">TYPE OF I/O</th> <th data-bbox="1122 932 1252 963">PTRNAME1</th> <th data-bbox="1284 932 1414 963">PTRNAME2</th> </tr> </thead> <tbody> <tr> <td data-bbox="919 982 1013 1014">linear</td> <td data-bbox="1154 982 1219 1014">read</td> <td data-bbox="1317 982 1382 1014">read</td> </tr> <tr> <td data-bbox="967 1014 1062 1045">frames</td> <td data-bbox="1138 1014 1235 1045">write</td> <td data-bbox="1300 1014 1398 1045">write</td> </tr> <tr> <td></td> <td data-bbox="1154 1045 1219 1077">last</td> <td data-bbox="1300 1045 1382 1077">first</td> </tr> <tr> <td></td> <td data-bbox="1138 1077 1235 1108">bound</td> <td data-bbox="1317 1077 1382 1108">last</td> </tr> <tr> <td></td> <td></td> <td data-bbox="1300 1108 1382 1140">bound</td> </tr> <tr> <td data-bbox="919 1171 1062 1203">sectional</td> <td data-bbox="1105 1171 1219 1203">current</td> <td data-bbox="1268 1171 1382 1203">current</td> </tr> <tr> <td data-bbox="967 1203 1062 1234">frames</td> <td data-bbox="1138 1203 1219 1234">last</td> <td data-bbox="1300 1203 1382 1234">first</td> </tr> <tr> <td></td> <td data-bbox="1138 1234 1219 1266">bound</td> <td data-bbox="1317 1234 1382 1266">last</td> </tr> <tr> <td></td> <td></td> <td data-bbox="1300 1266 1382 1297">bound</td> </tr> <tr> <td data-bbox="919 1329 1062 1360">physical</td> <td data-bbox="1089 1329 1235 1360">currentrec</td> <td data-bbox="1268 1329 1414 1360">currentrec</td> </tr> <tr> <td data-bbox="919 1360 1062 1392">I/O using</td> <td data-bbox="1154 1360 1219 1392">last</td> <td data-bbox="1317 1360 1382 1392">last</td> </tr> <tr> <td data-bbox="919 1392 1094 1423">readrec and</td> <td data-bbox="1122 1392 1219 1423">bound</td> <td data-bbox="1300 1392 1382 1423">first</td> </tr> <tr> <td data-bbox="919 1423 1045 1455">writerec</td> <td></td> <td data-bbox="1300 1423 1382 1455">bound</td> </tr> <tr> <td data-bbox="967 1455 1045 1486">calls</td> <td></td> <td></td> </tr> </tbody> </table> <p>The seek call is used to truncate or set the bound of a frame, and to set read and write pointers.</p>	TYPE OF I/O	PTRNAME1	PTRNAME2	linear	read	read	frames	write	write		last	first		bound	last			bound	sectional	current	current	frames	last	first		bound	last			bound	physical	currentrec	currentrec	I/O using	last	last	readrec and	bound	first	writerec		bound	calls		
TYPE OF I/O	PTRNAME1	PTRNAME2																																													
linear	read	read																																													
frames	write	write																																													
	last	first																																													
	bound	last																																													
		bound																																													
sectional	current	current																																													
frames	last	first																																													
	bound	last																																													
		bound																																													
physical	currentrec	currentrec																																													
I/O using	last	last																																													
readrec and	bound	first																																													
writerec		bound																																													
calls																																															

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
tell	(ioname, ptrname1, ptrname2, offset, status)	<p>returns the value of the pointer specified by ptrname1 as an offset with respect to ptrname2. The arguments ptrname1, ptrname2, offset have the same meaning as in the seek call. As an example, the tell call may be used to obtain the bound of a frame:</p> <p>tell(ioname, bound, first, offset).</p>
getstatus	(oldstatus, cstatus)	<p>used to replace the old outer call status, oldstatus, by a new (possibly updated) status using the same argument. The status for this call is cstatus, and has nothing to do with oldstatus. The getstatus call is not an outer call but an inner call to the transaction block maintainer. However, it may be invoked by the user to obtain the status of the transaction uniquely implied by oldstatus.</p>
upstate	(ioname, status)	<p>invokes each module in the iopath down to the DSM by means of another upstate call. On return back up the iopath, each module calls getstatus to update its status.</p>
hold	(oldstatus, cstatus)	<p>sets the hold1 bit in the holdbit string on. This is a bit string in the transaction block; setting any of these bits ON prevents de-allocation of the block. The status of the call is cstatus and is not related to oldstatus.</p>

