M0122

A GUIDE TO MULTICS FOR SUBSYSTEM WRITERS

CHAPTER VIII THE INPUT-OUTPUT SYSTEM

(DRAFT)

Elliott I. Organick

November 10, 1970

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Chapter 8

The Input-Output System

8.1 Introduction

In many early operating system designs the software known as the input output control system (IOCS) played a central conceptual and functional role. In the pre-multiprogramming, batch operating systems, in fact, many supervisory functions had to do with input output control -- e.g., control over queued jobs, control for management and operation of secondary storage, control for operation of display devices and other peripheral equipment, etc. A system programmer (or subsystem designer) for such operating systems could hardly prove his professional competence without acquiring a reasonable familiarity with the intricacies of the IOCS for his "installation".

By contrast the role played by the input/output control system in a long-lived Multics system is decidedly secondary, at least from a conceptual point of view and certainly tends to diminish over time. Even from a functional point of view the relative importance in Multics enjoyed by the software having responsibility for I/O may tend to attenuate in time. In fact, it will probably prove true that many or even most subsystem designers may be able to achieve their respective objectives while remaining entirely oblivious to the IOCS details of Multics. In the next few paragraphs of this introduction we shall enlarge on (justify) this viewpoint.

One of our objectives in this chapter is to describe the <u>degrees of involvement</u> in or <u>awareness</u> of I/O system details that are appropriate for the reader, depending on his interest and on the type of subsystem he may be planning to design. Before we can succeed with this objective it is believed that a reasonably complete top-down view of the Multics I/O system organization is needed. This we attempt in Section 8-2. Those reading this chapter in its entirety will (hopefully) gain some insight on the relationship of the I/O system to the central supervisory functions of Multics (especially the file system) that have been described in preceding chapters.

There are two related views of Multics which suggest a secondary role for I/O in Multics: First, there is the central fact that the file system makes known and dynamically links files that are stored in the hierarchy, i.e., within the system, to the processes that legitimately request this service. It does not matter whether these files reside on drums, disks or tapes. The users (or for that matter other supervisory modules) are unaware of any explicit data movement in accessing these segments even though physical transfer from actual secondary devices to central memory may in fact be involved and some duly incurred. The required data or procedure object from the hierarchy is made part of the virtual memory of the "requesting" process in a manner such that any data movement that is involved in entirely transparent at the level of ordinary source coding. In other systems, particularly in most earlier ones, a request for an information object on secondary storage always required an explicit request for an I/O transfer in the "source/sink" sense. That is, the source of the object desired had to be identified as a named object and/or location. Correspondingly, if information was to be removed from main memory, it was necessary that the destination (sink) be identified as a named object and/or location, and the transfer (often along with parameters to afford control for a particular type of transfer) be explicitly initiated.

The second view stems from the fact that the information storage within the Multics file system is open ended, being basically a growing storage. Hence, over time there will be a tendency for an increasing proportion of the information needed by a process to be made known (added to virtual memory) through the service of the file system. The corollary observation is the diminishing frequency of need for data and programs that are "original", i.e., that originate outside the system during execution of the process and hence must be input via an I/O activity. In the limit, the Multics I/O activity will be related only to the one or more I/O devices that a user's process would have direct control over, normally for conversation with the system. In most cases this is simply his typewriter or TV console. Moreover, in such cases, thoughtfully designed system default mechanisms are supplied, offering the programmer the option to remain oblivious to specific functions of the I/O system, and to the fact that his process is actually making use of this facility.

The reader should not jump too quickly to the conclusion, however, that the Multics designers' principal objective has been to erect a barrier that prevents the (system or user) programmer from acquiring and exercising full control over I/O devices, whatever they are, be they tapes, special display devices, special communications channels, etc. On the contrary, user processes are able to "negotiate" with the system administrator, who controls distribution of I/O resources, to acquire particular I/O devices (and/or channels). Then, with user code, the user process may program the control of these I/O devices (and channels) and operate them with the full freedom that is normally accorded a hardcore system programmer. One of our purposes in this chapter is to provide at least an initial guide to this programming flexibility for those interested.

In brief, the Multics I/O system has been designed using guidelines that would be followed in the design of any good multi-purpose tool:

a) the simplest, most commonplace use of it requires only a minimum of knowledge and skill -- and the overhead for such simple (common mode) use is also minimized.

b) to extract more tailored (special purpose) services there is added cost -- both in the time that must be committed to understand how the tool works and in the actual overhead that will be incurred in execution.

8.2 I/O System Organizational Overview

In our introduction we claimed that there are different levels or degrees of potential involvement in I/O system implementation that would be appropriate to each subsystem application. We shall enumerate and exemplify these in succeeding sections. But, first, we are in need of an effective frame of reference to guarantee meaning for the delineations we shall be making. An organizational overview of the I/O system is what is wanted for this purpose.

Such an overview must begin by recognizing an overriding design objective for any general I/O system; namely: the input/output operations stated in the programs or service procedures that a user writes should specify only those device functions that are required for the application at hand, leaving to the system the responsibility for gauging the degree of device independence implied by the user's request. In this way a user who invokes such service procedures is free to designate substitute devices as may be appropriate, while adhering to the device dependencies that are implied by the stated I/O function requests. (For most ordinary users whose sole I/O device is normally just the console, this objective amounts to an opportunity for the user to state his I/O requests in a manner that implies device independence. Moreover, the identity or special idiosyncrasies of the particular I/O device used in this fashion is of no concern to him either.) For this reason user-coded I/O operations of a process should ordinarily be independent (or as independent as feasible) of the particular device and model, or even of the type of device, e.g., typewriter, as opposed to teletype or tape.

There are two clear reasons for this crucially important objective. First, we must presume that at any given time a system will generally accomodate several types of I/O devices and models. Each is likely to require different programmed control. Each may have different character sets, and may be intrinsically different in various respects (e.g., line printers are not backspaceable, tapes are; some tapes can be read backwards as well as forwards, while card readers are never designed to read cards backwards, etc.) Second, we presume that I/O devices become obsolete and, over time, are replaced by new models of the same or different types, e.g., keyboard-TV versus typewriter. Clearly, if programs are to be reusable, if processes are to be repeated with minor or no variation in the nature or effect of their I/O operations during reuse of these programs, then recognition of device independence must be a planned part of the programming system for I/O operations.

One approach to design for the needed device independence is to regard the I/O resource needed to complete any given I/O operation not as a <u>real</u> or physical resource, as for instance a particular card reader, but as a <u>virtual</u> (pseudo) I/O resource that is described in terms of the functions it must be capable of performing, which is mapped by the system to a particular real resource at run time, using whatever I/O device is available and convenient. The analogy here is with virtual memory, regarded as a resource, which is mapped by the system into particular blocks of core memory using the segmentation and paging features of the hardware and in a way that is transparent to the user. Such an approach implies that all available input devices, regardless of type (or location) are in some sense acceptable equivalents and all output devices are correspondingly equivalent.

