A Multiprogramming Monitor for Small Machines

GARY D. HORNBUCKLE
University of California, Berkeley, California*

INT, a combination hardware/software monitor designed to control a wide variety of real-time input/output devices, is described. The simple hardware additions provide a uniform device to machine interface for such elements as keyboards, graphic input devices, and interval timers. The software relieves the user program from the details of input/output timing, buffering, and task scheduling and provides parallel processing capability. User programs communicate with the monitor through a small set of meta-instructions which consists mostly of machine-language subroutine calls.

1. Introduction

A wide variety of small, general-purpose machines are currently available primarily as process control computers. Although advertised as providing easy inexpensive interfacing of many devices, in reality the I/O multiplexing is very rudimentary, usually only one interrupt is available, and good monitor software is nonexistent. This paper is a report on a powerful combination hardware/software I/O monitor which resulted from an effort to interface a CRT graphic display to a small machine, the PDP-5/8,[1] which serves as a remote terminal to the Project Genie SDS-940 time-sharing system [1]. The first version began operation in August, 1965.

The handling of I/O from a variety of different word-length and speed devices can be easy or difficult depending upon hardware and software monitor. At the difficult extreme there is no monitor (users program I/O commands directly) and no (or very primitive) interrupt facility requiring many test/skip instructions just to decide what is happening. What is easiest, of course, depends upon the user's intent—some prefer FORTRAN-like "print" or "read" statements. However, such facilities require large overhead, and hence are out of the question for providing very flexible I/O with small supervisor overhead. The combination hardware/software monitor described here allows close control of the I/O device but prevents interference from attempted multiple use of the same device. It channels data to or from any size buffer in user memory, allows parallel processing, and controls task sequencing.

Although the monitor described here, called INT, is a particular implementation on a small general-purpose machine, a variety of devices are interfaced. The scheme is considered to be general, and the meta-instructions are thought to be a minimum set. Similar hardware and software could be easily implemented on a wide variety of machines.

2. Hardware

The hardware portion of INT is a strictly external unpluggable addition to the machine and provides a uniform interface for 16 devices, each of which can either transmit or accept data. To INT, a device is simply a unique binary signal (called the *attention flag*) which when true indicates that the device is seeking attention. The primary function of the hardware is to multiplex the 16 attention flags through the machine's single interrupt facility. Two versions of INT are presently operating, each interfacing a CRT display console, an ASR 33 teletype, a lightpen, an x, y-coordinate input device, a 5-key handset, a communications link to the 940 time-sharing system, and a memory tube/keyboard console [2].

2.1 *Scanner.* The original decision to add hardware was motivated by timing considerations. A software scan called "polling" of the device attention flags would have been too slow for the machine to keep up with the most demanding device (one word every 250μsec). On the other hand, complex priority interrupt hardware was unnecessary because all attached devices were to be asynchronous, i.e., they could wait almost indefinitely for acknowledgment of the attention flag without data loss. Hence, a hardware scanner (counter) was built which sequentially tests the 16 attention flags and, when finding one true, stops scanning (counting) until the machine has serviced the device. The scan count provides a 4-bit *device number* which can be used by the software as a table index.

2.2 *Arm/disarm.* One of several hardware changes occurred when the software necessary to keep track of the devices which should be instantaneously listened to and acknowledged grew excessively cumbersome. A hardware arm/disarm feature was added consisting of a separate flip-flop for each device to serve as a flag to selectively and instantaneously ignore its attention flag (hence called the *arm/disarm flag*). One of the 16 arm/disarm flags is *armed* when the machine wishes to listen to that device, or *disarmed* otherwise. Logically then, the condition for single machine interrupt I is given by

$$I = C_0 \cdot A_0 \cdot R_0 \vee C_1 \cdot A_1 \cdot R_1 \vee \cdots \vee C_{15} \cdot A_{15} \cdot R_{15}$$

where C_i is the device number or scan count, A_i the attention flag, and R_i the arm/disarm flag for device i. Since the set C_i are counter values, simultaneously occurring attention requests are considered one at a time.

Arm/disarm flags, hardware or software, provide an ability for the program concerned with a particular device to control that device without concern for others. In some

* Department of Electrical Engineering. This work was supported in part by the Advanced Research Projects Agency, Office of the Secretary of Defense, Washington, D. C.