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
release	(oldstatus,cstatus)	resets the hold1 bit set by the hold call to OFF.
readrec	(ioname,reccount,workspace,nelem,nelem, status)	This call is intended solely for devices concerned with physical records such as card readers, printers, and magnetic tapes. It is accepted by the device control module for tape and the unit record device control module. It is also accepted by DSM's calling these device control modules when the device attachment mode contains a P (physical). The argument reccount indicates the number of records that the readrec call represents. The call is similar to the read call, except that nelem and nelem are arrays of element counts, and workspace is an array of pointers to the corresponding workspace.
writerec	(ioname,reccount,workspace,nelem,nelem, status)	similar to the readrec call: writerec provides means of writing physically oriented records.
localattach	(ioname1,type,ioname2,mode, status)	identical to the attach call except that the scope of the resulting attachment is specific to the process issuing the attachment rather than global to the process group.
localnoattach	(ioname1,type,ioname2, status)	identical to noattach except that the scope of the subsequent attachment to be prevented is local to the process issuing the call.

Figure 10. I/O Outer Calls (continued)

CALL	ARGUMENTS	FUNCTION
divert	(ioname1,newioname,mode,status)	suspends any current I/O on the attached device specified by ioname1 and allows immediate initiation of new I/O on the ioname specified in newioname. If ioname1 and newioname are identical, ioname1 is renamed.
revert	(ioname1,mode,status)	reinstates the original attachment suspended by the previous divert call.
invert	(ioname,status)	destroys the original iopath for the given ioname and destroys subsequent diverted iopaths, except for the most recently diverted iopath. This call is chiefly used internally by the I/O system.
restart	(ioname,status)	restarts input/output for the given ioname after a quit. This call is used mainly by the overseer.
trace	(ioname,modname,nextlist,status)	provides the module name (given by the argument modname, e.g., twdsm) to which the I/O switch would switch as a result of a call to the given ioname. It also provides the potential ionames which could arise in the next call along the iopath (nextlist). By using sequences of trace calls, the iopath associated with an ioname may be obtained.

Figure 10. I/O Outer Calls (continued)

XI. FUNCTIONS OF OUTER I/O MODULES

A. DEVICE STRATEGY MODULES (DSM'S)

Many peripheral and/or terminal devices share common attributes and can be grouped into classes. Each device strategy module (DSM) in the I/O system is designed to function for a particular class of devices. For example, the 1050, 2741, and TTY-37 are similar in some ways; therefore, a device strategy module can be constructed for handling the features that these devices have in common. The following types of device functional characteristics may be collected into a device strategy module:

- 1) Queueing I/O requests to a DCM.
- 2) Read-ahead and write-behind buffering.
- 3) Conversion of physical records to logical records and vice versa.
- 4) Keeping pointers to the next record or word, or both, to be read or written.

A DSM exists for every device in the system and the DSM is part of the user's working process.

B. DEVICE CONTROL MODULES (DCM'S)

Interfacing with the outer I/O modules (DSM) and the hardcore I/O (GIM) are a number of modules called device control modules (DCM's), all of which are a part of the universal device manager process group and all of which are in ring 1. The control of a particular device is the function of the DCM, and there is one DCM for each type of device attached to the system. The DCM converts the device capabilities assumed by the other outer I/O modules into the actual capabilities of the device and creates the lists of DCW's that are passed to the GIM. Therefore, the systems programmer who writes the DCM for a particular device must know its capabilities and is responsible for assuring

that the DCM sets up the proper DCW's within any list that the DCM may issue. Since additions and changes are most likely to occur at the device control level, (e.g., the addition of a new feature on the I/O device) DCM's are kept as small and simple as possible. In general, a DCM is no more complicated than the functions of the controlled device require. The DCM (and also other outer I/O modules) are not concerned with the communication path between storage and the I/O device; this is a function of the GIM.

C. FILE SYSTEM INTERFACE MODULE (FSIM)

The file system interface module is incorporated into the I/O system to permit the file system to be treated as if it were an I/O device. The I/O system can read or write a file using the same logic as it does for reading or writing from a device. Therefore, along with the GIM, the FSIM is a possible termination point for an iopath. If data storage requires more than one file, the file system interface module of the I/O system designates a group of files to hold the data. The important consideration for the user is that I/O can handle a file as if it were a device, thus making the files independent of the device containing them; this concept also dovetails nicely with the file storage hierarchy scheme where little-used files are on slow devices and frequently-used files are on fast devices.