Unfortunately, even if we exclude the user's own console, which normally must be fixed during the life of the process, it is still a poor analogy if interpreted too strictly. The user must, when he so chooses, be able to decide what I/O devices he wants used (when there is a choice available to him.) If, for instance, he wants to develop a subsystem whose output may optionally drive either a dedicated line printer, 30-column card punch or 80-column card punch, he must be allowed to specify which one, or which combination of two or three, and in what prescribed order. In short, a completely flexible I/O system must provide for user designation of the specifics of certain I/O operations -- and even of user-provided devices (or simulated devices) in certain cases. For example, a user develops a subsystem whose output will drive a newly acquired display device. He may be required to furnish the detailed I/O coding for the control of that device (later referred to as a Device Interface Module or DIM). Those interested in seeing what is involved for such an application will hopefully find this chapter helpful, but should regard this entire chapter merely a jump-off point for more extended reading of the MSPM).

Certainly some kind of compromise arrangement is needed whereby some users (most, in fact) may code their processes so that I/O is regarded as employing virtual resources while others may code I/O operations by partially or completely specifying the devices to be used and the programmed control to go with it. The former use the so-called package I/O calls, such as ios_$read_ptr, while the latter will come to grips with and effectively utilize the basic functions of the I/O system itself in varying degrees of involvement. Some details of these calls and related techniques are described in sections 8.3 and 8.4. Readers may already have encountered descriptions of these calls in the MPM.

The particular design approach taken in Multics is based on two practical requirements, one having to do with the discharge of the system's responsibility for dispensing and recovery of all real I/O devices, and the other having to do with the run-time mapping of valid user-coded I/O operations, regardless of their degree of specificity, onto specific devices and in the manner and with controls appropriate to those specific devices.

First, it is recognized that at any given time, as a consequence of the I/O device needs of a process, certain specific I/O devices (or device capabilities) must be allocated to each or any given (user) process. The system's decision to allocate from available I/O device resources to a process will be made for any of several reasons. For ordinary situations the system is able easily to infer those needs, e.g., the console is needed on which a user logs in. In more exotic cases, the user can negotiate these allocations in advance with the system's administrator, or eventually obtain these resources at run-time via commands or library subroutine calls.

Second, any programmed I/O operation should at source level, at least, be expressed (coded) in a general way that specifies the I/O source or sink, not by its device designation but only by a placeholder name for that source or sink. (Moreover, as an added convenience to users, it may be possible to code certain standard I/O operations so that even this name may be inferred from context.)

For example, [and here we illustrate only schematically], rather than use a specific device designation, even though that device may in fact already be allocated to a process at the time its use is wanted, such as in the following forms:

```
          read from "card_reader_2" into area_23;       )
or        input ("card_reader_2", area_23 )     ;       |
or        read ("device 35_2", area_23    )     ;       >  ①
or        input ("console 204", area_23   )     ;       |
or        call io (input,"console 204", area_23);       )
```

we might instead say:

```
          read from the stream named "Billy" into area 23;   )
or        read ("Billy", area_23);                           |
or        call read ("my_console", area_23);                 >  ②
or        call io (read, "my console", area_23);             )
```

depending on the syntax of the coding language being used.

Here in examples ②, "Billy" and "my_console" are simply identifiers for sources of data. For such a read statement to have any meaningful effect, the specific device represented by that identifier must be *bound* to or "attached" to (i.e., associated in some way with) "Billy" or "my_console" at some time *after* the device is allocated to the process and *before* the read statement is executed. The Multics I/O system is responsible for maintenance and supervision of these device-source name associations. Like-wise for output, names for *sinks* are used in write statements rather than actual output device designations. Thus by analogy to the read examples in ② above we could conceivably picture something like

```
          write ("his_console", "format 12", area_22);      ③
```

in which "his_console" is here intended to suggest the name of some sink (output device). The attachment at any given time may be to one of a set of several (different) devices. Thus, if a single process had several consoles allocated, the process could simulate a "party-line" conversation on the several consoles where the name "his_console" could be attached and reattached, possibly cyclically, among the several different allocated devices.

A generic name for elements of the set {source, sink} that has now found favor is stream. We shall use this term frequently hereafter. Thus the term "stream name" refers to either a name of a source or a name of a sink. It is clear why the word stream is selected since an input or output operation suggests a stream of information (words, or characters, or bytes, or bits) flowing from a source (input device) or to a sink (output device). Conceptually, the attaching of a stream name to a particular device is a form of parameter binding. The device designation plays the role of the actual argument and the stream name that of the formal parameter. In order to apply more than one "argument" to the same "parameter" Multics provides for the detaching of a device (designation) from a stream name so that subsequently another device can be attached to the same stream name.

To carry out a read or write operation (call) of the type suggested in ② and ③ above, the following steps can now be visualized. The system module that receives and is responsible for "interpreting" this call must first perform a table look-up (in a per process, per-ring data base) to determine the device designation (and type of device, constraint rules, if any, for use, etc.) that is currently associated with the named I/O stream parameter. [Because attachments are maintained on a per-ring basis, a subsystem that executes in a special ring can have a distinct set of stream name "meanings".]

In principle, assuming the I/O call parameters are consistent with the data kept in this so-called "attach table", this same I/O control module can then convert this request into an I/O action -- i.e., by initiating the desired I/O operations after generating the required channel commands*, etc. Because the system must be capable of supporting an open-ended number of devices, device types, and controllers, considerably more modularity is called for. So, in actual fact, the I/O control module merely transmits the now more specific I/O request as a call to an appropriate "specialist" module (called a Device Interface Module or DIM). There is one such specialist module for each type of device. This DIM in turn takes charge of getting the I/O request accomplished as suggested in Figure 8-1.

To get a better grasp of what the specialist module's job is, it is worthwhile to degress momentarily to consider some of the special characteristics of the GE645 I/O controller hardware. The input/output controller hardware of Multics is designed so that each individual I/O device may be (and in fact normally is) in effect connected to a separate I/O channel. By I/O channel we mean (here) conceptually a separate I/O processor capable of accepting commands that carry out

---

* IBM set a trend by calling the I/O channel instructions on its model 709 computer commands to distinguish them from the instructions of the CPU. This distinction became a rather conventional notation that has remained popular for over ten years. In the GE645, however, these channel instructions are called "data control words" or DCW's.

I/O operations.* Hence, when we speak about allocating a device to a process (group) we shall also take for granted that the system also more or less permanently allocates a channel for this device. The identity of the channel, the identity of the device connected to it, and the identity of the current owner process (group) can be regarded as system-maintained in conveniently-organized (ring-0) system data bases available to the modules of the I/O system.

Communication with the channels, i.e., initiating their activity and supplying them with their needed commands, receiving and interpreting the status information that they return, etc., is achieved in the GE645 system by providing a peripheral processor called a GIOC (Generalized I/O Controller). This active hardware device acts as a high speed "broker" to manage (and multiplex) the communication between memory and the many I/O channels (which are themselves packaged in the GIOC). Each GIOC is logically organized to provide half duplex (one way) communication service for up to 2000 input devices, output devices, or devices that alternate as input and output devices**, or up to 1000 full duplex units such as typewriter or keyboard-TV consoles.