[1] The Digital Equipment Corporation PDP-5 and PDP-8 are small, general-purpose computers with 4096 words of 12-bit memory.

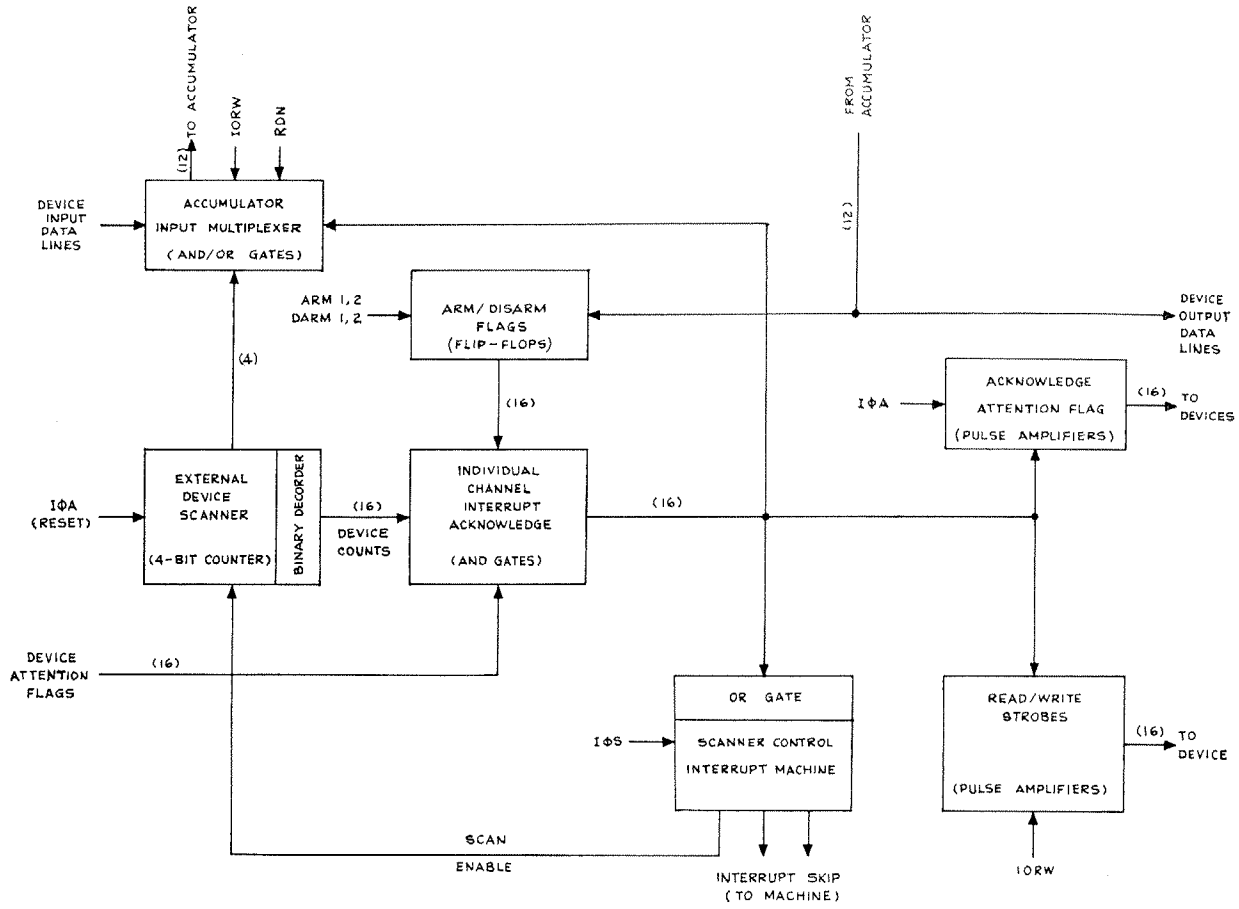

FIG. 1. Block diagram of INT hardware. The machine instructions IOA, IOS, IORW, etc., are also decoded in the INT hardware.