To use a file as a device, the following steps are taken:

- a. Request for I/O action within a file system file.
- b. I/O calls the file system for the file.
- c. If the file is in core or on the drum, the file system satisfies the request immediately and the file is available to the user to read and write through I/O.
- d. If the file is on another medium (disc, tape, etc.) the file system must, in turn, call I/O to make the file available.
- e. When I/O makes the file available, the file can be read or written (as in "c" above).

XII. HARDWARE I/O (GIM)

A. GIM FUNCTIONS

The I/O routines that are most directly concerned with the manipulation of the hardware constitute the GIOC Interface Module (GIM). The GIM is the sole interface between the GIOC and the I/O modules within the universal device manager process group that are concerned with the operation of devices (DCM's). The GIM is faced on one side with the hardware present in the GIOC and on the other side with Multics processes which want to use the GIOC hardware to control the amount and type of I/O activity. Figure 11 diagrams the operations within the GIM.

The GIM has direct control over the facilities of the GIOC because it is the GIM that interprets the lists of DCW's that define I/O activities; creates channel control information (such as the CIW, COW, and LPW); and places these created control words, along with the DCW lists, in the proper location in the GIOC mailbox area, or within the connect, list, or data channels of the GIOC. Thus the main function includes the proper placement of the DCW's and DCW lists and the creation of the primitive commands and pointers needed to activate and run the channels of the GIOC. The GIM also receives status words from the GIOC, under direction of the SCW's. The systems programmer can make calls to the GIM to determine status for the purposes of passing this information back to the UDMPG and the user's working process.

B. DCM/GIM INTERFACE

The interface between the GIM and the DCM is a list of DCW's which is passed to the GIM for each I/O transaction. The list interface to the GIM provides a minimum of bother to the GIM user with regard to GIOC idiosyncracies of absolute addresses, protection, appending, segmentation, etc. also, the list