We can now return to complete our view of the I/O system's handling of read/write calls as suggested in Figure 8-1. Each device interface

---

* Those who later have occasion to study the reference literature on the GE645 I/O Controller hardware will find that the term channel is used in a more restricted and technical sense. There, several such channels, taken together, comprise what we speak of in this chapter as an I/O Channel.

** Such as the IBM 2741 typewriter console.

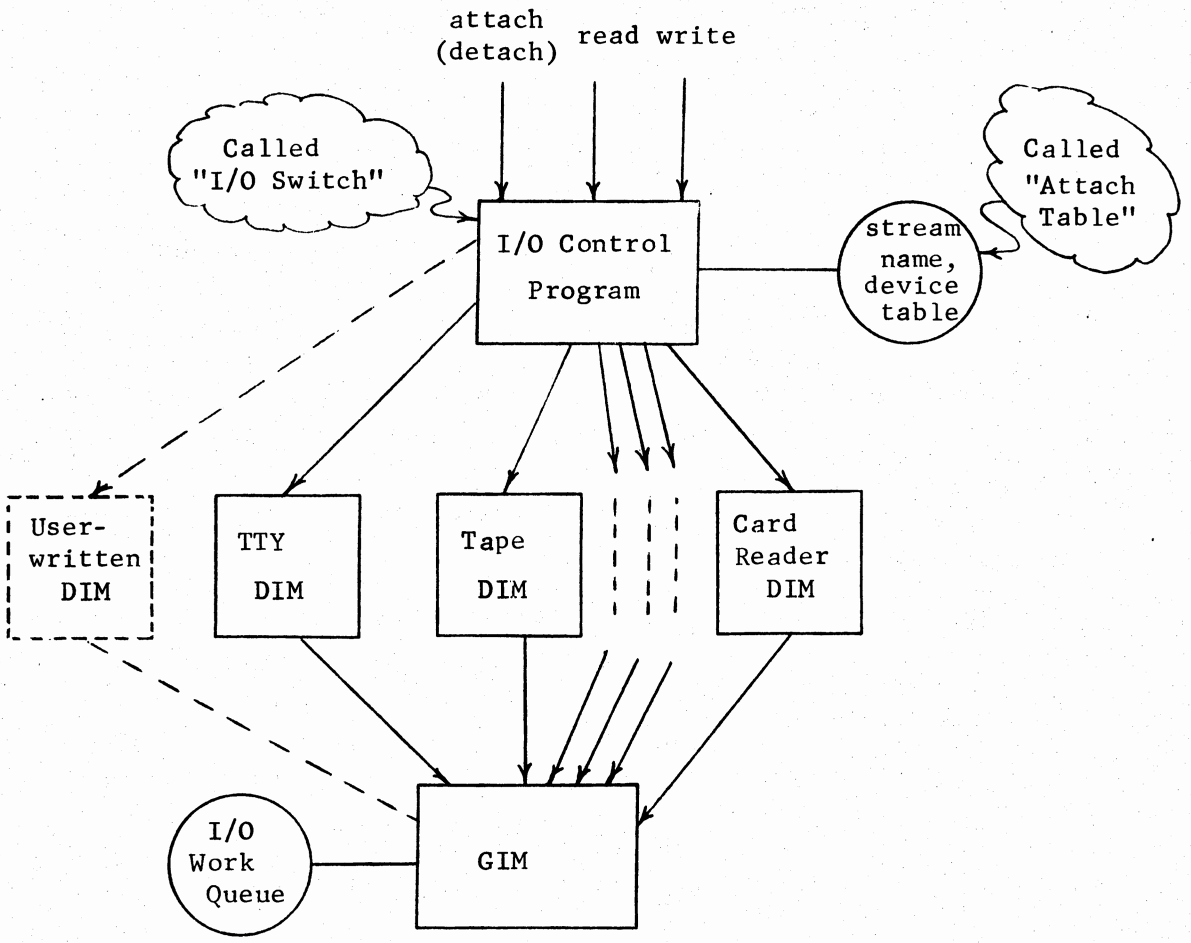


Figure 8-1 First simplified view of I/O system organization

module (DIM) performs a number of functions: The description that follows is inspired by a recent description of Graham*.

<u>The DIM converts a device independent request into a device dependent one</u>. In doing this, it must compile a program for the hardware input/output controller GIOC (which it can in turn supply to the target channel). The compiled program reflects the idiosyncracies of the particular device to which the stream is attached. It (the program) may include line controls in the case of remote terminals, select instructions in the case of tapes, and so forth. In addition the device interface module may need to convert the internal character code used by the system into an appropriate character code for the device. Typewriter terminals for example, come in many different varieties. Virtually every different variety has different character codes.

The device interface module, after compiling a program for the GIOC, calls a module that serves as an interface for the GIOC to start the I/O using this GIOC program. It is the DIM's responsibility to interact with the GIOC Interface Module (abbreviated as GIM) until this I/O request has been completed. This may require several calls to the GIM depending on the format of the channel programs that the GIOC can provide to the channels for execution.

---

* "File management and related topics", by Robert M. Graham, © 1969, p. 48. Course notes issued at the 1969 Engineering Summer Conference, University of Michigan on Computer and Program Organization.

The GIOC Interface Module (a ring-0 CPU program) is responsible for the overall management of the GIOC. Thus, the GIM is also responsible for overall monitoring of the operation of the GIOC. This requires answering interrupts (i.e., that its code acts as an interrupt handler for), recognizing completion of tasks (and transmitting to its caller status information deposited by the GIOC).

A final point of explanation for Figure 8-1 regards the four indicated entry points to the I/O Control Program. The entry attach is always employed (to establish the appropriate stream name-device association in the attach table) prior to utilizing the entry points read or write for the same streams. The entry point detach is used to nullify a previous attach stream name-device pairing.

Generalization of the device concept to include files.

If we now add one powerful, in fact crucial, generalization to the I/O system organization picture painted thus far, we can then see the actual Multics design overview in its entirety. That generalization is the one which permits segments to be substituted for I/O devices in the association with stream names (let us pause briefly to let the last sentence sink in.) What we mean by this remark is that any named segment of a user's process may be employed as a source or sink for a read or write function, as if it were (or in lieu of) an actual I/O device.

There are a number of important applications that are now possible as a result of this type of generalization. For instance during a console debugging session, information can be read out of storage on to the user's console for visual verification. Once verified, the results of subsequent but similar computations might be more appropriately outputted to segments and thus saved as files. To achieve this objective, the programmer might follow these steps:

1. (For debugging) Call attach to associate the stream name, say, "user_output" with his console, e.g., tty302.
2. (For production runs, after debugging) Call detach to nullify the previous call to attach. Call attach once again, this time to associate the same stream name "user_output" with, say, <results_file>.

Other applications are those that make it possible to simulate I/O devices for interactive or conversational interplay, sending mail, i.e., output to data bases shared with other users, converting console sessions to run absentee by placing on one file the sequence of commands which can be read as an input file and placing (writing) the series of resulting responses on another file. (The dialogue's results can be examined later at leisure by asking for a printout of the output file.)