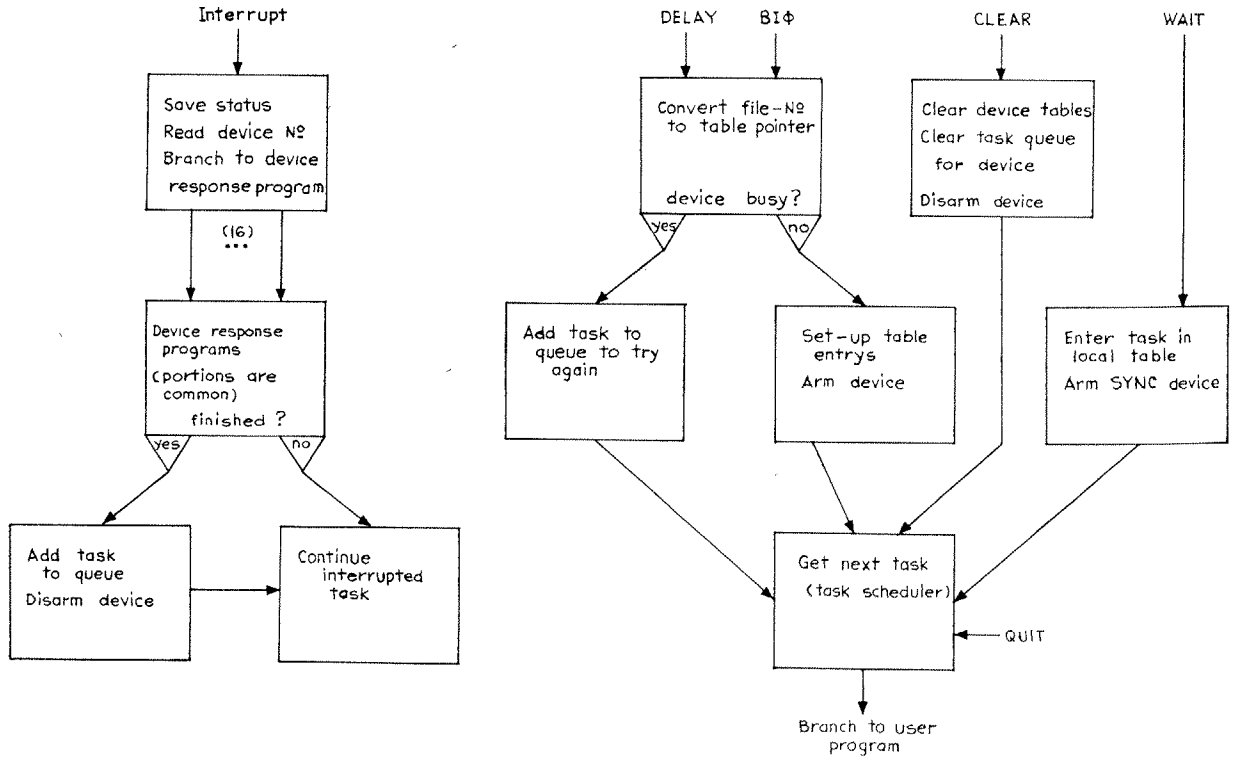

FIG. 2. Implementation of meta-instructions in INT. CONTINUE is a single instruction executed by user program which causes an interrupt. FORK simply adds a task to the queue and continues.

cases the attention flag should be ignored only temporarily. For example, it is characteristic of some output devices to demand attention when they can accept more data, and if the program has no more output the demand is ignored until such time as output is requested by the user program. The attention signal cannot be forgotten or control of the device is lost, hence the arm/disarm and attention flags must be given parallel significance. A device attention-interrupt is allowed to happen only if *both* flags are true. An unacceptable approach would be to cause an automatic attention reset if the device were disarmed. An equally bad situation would occur if the attention signal were a pulse which is not allowed to set the attention flag if it is disarmed. The arm/disarm flag state, however, is available to the device. This feature is used, for example, by the lightpen to avoid hanging up the display when a match occurs while the pen is disarmed.

By the time a second system was to be built, enough confidence had been gained in the monitor software to warrant a redesign of the hardware. The only significant innovation was the use of the device number to avoid having unique machine instructions to read/write each separate device. In addition, this latest version has a uniform 16-pin cable plug for each device as follows:

(a) 12 bits—data in or out
(b) read or write strobe from machine
(c) attention flag from device
(d) arm/disarm flag from machine
(e) acknowledge attention flag from machine.