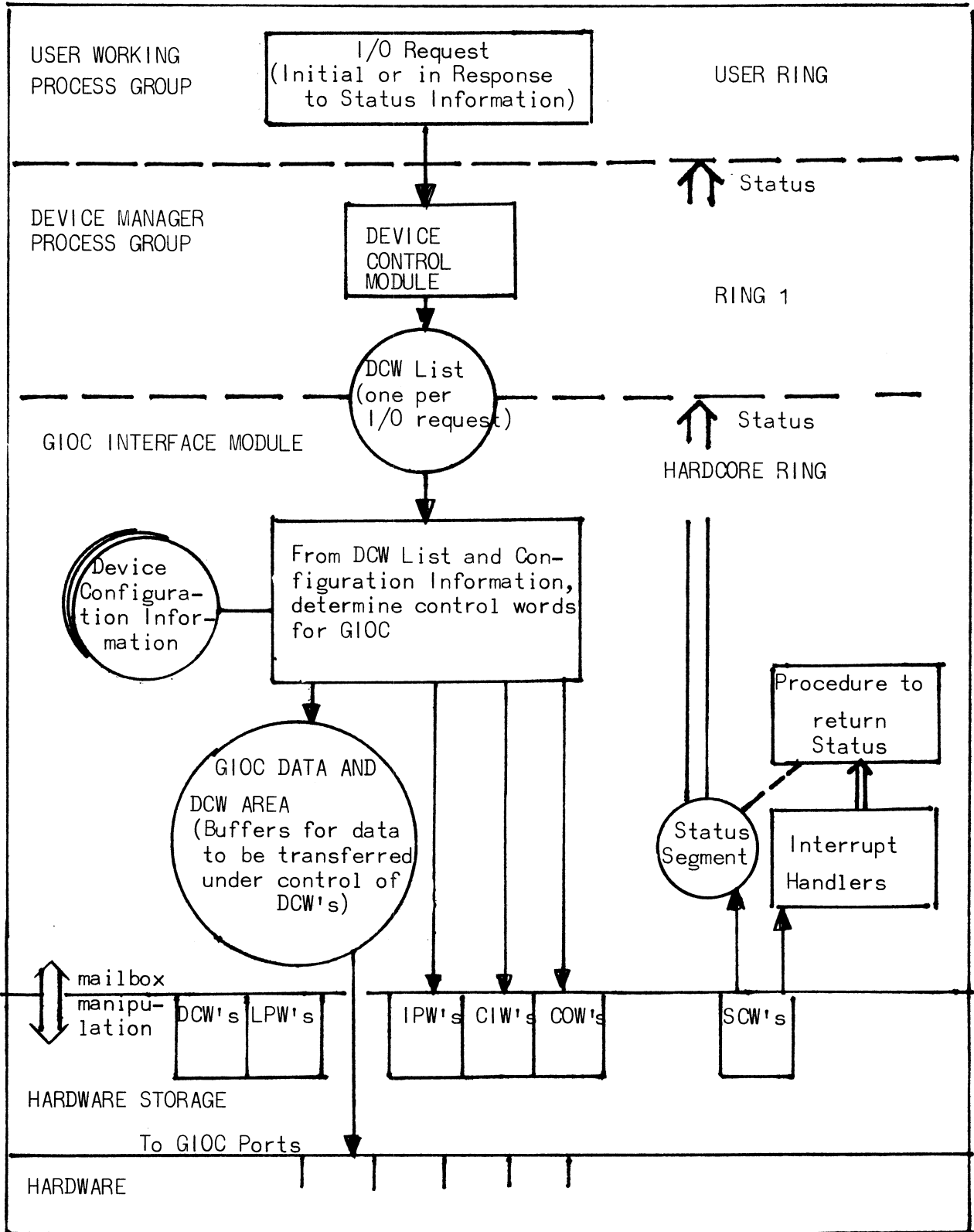


Figure 11. Outline of GIOC and Its Functions

interface allows flexibility in operating through the use of a list of DCW's that can cause the device to perform only the necessary functions that are needed to perform a particular I/O task, provided the necessary functions are within the capabilities of the device.

Once the GIM receives a list of DCW's and is informed by the user where to start in the list of DCW's, the GIM can create a list of LPW's for the list channels in the GIOC. Also, the GIM determines, from the source of the call for the I/O, the type of device requested; i.e., the DCM that called the GIM is unique to the type of device. The type of device determines which connect channel within the GIM to use since there are two fast channels and one slow one, and the slow channel is always used for TTY transmissions. After determining which connect channel to use, the GIM creates the appropriate pointer (IPW) to the appropriate connect instruction (CIW) and creates the COW (connect operand word). When the COW is executed, the channel connection is made and the pertinent list and data channels are activated to transmit data to and/or from storage and the device via the GIOC (under control of the DCW's in the DCW list which was passed from the DCM to the GIM). To activate each list of DCW's, a single CIW is executed by the connect channel. The IPW queuing facilities of the GIOC are not used and the connect channel is not allowed to store status; rather, the GIM observes the connect channel mailbox to see if the latest CIW has been executed. When it has been executed, a new IPW, pointed to by a new CIW, can be placed in the connect channel. The channel can then be reactivated. Because the connect channels are fast and because the user requests to begin I/O are not queued but are looped in memory by the GIM until they are serviced, the exhausting of an IPW in a connect channel does

not generate an interrupt; instead, the connect channel is almost instantly reactivated to service another list of DCW's, if one is waiting.

C. STATUS INFORMATION AND THE GIM

As stated previously, status words are passed back to the GIM through the GIOC status channels under control of the SCW's. These status words are queued within the GIM data area. If an interrupt is generated (such as by the receipt of an emergency or a termination status word), the GIM interrupt handler, a hardcore supervisor module, responds to restore previous conditions and pass interrupt status return information to the GIM which, when called, will pass status information back to the device manager process (DCM) for the device which caused the interrupt. (Subsequently, it is presumed that the status information will be passed back to the user's working process for appropriate response, which will probably be the generation of further calls to I/O for retries, checking, or new I/O, etc.) In addition, the user (i.e., the systems programmer) can query status through calls to the GIM. One call:

`get_cur_status`

gives the current position of the pointer (LPW) to the list of DCW's, which tells the user how far along the particular I/O request has progressed. This is the quickest and cheapest way of determining status. Another call for status:

`get_status`

causes the status list to be serviced in its entirety. While the user "pays something" to get this information, it may be desirable to keep the user's status information from being overwritten, or it may be necessary to determine the occurrence of some significant event, such as the completion of a read or write of a block of data.

D. HARDWARE I/O SYSTEM DATA BASES

Data bases and structures within the GIM are divided into two categories, system-wide data bases and per-device data bases. System-wide data bases are used throughout the GIM as well as by the hard-core I/O.