This remarkable generalization of the notion of input/output is achieved in a mechanically almost trivial manner in Multics. The trick is simply to create another specialist DIM called the file system interface DIM, as shown in Figure 8-2, which completes the I/O system organization overview. Again we follow closely Graham's description of this module's function.

The file system interface DIM functions like any other DIM. However it does not call the GIOC Interface Module. <u>The file system interface DIM is used to make a segment look like an I/O device</u>. In its (per process) static storage, the file system interface DIM maintains a table holding status information for each segment which is being referred to as a device. When an attach call is made to the I/O control program for attaching a stream to a segment (instead of to an actual device), the requested segment is initiated as a known segment (if not already known). [See Chapter 6 as a refresher for the details on making a segment known.] The file system interface DIM maintains in the table of status information separate read/write indexes of the current positions in the segment where reading or writing is taking place. Subsequent read or write calls are processed by the file system interface DIM and consist of copying the requested information into or out of the segment at the position of the appropriate read/write index. After the copy is made, the index or "current pointer" is updated to the new position of the segment.

We have just seen how the I/O system may behave as a "customer", of the file system which supplies needed services. One might wonder if the reverse of these roles is ever true. That is, does the file system through its page control module, when seeking to transfer a page of information from/to core and disk or tape storage, ever find itself to be a customer of the I/O system? Emphasis on objectives of modularity might cause one to guess the affirmative. In actual fact, however, considerations of efficiency have dictated that paging I/O in Multics be treated with special purpose I/O software that greatly streamlines the processor's task in initiating and controlling such I/O. The details of this special purpose

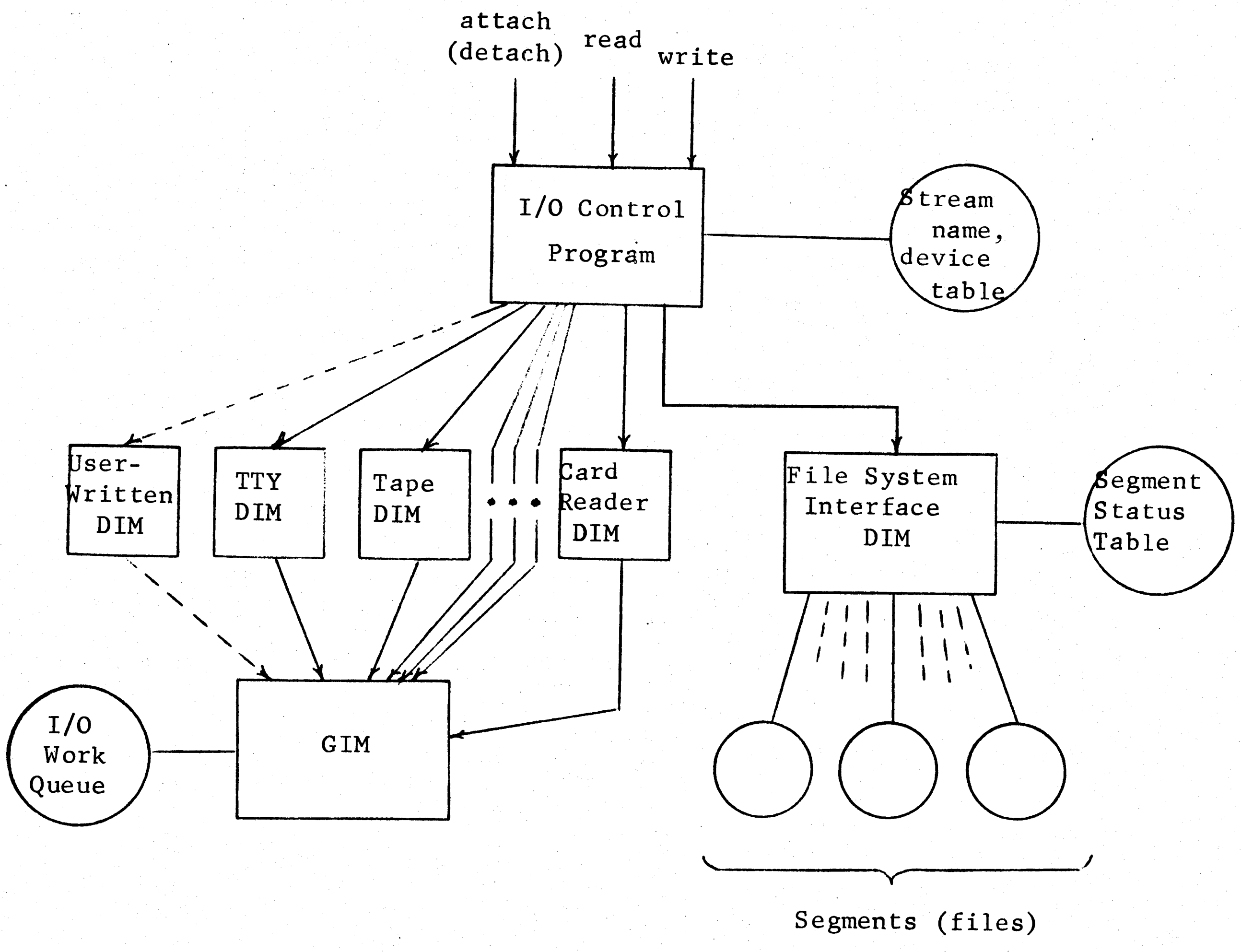


Figure 8-2 Complete simplified view of the Multics I/O system organization

software should be of no interest to subsystem designers and for this reason will not be covered here.

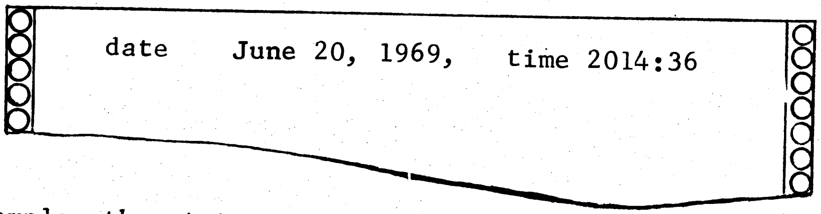
8.3 Packaged I/O for Communication with the Console

Typical of all operating systems that support interactive processing, Multics furnishes several routines (and enables them) to satisfy the typical user's need for an easy-to-express console I/O request, i.e., one that requires a minimum of acquaintance with the I/O system organization. These routines have been referred to as "packaged I/O"*. They serve effectively as interfaces with the I/O system by mapping the received call into the appropriate (and more technical) calls to the I/O system.