2.3. *Implementation Details.* Figure 1 is a block diagram of the existing monitor hardware. The external device scanner is a 4-bit, 1-megacycle binary counter whose output is decoded into 16 possible device counts. Each device count is "anded" with a device attention flag and arm/disarm flag. When all three become simultaneously true for a particular device, the scanner is inhibited from counting further and the single machine interrupt is caused. The machine then reads the 4-bit device number into the accumulator and uses the number to branch to the program within INT responsible for the device. Within this *device response program*, data is read in or out and the appropriate gating signals are sent to the device, which then clears its attention flag. The scanner is also reset and starts at device 0, which provides a sort of priority in the case of simultaneously occurring interrupts.

With the current software, attention requests can be acknowledged every 36μsec with a PDP-8 or 222μsec with a PDP-5.[2] Arbitrarily many data words can be exchanged between the device and machine at a given attention request, although only enough multiplexing gates are included in the hardware of INT for up to fourteen 12-bit words, which can be allocated in any way to the 16 devices. Several of the devices currently attached have no data associated with them, such as a panic button.

The special instructions for INT are:

(a) IOS —sense attention flag
(b) IORW —read/write data
(c) IOA —acknowledge attention flag
(d) ARM1,2 —arm
(e) DARM1,2—disarm
(f) RDN —read device number

IOS is used by disarming all devices except the one to be tested; then IOS will skip if the attention flag of the sensed device is true. IORW causes reading or writing; the gating strobe is sent to the device in either case to enable device multiplexing of the data lines for more than two words exchanged per attention request. A special provision is built in to allow a few devices to input 24-bits (two words) of data. IOA generates a signal which the device uses to clear its attention flag. The PDP-5/8 allow any combination of IOS, IOA, and IORW to be executed simultaneously. The arm/disarm instructions selectively arm or disarm devices based upon a mask in the accumulator. If the bit corresponding to a particular device is off, the arm/disarm flag of that device is unaffected. Two arm and disarm instructions are necessary because of the 12-bit accumulator and 16 devices. RDN actually reads the device number and the appropriate bits to make up an instruction which when executed causes a branch through a 16-word transfer-vector.

3. Software

The software monitor provides parallel process capability which allows overlapped I/O. Programs to handle any combination of the attached devices simultaneously are easy to construct because of the simple yet powerful meta-language which relieves the user program from the details of timing, buffering, and scheduling considerations. The meta-instructions BIO (block input/output), FORK, QUIT, WAIT, CONTINUE, CLEAR, and DELAY are described below and illustrated in Figure 2.

The user normally wishes to initiate a macro-sized operation such as read/write a block of data, and, while the I/O is being performed, proceed on another task, perhaps to initiate operations for other devices. In the simplest case, the user may wish to listen for input from several devices simultaneously. To allow this, INT has a *task queue* for handling multiple tasks and uses the INT hardware for overlapping I/O (device attention requests) with processing.

The usual situation is that a user program (also called *task* or *process*) is being executed at the same time attention is being sought by devices wishing I/O. A machine interrupt causes temporary suspension of the user process while the I/O is performed by the *device response* program within INT, after which the process is allowed to continue. I/O may be going on simultaneously with any of the armed devices, and any one attention cycle takes only

[2] These times are based upon a 1.5μsec and 6.0μsec cycle time for PDP-8 and 5, respectively and a single 12-bit word input or output.

a small amount of time from the interrupted process (see the times given above).

When a user process is finished or "hung-up" (temporarily suspended) to await a requested I/O operation to complete, a new task is taken from the task queue and started up. New tasks get on the queue by an explicit user request (parallel processes) and by a device response program which has detected an I/O completion (previously suspended tasks). The very simplest task scheduling is used—first come, first served; a task is allowed to go to completion or an I/O hang-up. A priority scheduling scheme would cause no special difficulty but was considered unnecessary since the user programs are considered to be well behaved; future results may alter this policy. Such a case might occur if two or more user programs were operating simultaneously, one of them undebugged, for instance. In no cases do user programs repeatedly test or poll the monitor or execute the I/O or arm/disarm instructions described above.

3.1 *Meta-Instructions*

BIO—*Block I/O.* The single statement for initiating I/O is BIO (d, s, e), where d is a device *file number* (not the device number used by the hardware) which serves to uniquely define the I/O device and to address certain tables within INT. The number of words to be transmitted or received is identified by the block starting address s and ending address e. A word or character instruction is not provided because little (one word) would be saved in the calling sequence, extra monitor program would be necessary, and because word operations are possible anyway (with $s = e$). Characters are not expected to be packed in this application, so that character and word operations are identical.