1. Static Storage

Many system parameters relevant to GIOCC operation throughout the GIM reside in a static storage segment, `hcio_stat_`. These values are set at I/O system initialization time and are never altered except for dynamic reconfiguration (e.g., add a GIOCC to the system) or system experimentation.

In addition, `hcio_stat_` contains all the equivalences between symbolic errors detected in the GIM (e.g., system or machine error) and the error code returned to the user.

2. Channel Assignment and Status Table (CATCST)

The channel assignment portion of the CATCST contains information relating each GIOCC and device to the Multics configuration. The CAT serves as a general index to most of the GIM data bases and devices and serves a fundamental role in the operation of the GIM.

All status is kept in hardware queues until called for or overwritten. The CST portion of the CATCST contains information about the hardware status queues and about some hardware interlocks.

3. GIOCC Mailbox Areas

The GIOCC mailbox areas are the heart of the GIOCC operation. It is within these areas that the GIM reserves complete license for manipulation. The implementation of the GIM requires one distinct mailbox segment for each GIOCC attached

in the Multics configuration. Initial implementation of the GIM has given the name "gioc_mbxN" to these segments. N covers the range 1, 2, ..., "ngiocs." Suitable declarations for the GIOC mailbox areas are shown below.

```

dcl 1 mailbox based(p),
    2 scw(0: 3),                /* the 4 status channels */
    3 scwa bit(36),            /* word a */
    3 scwb bit(36),            /* word b, the active one */
    2 x(4) bit(72),            /* 4 unused boxes */
    2 cpw(0: 2) bit(72),        /* connect channel pointer words */
    2 x1(3) bit(72),           /* 3 unused boxes */
    2 data(7: 2047),           /* data channels */
    3 lpw bit(72),             /* list pointer word */
    3 dcw bit(72),             /* DCW mailbox */

```

4. GIOC DCW Area

Since the GIOC does not have a counterpart to the 645 processor appending hardware, the DCW's necessary for driving the devices attached to a GIOC must reside in core in a manner reminiscent of absolute programs. That is, the DCW area must be a wired-down segment that is either unpagged or pagged contiguously. The GIM is responsible for allocating and freeing space within the area so that a user's DCW's may be placed in the area and released after they are no longer needed.

5. GIOC Data Area

In the same spirit as the GIOC DCW area, a wired-down segment is needed for a user's data buffer area. On all I/O instructions involving writing, the user's write buffer is first copied by the GIM into the GIOC data area. Subsequent I/O via the GIOC writes the data onto the appropriate device. Similarly, on calls involving reading, the GIOC initially places the data in the GIOC data area. Subsequent actions of the GIM copy the data into the user's buffer area. The GIOC Data and DCW areas are the same segment, `gim_abs_seg`.

6. Channel Copy Table

This table is a per-device table which gives information about DCW lists, especially data areas allocated and user's buffers into which he wants real data copied.

XIII. GIOC

A. GENERAL

The GIOC controls the data transfers between the 645 processor and the many types of devices attached to the operating system. The GIOC hardware consists of a controller and a combination of adapters to suit the I/O devices. All the components of the GIOC can operate simultaneously.

The various types of adapters in the GIOC are independent units which can be installed to fit the requirements of the peripherals and terminals. The adapters contain one or more data channels which provide an interface between the GIOC and the peripherals and/or terminals. The adapters also contain list channels which provide information to direct the operation of the data channels.

The GIOC controller is connected with hardware interface ports to storage.

The controller receives information from either storage or from a device adapter and responds to the information to direct the performance of the input/output operation. The controller contains three connect channels which can activate channels in any adapter. The controller provides access to storage for all the channels in the GIOC. In addition, the controller stores the status of the various I/O operations from the status channels in the GIOC status queues.

Figure 12 illustrates the organization of the GIOC, the channels associated internally in the GIOC, and the control words handled by these channels.