One of the simplest of the interface routines currently available is the simplified print routine called ioa_ which can be used to construct and print messages made up of character strings, integers and pointer values. For instance, a call written in PL/1 Syntax, might be:

```
call ioa_ ("date    ^a ^d, ^d,   time ^d:^d",
                "June", 20, 1969, 2014, 36);
```

When executed this call would produce a typed line that might appear as:



```
date    June 20, 1969,    time 2014:36
```

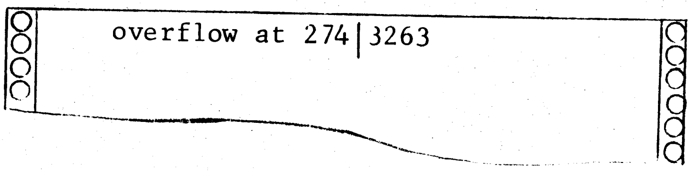
As another example, the statement,

```
call ioa_ ("overflow at ^p", pointer_variable);
```

---

* This is terminology used in the Multics Programmer's Manual, Section II, 3.7

would produce when executed a typed line that might appear as



```
overflow at 274|3263
```

The first argument is a control string which is a form of format code somewhat similar to a Fortran format (but more limited in purpose). We'll not dwell on the details of this routine or its variants as they are well described --- with additional illustrations --- in the MPM. We only consider here how functions like ioa_ behave in the context of the operating system organization structure just overviewed in the preceding section.

To understand how ioa_ is able to do its intended job we must appreciate the following system service conventions that are obeyed for each process created at log-in.

First, the identification of the console device-channel pair on which the log-in attempt is made is noted by the user control process (i.e., the "answering service") that responds to the dialup. When log-in validations are completed and the process is created on the user's behalf, the console device is allocated to that created process and the stream name "user_io" is "attached" to this device by calling the I/O control module at the entry point <u>attach</u> and supplying as arguments the necessary information to complete a suitable entry in that process' attach table. (Note that the interface routine can be oblivious to the type of device the user is logging in on. The I/O control module has responsibility for directing the I/O request to the appropriate device interface module). In Section

8.4 we shall take a closer look at this bit of business, but for the moment we merely note that in fact several calls are made to the attach entry, resulting in a table entry that shows the names "user output" (as well as "user-input") as synonyms for the stream name, "user_io", where "user_io" means the tty he logged in on. Second, we must also appreciate that ioa_ is written to call the write function using "user_output" as the I/O name. In general, packaged I/O interface routines always use the fixed names "user_output" or "user_input" as I/O names for their function calls to the I/O system.

We now see that when the log-in process is completed, and the user is "put in charge", the necessary connections that essentially enable ioa_ to perform properly have all been made. The same connections (i.e., attachments) are also made for other packaged I/O routines. Of particular interest in this category is a pair of routines for reading or writing typewriter I/O. These two routines, whose full names are ios_$read_ptr and ios_$write_ptr, are fully described in the MPM.

For example,

```
call ios_$read_ptr (stringv_ptr,rdmax,rdcount);
```

requests that up to rdmax characters be accepted from the console for assignment to the character string variable, pointed to by stringv_ptr. The output argument rdcount reports the number of characters actually read from the typed line.

A subsystem writer will be pleased to observe that the effective sources or sinks for packaged I/O routines are easily changed to any device or file of his choosing. For instance, if the only explicitly called I/O routines in the subsystem are of the packaged variety, then to convert the subsystem to run absentee, the only requirement for the change is to cause execution of an I/O system call to detach the "user_input" and/or "user_output" synonyms from the stream name "user_io", and then cause explicit attach calls to reassociate the names "user_input" and/or "user_output" with the designated files (named segments). These I/O system calls could be executed (preferably at command level) just after log-in or at any time afterward when absentee mode execution becomes appropriate. This flexibility may motivate some readers to investigate the attach call and the related I/O system calls that are discussed in the next section.

8.4 I/O System Calls (ios_)

These direct calls to the I/O system, a total of over twenty by current count, are enumerated and fully described in the MSPM as well as in the Multics Programmer's Manual. (The details for specifying all the arguments for these calls, however, are somewhat scattered throughout the latter.) Here we shall give brief descriptions of some of these calls and their use but avoid a detailed technical description. Four of the most frequently used entry points are listed in Table 8-1.

8.4.1 <u>ios_$attach</u>

To attach a device to a stream name, the programmer must specify the following (indicated arguments* are given to the left):

* The first three arguments (ioname1, type, and ioname2 must be supplied as character strings).

Table 8-1

The Most Frequently Used I/O System Entry Points and Their Arguments

Entry point name	Arguments										
	ioname	ioname1	type†	ioname2*	mode†	wkspace	offset	nelem	nelemt (output)	disposal	status‡ (output)
ios_$attach		✓	✓	✓	✓						✓
ios_$detach		✓		✓						✓	✓
ios_$read	✓					✓	✓	✓	✓		✓
ios_$write	✓					✓	✓	✓	✓		✓

* for details, see Section 1.41, Reference Data Section, MPM.

† for details, see Sections 1.4 and 1.41, Reference Data Section, MPM.

‡ for details, see Section 1.4, Reference Data Section, MPM.

ioname1	-- the stream name
type	-- the type of device, to designate the appropriate DIM, e.g., "tdsm" for the tape DIM, or "file" for the file system interface DIM.
ioname2	-- the device designation, e.g., "tty138" for a typewriter, or "Harry" for a file (i.e., a relative path name)
mode	-- a code to designate the kind of restraints to be placed on the use of the device, its method of accessing, and its interpretation of the data representation (and any other type of constraint or optional use or function that is deemed appropriate during this particular attachment.)
status	-- the name of a (72-bit) variable to record the response of this request .. i.e., in which to receive the reflected error messages, warnings, or other advice from the modules that are the dynamic descendents of this call.

The MPM (Reference Data Sections 1.4, and 1.41) maintains an up-to-date list of all the DIMs that one can designate in an attach call. This reference also provides the subsystem writer a complete list of function calls besides attach, detach, read and/or write that are meaningful for each system-supplied DIM. The same reference supplies a list of the default modes that are set for use in package I/O read/write calls. The coding for modes recognized by the system-supplied DIM's is explained in the same reference. (Subsystem writers who must write their own DIM's are urged to follow this coding although the particular code scheme that one devises is up to the DIM designer, since the I/O control module simply transmits this code to the target DIM and is otherwise oblivious to it.) The mode code for system-supplied DIM's is simply a character string of concatenated pre-defined code characters, one or more characters per each <u>use</u>, <u>access</u> or <u>data</u> mode. For example, a mode code to attach a tape unit for reading only (use), forward only (access) and only logical, linear records (data) would be the concatenation, "RFGL" for <u>r</u>eadable, <u>f</u>orward, lo<u>g</u>ical, <u>l</u>inear.

In the initial Multics implementation the mode argument may be null (i.e., "") because the I/O switch is not programmed to check the mode. Ultimately, however, it is planned that the "switch" will check the <u>use code</u> portion of the mode argument (i.e., to see if codes R, for readable, and/or W for writable are compatible with the attached device) and will pass the remainder of the mode argument (access mode and data mode codes) to the DIM for further compatibility checks.