Certain devices such as the panic button have no data associated with them. In these cases, the number of words $e-s+1$ indicates the number of times the device seeks attention, such as number of panic button depressions, rather than number of data words. In those cases where devices have more data than one machine word per attention request, e is the last machine word in the buffer and $e-s+1$ is the number of words input or output, rather than number of attention traps.

In all cases, the user's program is hung-up at a BIO; that is, another task (if one exists) is initiated and the hung-up task is continued *only* when the operation requested is completed. Hang-up can be avoided, if so desired, by creating a parallel task as follows:

FORK(p)	create parallel process p
BIO(d, s, e)	do I/O
QUIT	quit when I/C complete

p: parallel process to avoid BIO hang-up

Attempts to perform I/O on busy devices are remembered (until the task queue overflows) by adding to the bottom of the queue a task which causes the BIO for the busy device to be repeated periodically until the device becomes free. However, the order in which the requests are completed has no relation to the order in which requests are made. In cases where order is essential, the hang-up at BIO avoids multiple hanging requests. On the other hand, the use of parallel processes can allow a process to communicate with a given device without having to concern itself with whatever other process is using it. This feature is useful if several processes are using a continuously changing input data variable whose values can be more or less randomly distributed between processes. For example, two separate processes could use RAND tablet coordinates [3] which appear to each process to occur at one half the normal rate. Also, several independent processes can output simultaneously to the same device if, for examples each BIO were to write a block consisting of the data plus information sufficient to uniquely identify the data.

FORK. Parallel processes are created by the instruction FORK(p) where p, a machine address, is simply added to the task queue and the process causing the FORK is continued. From that point on, the processes have identical status—no structure or hierarchy is remembered. The FORK was a byproduct of the implementation of BIO but provides much of the power of INT.

QUIT. A process is terminated by the QUIT instruction, which is simply a transfer to the task scheduler.

WAIT, CONTINUE. Intended to be used in pairs, WAIT(i) and CONTINUE(i) provide a means for synchronizing processes. The index i simply provides a number of pairs (5 in INT).

WAIT(i) executed by some process A, say, causes A to be suspended until another process, say B, executes a CONTINUE(i). Process B proceeds and A is added to the task queue and continued later.

CONTINUE is in reality an instruction which sets a device attention flag (called the *SYNC* device) and thus causes an interrupt. The *SYNC* response program then adds the corresponding suspended process (which has executed a WAIT) to the task queue. The order of occurrence of WAIT/CONTINUE pair is of no concern; WAIT causes the SYNC device to be armed, CONTINUE sets the attention flag. A *join*, complement of FORK, is implemented with a WAIT/CONTINUE pair; the CONTINUE is followed by a QUIT.

CLEAR. Another means of process communication is the ability of a process to CLEAR(d) all knowledge the monitor has about the state of device d. All the tables are cleared, the device interrupt disarmed, and all tasks in the queue relating to the device are erased. One of the device interrupts (panic button) causes INT to clear all other devices as a matter of course in its attempt to recover from a *panic*. In the current system, a *disaster* (a panic which has destroyed the panic recovery routines) is recoverable only through a bootstrap loader.

DELAY. An interval timer capability is provided by DELAY(t), which causes the process to be temporarily

suspended, or delayed, for t units of time, where a unit is approximately 30msec.

As can be seen from the meta-instructions available, INT is a small rudimentary time-sharing system. Even memory-swapping could be performed through the communications link to the 940. However, no plans are currently under way to exploit these possibilities. The parallel process capability is viewed here as a means for writing nontrivial control programs for a wide variety of I/O devices rather than a way to run several computations in parallel, although the latter is clearly possible.

3.2 *Implementation Details.* The software may be broken down as follows:

Meta-instruction subroutines
Task scheduler
Task queue
Device response programs
Device tables

Initially all devices are disarmed and the task queue empty when the user program starts. Assume that at some point the user wishes to do I/O and executes at location p a BIO(d, s, e). The BIO subroutine first checks to see if the device d is busy, and if not, sets up the first 3 words of the 4-word device table entry for device d as follows:

Word	*Meaning*	*Initial value*
1	buffer pointer	s
2	words remaining	$e - s + 1$
3	suspended task location	$p + 1$
4	arm/disarm mask	(unique for device d)

Device d then becomes busy by definition since entry 2 is nonzero. As mentioned above, d is actually the device *file number* and is the relative location of the corresponding 4-word block within the device tables. Finally, BIO uses entry 4 to arm the device without affecting or knowing anything about the state of other devices.