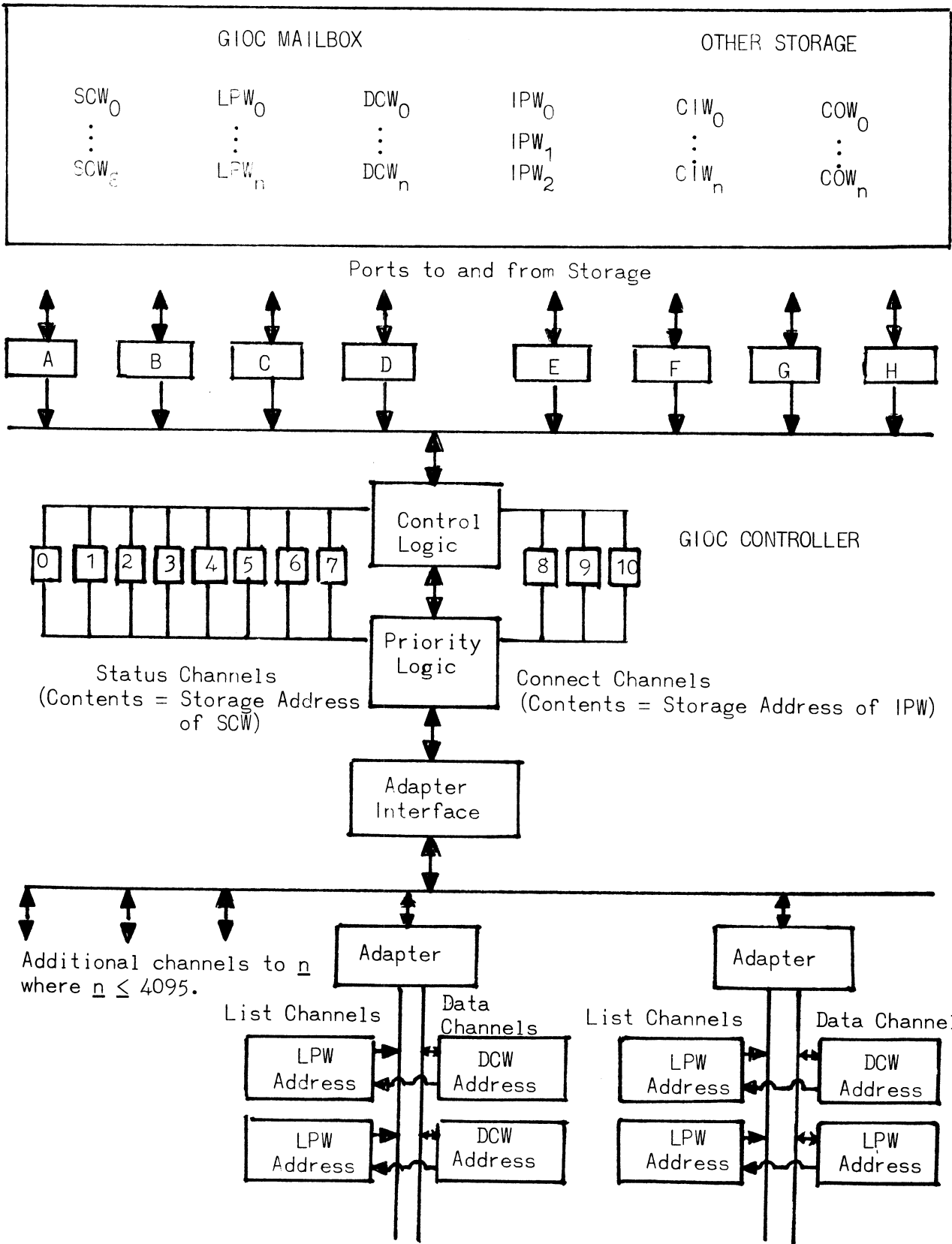


Figure 12. GIOC - Internal Organization

B. CHANNELS AND CONTROL WORDS

1. Types of Channels

A GIOC channel is a controlled path between a source and a destination through which information can flow. Four types of channels are used in the operation of the GIOC; these are: connect channels, status channels, data channels and list channels (see Figure 11).

Connect channels control the transfer of instructions (in the form of Channel Instruction Words) to data channels to start individual I/O operation. The operating system arranges the Channel Instruction Words in a list in storage. A connect instruction issued by the operating system (as a result of a read/write call) initiates the operation of a connect channel in the GIOC. A connect channel contains Channel Instruction Words which can activate any list channel.

Status channels control the transfer of status information from any data channel, list channel, or connect channel to storage. The status channels arrange the status words in sequential lists in storage. There are four standard status channels; these status channels are part of the GIOC.

Data channels control data transfers between storage and peripheral devices or terminals. There is one data channel for each peripheral subsystem or communication line connected to the GIOC. Each data channel provides the necessary buffering and hardware to control the attached device. Because of the diverse requirements for interfaces and for buffering, there are a number of different types of data channels. All data channels are half duplex; two data channels are required for full duplex operation. Each data channel is a part of a specific adapter within the GIOC. The number of data channels in the GIOC is determined by the overall configuration and requirements of the peripheral and communication lines.