A complete explanation (interpretation) of the output argument called <u>status</u> is also maintained in the same Reference Data Section (1.4). Many of these error explanations should make more sense upon completing the reading of this section (8.4). Attempts to attach a device should fail if there is an incompatibility of the supplied arguments either with what is already recorded in the attach table for the process, or with what is recorded in or known about the target DIM.

An attempt to attach a named segment for use as an <u>input</u> file will result in an error status return from ios_$attach if the specified segment cannot be found. An attempt to attach a specified segment for use as an output file will result in one of two actions. If the segment cannot be found, then one will be created and used (without comment). If the segment is found, writing into it will be by appending to the end of it.

In MPM parlance the I/O control module (which maintains the attach table) is referred to as the "I/O switch" because during function calls like ios_$read or ios_$write, the job of this module is to route or switch the incoming call not only to the appropriate DIM, but also to direct the call to the appropriate entry within the target DIM.

Each target DIM has an entry point transfer vector, one entry per each of the functions supported by that DIM. The transfer vector is consulted when the DIM is called by the I/O switch for routing the call to the appropriate functional entry point, e.g., read, write, backspace, rewind, etc. A call to a function that is not supported within the particular DIM (e.g., to read a printer) will reflect an error code when and if it is called. (Section 8.5 gives more details on the design of a DIM including naming conventions for DIMs and for the calls the DIMs may receive.)

By special design, a user who wishes to provide synonyms for a given stream name may do so by executing a call to ios_$attach in which the arguments take on their special meaning when the argument called type is supplied with the value "syn". For instance to assign the string "his_io" as a synonym for an already-attached stream named "her_io" one could write:

```
call ios_$attach ("his_io", "syn", "her_io", mode, status);
```

8.4.2 ios_$detach

To detach or nullify a previous attachment, i.e., delete the entire attach-table entry previously recorded for a given stream name -- remember, there may be one and only one entry for each stream name -- one calls ios_$detach and names the streamname and, as a redundancy check, the (presumed) device associated with that stream. A status return argument provides a report of the resulting action. Incompatibility (or invalidity) of the input arguments will result in an error message reflected from the I/O control module (I/O switch).

The input argument called disposal may be null (i.e., "") for most applications. It is planted in this call for future use, to provide special instruction to the system or operator for the disposal of dedicated I/O devices such as tapes, and/or magnetic tape drives. (Both tapes and drives can be considered as independent resources to be disposed of.)

When fully implemented, a null value for the disposal argument will mean: take the default action and close out the device to allow future assignment to another user (e.g., dismount the tape). Alternatively, a value of "hold" for the disposal argument will mean: keep the device active ("I will be back").

8.4.3 ios_$read

To execute a read the caller is obliged to name the input stream (source) and a pointer argument (workspace) that identifies the destination in the process' virtual memory where the input data is to be transmitted. Additional qualifying input arguments are the offset and the number of elements, nelem, that are to be read. The offset is from the beginning of the workspace in which to begin receiving the input data. [Offset is measured in elements where the element size is usually set by a default convention for each DIM. Typical element sizes might be 9 bits for character-oriented streams or 36 bits for word-oriented streams]. The number of elements specified is, of course, subject to the current upper-bound restraint imposed by the segment size of the workspace.

There are two output arguments, nelemt and status. The former provides a report on the actual number of elements read into the workspace while status provides the normal advice on success or degree of success of the read operation. The report can reflect error reports transmitted from a variety of sources, including the GIM, or in the case of the file devices, from file system modules. (It may even reflect an error message that alerts the user to trouble caused during the preceding transaction -- an indication of interest during certain asynchronous reading modes described later.)

8.4.4 ios_$write

As can be seen from Table 8-1, this call employs the same set of arguments. Their interpretation is what would be expected by symmetry with ios_$read. The workspace pointer and offset identify the place from which writing-out is to begin. The number of elements to be written out, nelem, and the number actually written out, nelemt, provide input instruction and output reporting respectively. Status provides additional information to complete the reporting responsibility.

8.4.5 Other calls

Several calls are available to provide the subsystem writer more flexibility or control of the use of I/O devices. For instance,

ios_$changemode

allows one to alter the current mode of a given attachment. Several calls deal with control over the synchrony of the read or write operations and/or of the workspaces employed in these operations. We don't burden the reader here with the actual names of these calls or their arguments. These can be found in the MPM. We will, however, digress here to discuss the subject of I/O synchronization control at a conceptual level.

Read/write synchrony refers to the type of coupling one wants (loose [asynchronous] or rigid [synchronous]) between the actual initiation of the I/O transfer and the corresponding read/write call in the user's program. Workspace synchrony refers to the type coupling one wants (loose or rigid) between the point in time when the I/O workspace has been filled/emptied and the point in time when the program may resume execution beyond the read/write call (that would cause the workspace to fill/empty.) These two types of synchronization are mutually orthogonal, so a user may wish to specify particular combinations for his subsystem application when other than the systemwide default selections are wanted. The next few paragraphs elaborate on each type of synchronization control and suggest several applications.

Read (synchronous or asynchronous)

Shall "read-ahead" be permitted or not? That is the question. By read-ahead (asynchronous) we mean permitting the system to anticipate our program's read request by issuing an I/O order to read the attached device before our program actually executes the corresponding read call. Read-ahead is what is meant by read asynchrony and is precisely what is wanted in the typical console read operation. This allows the information that the user types ahead to be available in core when the program issues the read call. Hence, the system's default mode for typewriter input is asynchronous.

A read synchronous operation implies, "Don't read a thing from the device until a call for it has been issued. This input mode lock-steps a user to the program, thus in effect reversing the normal master/slave interactive relationship between them. Now the program is in control of the user rather than vice versa. Read synchronous might be useful in certain computer-assisted instruction applications or in situations where, say, no further requests may be accepted from an inquiry station that is attached to a subsystem.

Write (synchronous or asynchronous)

Shall "write-behind" be permitted or not? This is the question here. If so, then return is possible from the write call before all output information designated in the write call has been transferred to the device. In most applications write asynchronous is acceptable and in fact, highly desirable for the sake of efficiency (so long as it is safe). Write asynchronous is the system's default decision. There are cases, however, where write synchronous operations are required. The system, for example, uses this mode of output during automatic logout of a user to be sure that all messages and other I/O transactions have been completed before taking subsequent action.

<u>Workspace</u> <u>(synchronous</u> <u>or</u> <u>asynchronous</u>)

Shall permission to return from a read/write call be permitted before the workspace designated in that call has been filled/emptied? That is the question asked here. The system's default decision is <u>workspace</u> <u>synchronous</u> (i.e., <u>no</u> return of control from the DIM until the workspace has been ascertained to be filled for the designated read (or emptied for the designated write). Conceivably some speedup of a read/write asynchronous action can be achieved by opting for workspace asynchronous but this is risky practice because a succession of reads (or writes) could conceivably cause chaotic overlaps in the workspace areas; so, unless there is a special purpose application where the user feels safe in doing so, workspace asynchronous is <u>not</u> recommended.