Rather than return to the user program at location $p+1$, BIO transfers control to the task scheduler which searches the task queue for the next active task and starts it if one exists. Otherwise the scheduler simply loops waiting for a task.

Meanwhile, interrupts occur for the armed devices. Initially at each interrupt, an RDN instruction is executed which reads the 4-bit device number and uses it to transfer to a unique device response program. The response program then reads or transmits a word into or out of the memory cell whose location is the first entry of the appropriate block of the device tables, increments that entry, and decrements entry 2, the words remaining. If the result is zero, the block I/O has been completed and entry 3, suspended task location, is added to the task queue and the device is disarmed. With the exception of the panic button, interrupt response programs always continue the interrupted task. Most of the device response programs are common since the pointers and counters are identically arranged for all devices in the tables.

The task queue is simply a 16-word ring buffer operated in a first-in/first-out fashion. A word contains the memory address of active tasks; initial values of the machine status registers are assumed 0 for a task. If a process were not allowed to go to completion but arbitrarily suspended to run another process, the machine status would have to be saved. Two pointers define the beginning and end of the queue which is empty if beginning equals end. New tasks are added to the end of the queue by FORK. QUIT simply invokes the task scheduler.

WAIT and CONTINUE simply make use of one of the interrupts and a small table to provide for several WAIT/-CONTINUE pairs. CLEAR does the obvious thing for the device referenced.

The current implementation is approximately 300 words long, 200 of which are instructions with the rest data or tables; this seems remarkably small compared to other general-purpose, multiple-task monitors.

4. Example

Figure 3 illustrates a variety of INT's features in a program for a graphic display. An object is pointed to, by a RAND tablet stylus [3] for example, until the object flashes (slowly blinks). Then a button (the tablet stylus) is pressed which causes an identification of the object to be sent to the 940. Flashing of an accidentally selected object is stopped if the button is not pressed within two seconds. Otherwise the flashing is erased when the 940 acknowledges receipt of the message.

The first fork establishes a process A to continuously listen to the 940. The only function shown is to erase the flashing of a displayed object caused in F. The second fork establishes separate processes for listening to the RAND tablet input (C and D) and listening for a "match," a lightpen-like input from the pen (B, E, and F). Process C reads and processes pen coordinates and informs process E that the pen is down (i.e., the pen switch is closed). D waits for the pen to lift (pen switch to open), sends the coordinates to the 940, and restarts C and D. B waits for a match to occur, and then starts E and F so that the object pointed to can be flashed whether or not the pen is down. E is executed only if the pen goes down, and then clears the DELAY started in F and sends to the 940 the information about the object being pointed to. Process F causes the object to flash and starts a 2-second DELAY. Hence, if the pen switch fails to close within 2 seconds following a match, the flashing is locally ceased. Otherwise, the flashing continues until the 940 responds (in A).

5. Conclusions

No attempt was made at the time of the original design to pattern INT after other multiprogramming systems [4, 5], although some similarity exists. A FORK meta-instruction exists in most such systems with the parallel process structure remembered by the monitor, thus providing an ability for a process to transmit control information to its "controlling" process (that which created it)