List channels obtain data control words for subsequent control and operation of the data channels after they have been initialized. The operating system arranges the data control words for each I/O operation in a sequential list in storage. A list channel is associated correspondingly with each data channel. All list channels are adapter channels.

2. Control Words

Each channel has one or more control words associated with it; the control word directs the operations of a channel when information is being transferred between the GIOC and storage. This control word may reside in a "mailbox", which is a specific storage location. The mailbox, and consequently the control words, are accessed by both the GIOC and the processor. A specific type of control word is normally stored in the mailbox for each type of channel. These are:

<u>CHANNEL TYPE</u>	<u>CONTROL WORD TYPE</u>
Connect Channel	Instruction Pointer Word (IPW)*
List Channel	List Pointer Word (LPW)
Data Channel	Data Transfer Control Word (DCW)
Status Channel	Status Control Word (SCW)

Additional control words are used by the GIOC but are not stored in the mailboxes; these are:

Connect Operand Word (COW)
Channel Instruction Word (CIW)

3. Direct and Indirect Channels

In preceding text, GIOC channels have been classified and described according to type; the channels can also be classified as either indirect channels or

* The IPW is referred to as a Connect Channel Word (CCW) in other literature about I/O. This document uses the IPW terminology

direct channels in accordance with the place of residence of the control word associated with the channel. All connect channels, status channels, and list channels are indirect channels. All data channels are classified as peripheral (which can be direct or indirect) or as communication (which are indirect).

The control word for an indirect channel resides in a mailbox in storage. The channels are called indirect because each channel addresses storage indirectly through its mailbox. Indirect channels have sufficient capability to keep communication lines or peripheral equipment, such as card readers, card punches and printers, operating at full speed.

The control word for a direct channel resides in the channel. These channels are called direct channels because each channel addresses storage directly by means of the control word residing in the channel itself. Only data channels can be direct channels. Because of high transfer rate capabilities, direct channels are used for high-speed peripheral equipment, such as discs and magnetic tapes.

4. Control Word Functions

a. Connect Operand Word (COW)

The connect operand word is the word referenced by the operand portion of the connect instruction issued as the result of a read/write call in the operating system. The GIOC controller interprets the COW to determine, first, the value of the port number field (which must be placed in the COW previously by the operating system). This number determination causes the COW to be sent through the specified storage port to the GIOC. The GIOC also interprets the COW to determine:

1. Which connect channel is to be activated.

2. Whether the GIOC should be in test mode or in normal mode of operation (the mode is always normal except for hardware test and diagnostic programs).

b. Instruction Pointer Word (IPW)

The mailbox for each of the three connect channels contains an IPW. The IPW controls the transfer of channel instruction words to the adapter by providing a pointer to a list of these channel instruction words.

c. Channel Instruction Word (CIW)

The CIW is the word referenced by the IPW. Since the CIW is interpreted by the GIOC adapters, the format for the CIW varies according to whether it is to be interpreted by a peripheral adapter or interpreted by a communication adapter. A peripheral CIW specifies the operation code and device number; a communications CIW must indicate which data channel is to respond to the data transfer instruction and give coded details as to how the data channel is to respond.

d. List Pointer Word (LPW)

The mailbox for each list channel contains an LPW. The LPW points to a list of DCW's which are to be used by the associated data channel. The LPW also controls the transfer of these DCW's to the data channel (if it is a direct data channel) or to the mailbox of the data channel (if it is an indirect data channel).

e. Data Transfer Control Word (DCW)

The control word that directs data transfer across the GIOC to and from storage and the devices is called the data transfer control word (DCW). Information specified by the DCW includes the location and size of the data area. The number of DCW's used for an I/O operation may be 1, 2, ... n (where n = up to 4096).

The DCW's for an I/O operation are in contiguous storage locations, thus forming a list. The DCW being used resides in the data channel mailbox. The LPW specifies the location of the next DCW and how many more DCW's are to be processed. The I/O operation terminates when the LPW tally runs out, unless something happens to cause an earlier termination.

The mailbox for each indirect data channel contains a DCW when that data channel is involved in an I/O operation. Direct data channels hold their own DCW's and use the mailboxes only for the LPW's.

Some communication line remote terminals have special needs for information on the line, especially at connection time. Appropriate DCW's exist for these special conditions, for example, a literal DCW for transmitting a specific bit configuration repetitively. In this case, the DCW does not specify a storage location but a bit configuration to be transmitted and the number of times to transmit it.