To summarize our foregoing discussion, the defaults for the synchronization options are:

1. reading is <u>asynchronous</u>
2. writing is <u>asynchronous</u>
3. workspace use is <u>synchronous</u>

Still other calls provide the user an opportunity for such functions as abandoning data piled up as read-ahead in the input workspace so as to reuse this space for new reads. A symmetrical call aborts any, as yet physically unwritten, data that may be piled up as write-behind in the output workspace. These calls may be especially useful for designers of subsystems dealing with typewriter-like consoles where the read-ahead or write-behinds can become numerous in certain conversations. Often the need to reset the current pointers in the read or write workspace becomes essential to avoid frustrating the console user, for instance by accepting previously typed but now undesirable input or by typing out now unwanted results.

Still other calls allow the user to control (or determine) the size of input or output elements for next (or current) reads or writes. These calls may be useful in certain applications where the device is the typewriter or a file.

In addition, there are ios_ calls to allow the user to control (or determine) the set of read delimiters and/or break characters in input streams. <u>Read delimiters</u> are used to condition read calls (e.g., the new line character for typewriter read calls) so they halt their scan of the input buffer after a read delimiter is seen (and transmit all characters seen up to that point, i.e., up to and including the read delimiter, to the user). <u>Break characters</u> are used to control the action of an "interactive" channel so that it will trigger a hardware interrupt (when such a character arrives over the channel), so as to make all data read since receiving the last break character available to the user. Break characters are useful to achieve a form of code conversion or editing known as "cannonicalization" (see MPM, Section 2.5 for an orientation and a full explanation). They are also used to achieve erase and kill effects. These, too, are a form of immediate editing that has been found essential in the the typing of input streams. (Section 2.5 of the MPM elaborates). For typewriters, new line is not only set as the read delimiter but is also set as a break character.

When segments are attached as pseudo input devices, say for absentee jobs, it would be nice if the file system interface DIM could respond to both read delimiters and to break characters in input messages so as to fully simulate the action and effects of, say, reading typewriter input data. Although the file system interface DIM <u>does</u> accept delimiters (e.g., new-line), it does <u>not</u>, however, accept break characters, because it is not an interactive channel. Another point to note is that, as a result of a design decision, the file system interface DIM supports (permits) no code conversion during input from a segment.

The default values of new-line for the read delimiter and 9 (bits) for element size make it especially easy to have segments substitute for typewriter devices (that transmit ASCII character strings). But the user may choose any read delimiter and/or any element size simply by executing appropriate calls to ios_$setdelim and ios_$setsize for the streamname in question.

The fact that the file system interface DIM does not itself produce code conversion during input from a segment is no serious restriction either. The user's program that invokes the action of the file system DIM is certainly free to perform the needed conversion steps either before or after the simulated input operation.

Finally, as if this portfolio of possible user-originated I/O control were not enough, the Multics designers have planned an open end to the list in the form of a special, catch-all call,

ios_$order

This call permits the sophisticated subsystem writer to transmit special requests to the target DIM of subsequent read/write calls, such as the setting of hardware device modes on typewriters or tape drives, e.g., red or blue ribbon, high or low tape density.

8.5 Designing a DIM

Users may write non-privileged device interface modules* for a variety of purposes, usually to control a particular device or set of devices, but occasionally to serve as an intermediate interface with existing DIMs. To construct a DIM that controls an actual device the subsystem designer must become thoroughly acquainted with the channel adapter that communicates with the device (or its controller). The channel adapter is connected to

* The current implementation of Multics includes certain privileged (ring-0) system DIM's designed to satisfy certain high-performance requirements (e.g., typewriter response). The design details of these system DIM's are not considered in this Guide.

the GIOC. Additionally the programmer must become acquainted with the GIOC hardware, and with the software module that manages this hardware and that interfaces with the DIM on its out-going end. He must also be familiar with the system-supplied I/O switch which must call the DIM.

The DIM is the only module in the chain of calls that knows about the I/O device being controlled. On the user's side of the DIM the I/O switch routes the user's call to the proper DIM, checks for and transmits compatible sets of arguments, otherwise generating and returning appropriate error messages generated as a result of the I/O action itself. On the device side of the DIM the GIOC Interface Module (GIM) transmits control information to the GIOC, selects and reacts to interrupts received from the GIOC, and interprets and transmits status information to the DIM. The GIM also allocates and controls the use of channel buffers that must necessarily lie outside the user's virtual memory space. The GIM is designed so it need not care how the user makes use of a channel that it, as GIOC "manager", assigns to the user (through the DIM). Of the three modules in the chain, namely the I/O switch, the DIM, and the GIM, it is the GIM alone that provides for the system's safety in carrying out these various functions.

A user may also design a pseudo DIM which acts merely as an intermediate module between the I/O switch and one or more existing DIMs. To write a pseudo DIM one need know nothing about the GIM or the hardware it serves.

A "broadcaster" module would be an example of such a pseudo DIM. Its purpose might be to receive a single write call from the I/O switch and convert it to two or more ios_$write calls via the switch, this time to the actual DIM or DIMs that will in turn cause replication of the message on the devices in the "broadcast" set.

Schematically, this idea is easily displayed. For example, we may picture a call of the form:

```
        call ios_$attach ("A", "broadcast", "C", "D", "E", mode, status);
```

where we assume that "C", "D", and "E" are I/O stream names that are themselves already attached (to their respective devices).

The coding for the broadcast pseudo-DIM would then be such as to anticipate and process a call of the form

```
        call ios_$write ("A", workspace, offset, etc.);
```

so as to generate and execute the following three calls (and then return);

```
        call ios_$write ("C", workspace, offset, etc.);
        call ios_$write ("D", workspace, offset, etc.),
        call ios_$write ("E", workspace, offset, etc.);
```

In the subsections which follow we recite in a more systematic fashion a "checklist" of things a DIM writer needs to know.

8.5.1 Conformity with other DIMs

There are some general rules of conformity that are worth reviewing when approaching the design of a DIM. These ideas are given here.

1. It goes without saying that a DIM or pseudo DIM designer must become acquainted with and try to adhere to all the ios_ conventions for communicating between the user and the I/O system. In this way the new DIM can be used interchangeably with other DIMs and thereby preserve the appearance of device independence. Conventions which are enumerated in the BF sections of the MSPM have to do with:

   (a) interpretation of error codes,

   (b) attach modes,

   (c) treating default strategies re: element size, read delimiters, etc.

2. In the same spirit of maximizing uniformity of application, calls to ios_ through the switch should be matched to the device's functions in a meaningful way, if necessary, using the many special calls already in use by other system-supplied DIMs, e.g., ios_$changemode, ios_$setsize, ios_$seek, ios_$tell, etc., and as a last resort, ios_$order.

   Achieving this type of conformity can well have a high payoff during the debugging of a new subsystem that includes a new DIM. Remember that during early debugging of the subsystem the actual I/O device will probably not be physically connected to the system. So, the device will have to be simulated by using existing equipment, using the existing system-supplied DIM. (Letting the device be represented by a file and using the existing file DIM is regarded as the best choice.) Cutover to the actual DIM and the actual device should therefore present fewer problems if the new DIM presents a similar interface as the one used in the preliminary testing.