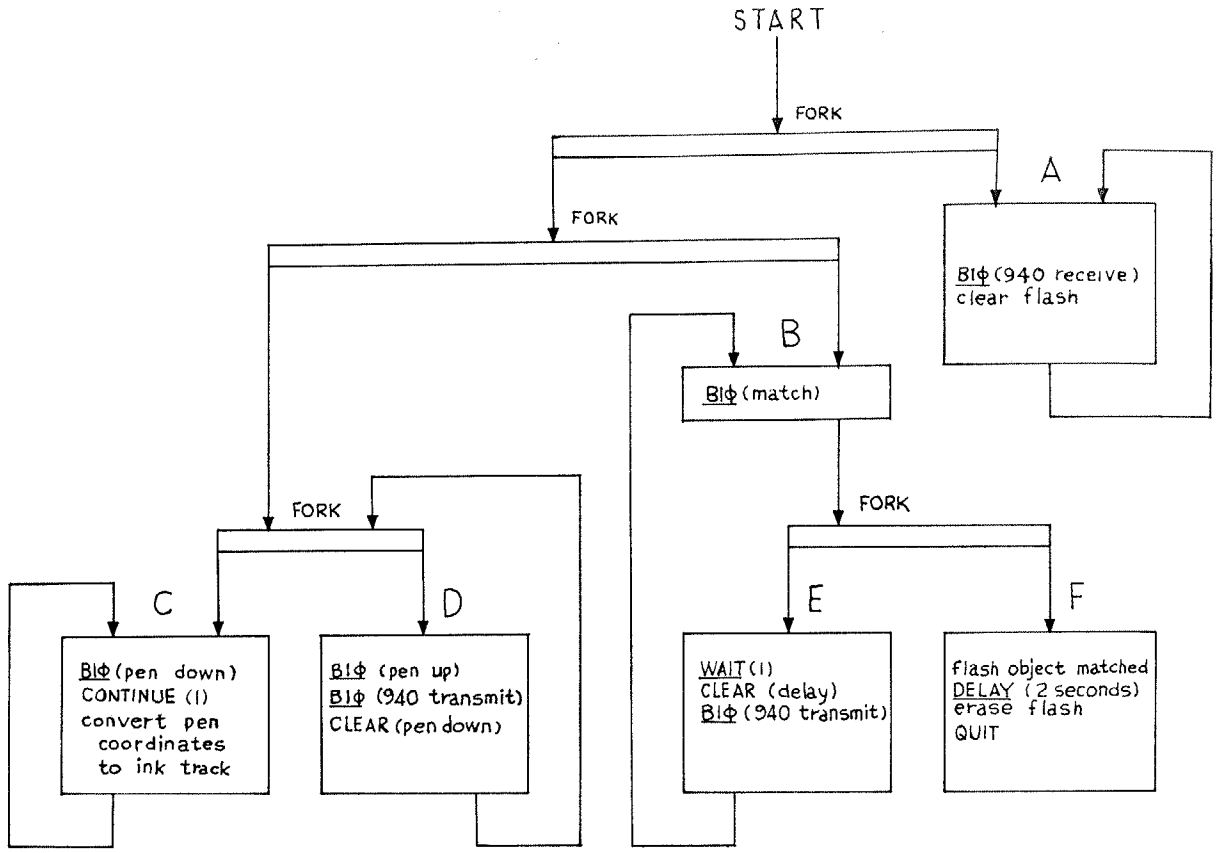

FIG. 3. Structure of user program for a graphic display example. Parallel horizontal lines indicate parallel processes started by a FORK instruction. Underlined meta-instructions cause temporary hang-up of the process.

without the explicit knowledge required by a WAIT/-CONTINUE pair. Also, with INT a process cannot (legitimately) obtain the instantaneous status of another process or for that matter an I/O device. In the latter case a SENSE(d) meta-instruction could be added to test skip if device d is busy. File handling (naming, directories, I/O, etc.) could be added in an "executive" which uses INT to control the I/O. The problems of process interference and memory swapping have been given little consideration because of the purpose for which INT was created.

However, as a monitor for I/O control within a remote graphic terminal of a powerful time-shared computer, INT has performed admirably. Suggestions for improvement are given careful consideration, but it has generally been the case that the amount of monitor software necessary for the addition of a new feature has been greater than the amount of software necessary for the user to implement the feature himself.

Acknowledgments. The author thanks William Teo and Barry Borgerson for their efforts in redesigning the original hardware to provide a simpler, more general package, and also Ralph Love, who as the first major user contributed significantly in a thorough shakedown of the software.

RECEIVED OCTOBER, 1966; REVISED DECEMBER, 1966

REFERENCES

1. LICHTENBERGER, W., AND PIRTLE, M. W. A facility for experimentation in man-machine interaction. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1, 1965, pp. 589–598.
2. HORNBUCKLE, G. D. GO, Genie graphical input/output system. Doc. No. 30.80.10, Project Genie, Dept. of Electrical Engineering, U. of California, Berkeley, Calif., July 1, 1966.
3. DAVIS, M. R., AND ELLIS, T. O. The RAND tablet: a man-machine communication device. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, Part 1, 1964, pp. 325–332.
4. LAMPSON, B. W., LICHTENBERGER, W. W., AND PIRTLE, M. W. A user machine in a time-sharing system. *Proc. IEEE 54*, 12 (Dec. 1966).
5. DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computation. *Comm. ACM 9*, 3 (March 1966), 143–155.