The various types of DCW's for both direct and indirect channels are listed in the charts in Figure 13 which summarizes all the control words used by the GIOC.

Control Word	Function	Location	Used By	Remarks
COW	Gives value of: 1. Port between storage and GIOC which is to be used. 2. Which connect channel to activate 3. Whether GIOC is to be in test or normal mode.	Placed in when the connect instruction is transmitted to the GIOC.	GIOC Controller.	Referenced by operand portion of 645 connect instruction.
IPW	Points to CIW.	Mailbox for a connect channel.	Connect channel.	
CIW (peripheral)	Contains: Peripheral operation code. Device number.	Current one is in mailbox for a connect channel. CIW's are kept in a list.	GIOC peripheral adapters.	Referenced by IPW.
CIW (communications)	Indicates which data channel is to respond; gives coded details as to how data channel is to respond.	Mailbox for a connect channel.	GIOC communications adapters.	Also referenced by IPW.
LPW	Points to a list of DCW's; controls transfer of DCW's by pointing to next DCW by counting the number of DCW's remaining in the current list.	Mailbox for a data channel.	GIOC data channels (both direct and indirect.)	
Normal DCW	Controls normal data transfer and sets certain special modes of operation.	Mailbox for indirect data channels.	Indirect data channels.	DCW Type 000.
Control Character DCW	Controls data transfer when it is desired to detect the receipt of a specific character or class of characters.	Mailbox for indirect data channels.	Indirect data channels.	DCW Type 001.
Tally Match DCW	Controls data transfer when it is desired to provide special indication that a specific number of characters have been transferred.	Mailbox for indirect data channels.	Indirect data channels.	DCW Type 010.
Transfer DCW	Used to transfer control from one DCW to another DCW.	Mailbox for indirect channels.	Indirect data channels.	DCW Type 011.

Figure 13. Summary of Control Words Used by GIOC

Control Word	Function	Location	Used by	Remarks
Instruction DCW	Used to issue instructions to data channels at pre-planned points in the data transfer sequence.	Mailbox for indirect channels.	Indirect data channels.	DCW Type 100.
Literal DCW	Used to define a specific character which is to be transmitted a specified number of times (including start and stop bits where applicable.)	Mailbox for indirect channels.	Indirect data channels.	DCW Type 101.
DCW (for direct channels)	Directs normal transfer of data, gives starting and termination information on data locations, etc. The DCW for direct channels varies in information and format depending on the nature of the attached device.	In the direct data channel.	Direct data channels.	Direct DCW lists are pointed to by an LPW word in the data channel mailbox.
SCW	Controls the transfer of status information (in the form of status words) back to storage where the system can make use of this information and/or take required I/O action.	Status channel mailbox area.	Status channel.	A corresponding type of status word is normally generated each time a status event (change) occurs; these are: <ul style="list-style-type: none"> - Terminate status for peripheral data channels. - Terminate status for communications data channels. - Exhaust status for peripheral data channels. - Exhaust status for communications data channels. - Exhaust status for connect channels. - External signal. - Internal signal. - Emergency.

Figure 13. Summary of Control Words Used by the GIOC (continued)

C. I/O HARDWARE/SOFTWARE INTERFACE

The hardware in the GIOC interacts closely with the software of the operating system. The features of I/O data transfer (such as indicating where the data is located, how much data is to be transferred, what devices are to be used, and how the data is to be transferred) are built into the GIOC hardware so that the hardware can act, under the direction of control words, to alleviate the user to the bother of specifying the numerous details of I/O operation to the system. The GIOC handles a vast variety of device (in fact, every peripheral or communication device in the Multics configuration with the exception of the drum). Thus, the GIOC presents a degree of symmetry in the interfaces between the various hardware devices and the software processes of the Multics system.

The general scheme of performing an input/output operation is as follows:

- 1) Control words (DCW's) are set up in memory.
- 2) The operating system issues a connect instruction, as a result of an I/O call.
- 3) The GIOC fetches the control information from storage (i.e., the mailbox area and the IPW) to ascertain what activity it should direct.
- 4) The data transfer takes place. Individual control words (DCW's) are used to control each transaction.
- 5) When the data transfer is complete, the GIOC obtains or builds status information from the channel and stores the status (using SCW's).
- 6) The memory module sends a signal to the processor, setting an interrupt cell so that the I/O device may be serviced.