3. As a general rule of good design the DIM should be constructed to expect any reasonable ios_ call and do something sensible with it, i.e., not reject it. For example, if a device has only one read delimiter, say the new line character, then a call transmitted via the I/O switch to set the delimiter to new line should be accepted, (and of course ignored).

8.5.2 <u>The DIM's Interface with the I/O switch</u>

When the call to ios_$attach is received by the I/O switch it in turn calls the attach entry of the target DIM. The latter must be coded to return a device pointer which can be used as an argument during subsequent function calls transmitted through the switch.

The I/O switch employs a special naming convention in designating the appropriate entry point in the DIM target. The switch is coded to accept a call of the form:

```
call ios_$attach (ionamel, type, etc.);
```

as signifying that there exists a transfer vector in the target DIM's linkage section whose elements point, via links, to the respective functional entry points in the target DIM. This transfer vector begins at type$type. For example, suppose one were writing a special DIM for the pdp7 computer regarded as a peripheral device. If the DIM is given the type name "pdp7", and if the call to attach were:

```
call ios_$attach ("input7", "pdp7", etc.....);
```

then the I/O switch assumes that the segment <pdp7> has an entry point located at <pdp7>|[pdp7] at which there is to be found a vector of transfer instructions, one to each supported functional entry point in the DIM.

Since the order in which these entry-point transfer instructions appear is fixed by still another system-wide convention for all DIMs*, the I/O switch is able to execute a call to the desired point (offset) in the DIM's transfer vector so as to reach the desired functional entry point. Thus, if the standard position for the read entry transfer is the third element in the vector, then a function call of the form

```
          call ios_$read ("input7", workspace, offset, etc.);
```

would find the I/O switch transferring into the third element in the transfer vector of <pdp7>, i.e., at <pdp7>|[pdp7] +2. Here it would be expected to find the assembled code for an instruction like

```
          tra     read-*,ic*
```

which transfers to the link named read whose contents, when snapped, will be an its pair pointing to where the DIM designer has placed the DIM's read function.

The DIM must be written in such a way as to return an error indication when there is an attempt by the switch to transfer to an entry point that corresponds to an unsupported I/O function. Here again, status reporting of attempts to utilize an unsupported function of a DIM is to be handled in convention-set way. The transfer vector element that would

* The particular order of these entries is to be found in a listing of the so-called "transfer vector template" which appears in the MSPM section that overviews the I/O system.

correspond to an unsupported function, e.g., ios_$seek for the typewriter DIM, is coded to send control to a small subroutine of fixed form* which fabricates the appropriate "entry-not-found" status word and returns to the I/O switch that called it.

By way of summary it is well to repeat that the DIM writer can only supply as entry points a subset of those which are in the "vocabulary" of the I/O switch (currently those twenty or so entry points given in the ios_ section of the MPM.)

8.5.3 The DIM's Interface with the GIM

With each read/write function call, the DIM fabricates Data Control Word lists (DCW lists) and passes these to the GIM for further modification and use. There is a series of calls outlined in the MSPM and in the MPM under "GIOC calls" which the DIM must make in order to give the GIM an opportunity to "set itself up" to receive and employ these DCW lists and to perform its various communication tasks. To make much sense of these calls the reader must acquaint himself with the GIOC hardware system reference manual. It is beyond the scope of this Guide to provide a sufficiently detailed explanation of the GIOC to make the DIM's calls to the GIM thoroughly meaningful.

* More details can be found in the same MSPM discussion that was cited in the preceding footnote.

A few general concepts can be conveyed here however. First, it should be understood that the GIM's work areas and I/O buffers must be wired down because they can only be referenced by the GIOC in an absolute addressing mode. The work areas are used for DCW lists that are compiled by the DIM and moved by the GIM from the virtual memory work areas designated by the DIM in its principal call on the GIM (at the entry point hcs_$list_change). These transmitted lists are transformed by the GIM before being activated so that the address fields in the DCWs are in loaded-and-ready-to-use absolute form. The work areas are also used for holding status words received from each channel and periodically copied over to corresponding work areas in the DIM's (virtual memory) data base.

The I/O buffers are also necessarily wired down and are referenced by the GIOC in absolute mode. The output buffer holds the actual data to be written out onto the channels. This data is copied by the GIM into its buffer from a corresponding buffer previously established and filled by the DIM. Similarly the GIM's input buffers are necessarily wired down and filled directly by action of the GIOC during input channel activity. This data is copied by the GIM from this buffer back onto the data array designated by the DIM. The GIM knows how to move data into its output buffer or out of its input buffer to/from the corresponding DIM data arrays because the call(s) made by the DIM to the GIM (at hcs_$list_change) provide the required pointers.

When the DIM is ready to ask the GIM to activate a DCW list, i.e., actually start the I/O (physically), the former makes a call requesting the

GIM to transmit a Channel Instruction Word for processing by the GIOC's so-called <u>connect</u> <u>channel</u>.

Prior to transmitting a DCW list the DIM must issue two preliminary or "set-up" calls to the GIM. The first call (to hcs_$assign) affords the user access to the desired channel. The DIM requests such access by supplying a symbolic name and receiving in return a uniquely-generated device index (17 bits) to be used in all subsequent I/O calls to the GIM for service on this channel. In the same call the DIM transmits an event channel name as an argument. This argument would be previously obtained from the interprocess communication facility (IPC) as a result of a call to ipc_$create_event_channel. It is by this event channel name that the process, when later blocked awaiting completion of a requested I/O operation, can be awakened by the GIM. The GIM acts as the interrupt handler for all I/O interrupts. It consults a privileged table which it maintains wherein are recorded the channel name, device index, process_id, event channel name 4-tuples. DIMs are normally programmed to call ipc_$block when processing reads or writes that are synchronous or when the read-ahead or write-behind buffers are filled.

Once the assign call has been successfully completed, the DIM, using the received device index as its key argument, can call the GIM again. This time the request is for the GIM to allocate sufficient wired down work space for the DCW list to be designated in subsequent calls that cause the DCW lists to be transmitted to that work space and assembled into "loaded" form.

The reader has hopefully been treated to the promised sketch of the DIM/GIM interface and is now on his own, ready for independent study, except for a few final remarks in the next section.

8.6 Final Remarks

A tour through the I/O system design would not be complete without observing that the I/O switch → DIM → GIM → GIOC chain is not absolutely mandatory as the only path allowed by Multics for I/O. A sophisticated subsystem designer will be the first to recognize that the I/O switch → DIM portion of this chain is strictly for user convenience. In principle, because the GIM may be called directly by the user, there is no reason why a designer could not bypass the I/O switch and DIM altogether for I/O to some special purpose or dedicated devices and communication or channels. In essence the subsystem would be written with code that achieves the equivalent of many I/O switch and DIM functions but calls the GIM directly using calls such as those suggested in Section 8.5.3. At all events the subsystem designer who chooses to travel this route should profit by a careful inspection of the I/O switch and DIM functions before launching into his own design that would allow a bypass of these modules.