REFERENCES

[1] R. W. Hamming, *Numerical Methods for Scientists and Engineers*. New York: McGraw-Hill, 1962.
[2] F. B. Hildebrand, *Introduction to Numerical Analysis*. New York: McGraw-Hill, 1956.
[3] P. Henrici, *Discrete Variable Methods in Ordinary Differential Equations*. New York: Wiley, 1962.
[4] J. R. Ragazzini and G. F. Franklin, *Sampled-Data Control Systems*. New York: McGraw-Hill, 1958.
[5] J. T. Tou, *Digital and Sampled-Data Control Systems*. New York: McGraw-Hill, 1959.
[6] E. I. Jury, *Theory and Application of the z-Transform Method*. New York: Wiley, 1963.
[7] "Numerical techniques for real-time digital flight simulation," *IBM Manual*, E20-0029-1, 1964.
[8] W. D. Fryer and W. C. Schultz, "A survey of methods for digital simulation of control systems," Cornell Aeronautical Lab., Buffalo, N. Y., Rept. XA-1681-F-1, July 1964.
[9] M. Blum, "Recursion formulas for growing memory digital filters," *IRE Trans. on Information Theory*, vol. IT-4, pp. 24–30, March 1958.
[10] A. Tustin, "A method of analysing the behaviour of linear systems in terms of time series," *JIEE*, vol. 94, pt. II-A, May 1947.
[11] A. Madwed, "Number series method of solving linear and nonlinear differential equations," M.I.T. Instrumentation Lab., Cambridge, Mass., Rept. 6445-T-26, April 1950.
[12] J. G. Truxal, "Numerical analysis for network design," *IRE Trans. on Circuit Theory*, vol. CT-1, pp. 49–60, September 1954.
[13] R. Boxer and S. Thaler, "A simplified method of solving linear and nonlinear system," *Proc. IRE*, vol. 44, pp. 89–101, January 1956.
[14] R. Boxer, "A note on numerical transform calculus," *Proc. IRE*, vol. 45, pp. 1401–1406, October 1957.
[15] W. H. Anderson, R. B. Ball, and J. R. Voss, "A numerical method for solving differential equations on digital computers," *J. ACM*, vol. 7, pp. 61–68, January 1960.
[16] M. C. Fowler, "A new numerical method for simulation," *Simulation*, vol. 4, pp. 324–330, May 1965.
[17] J. M. Hurt, "New difference equation technique for solving nonlinear differential equations," *1964 AFIPS Conf. Proc.*, vol. 25, pp. 169–179.
[18] A. P. Sage and R. W. Burt, "Optimum design and error analysis of digital integrators for discrete system simulation," *1965 AFIPS Conf. Proc.*, vol. 27, pt. 1, pp. 903–914.
[19] A. P. Sage, "A technique for the real-time digital simulation of nonlinear control processes," *1966 Proc. IEEE Region 3 Conf.*
[20] J. R. Tou, *Modern Control Theory*. New York: McGraw-Hill, 1964.
[21] R. Bellman and R. Kalaba, *Quasilinearization and Nonlinear Boundary-Value Problems*. New York: American Elsevier, 1965.
[22] R. McGill and P. Kenneth, "Solution of variational problems by means of a generalized Newton-Raphson operator," *AIAA*, vol. 2, pp. 1761–1766, October 1964.
[23] W. Hurewicz, "Filters and servo systems with pulsed data," *Theory of Servomechanisms*, M.I.T. Rad. Lab. Ser., vol. 25. New York: McGraw-Hill, 1947.
[24] J. E. Gibson, *Nonlinear Automatic Control*. New York: McGraw-Hill, 1963, pp. 94–159.
[25] J. M. Salzer, "Frequency analysis of digital computers operating in real time," *Proc. IRE*, vol. 40, pp. 457–466, February 1954.
[26] H. Freeman, *Discrete-Time Systems*. New York: Wiley, 1965.
[27] R. E. Kalman and J. E. Bertram, "A unified approach to the theory of sampling systems," *J. Franklin Inst.*, vol. 267, pp. 405–436, May 1959.
[28] R. K. Adams, "Digital computer analysis of closed-loop systems using the number series approach," *Trans. AIEE (Applications and Industry)*, vol. 80, pt. II, pp. 370–378, January 1961.
[29] M. Cuénod, "Methods de calcul a l'aide de suites," Lausanne: Imprimerie de la Concorde, 1955.
[30] J. M. McCormick and M. G. Salvadori, *Numerical Methods in Fortran*. Englewood Cliffs, N. J.: Prentice-Hall, 1964.

A Method of Processor Selection for Interrupt Handling in a Multiprocessor System

R. J. GOUNTANIS AND N. L. VISS

Abstract—**A method of assigning external interrupts to processors in a multiprocessor system is described. Features of a multilevel priority interrupt system are incorporated into a hardware component called the *Interrupt Directory*. The directory selects the most appropriate processor for servicing the interruption at the time the event occurs. The "appropriateness" for interruption is based on the priority level of a processor's current task, thus providing dynamic priority allocation of tasks. Queueing of interrupts is also provided. The arrangement described in this paper simplifies and increases the effectiveness of executive control programs. Implications of the Interrupt Directory on reliability and "fail-soft" operation are also discussed.**

1. INTRODUCTION

INTERRUPT handling presents common problems in differing system types. An excellent summary of interrupt features and problems has been presented by Borgers[1] and supplemented by Bennet[2] for what is evidently a single computer case. A multisystem[3] has the same problems, plus additional ones arising mainly from having more than one processor in the system.

One problem peculiar to multisystems is what to do with

Manuscript received June 30, 1966; revised August 22, 1966.

The authors are with the Univac Division of Sperry Rand Corporation, St. Paul, Minn.

[1] E. R. Borgers, "Characteristics of priority interrupts," *Datamation*, vol. 11, pp. 31–34, June 1965.

[2] J. G. Bennet, "Letters," *Datamation*, vol. 11, p. 13, October 1965.

[3] By a multisystem is meant a system consisting of two or more central processing units that can communicate without manual intervention. (See G. A. Blaauw, "IBM System/360 multisystem organization," *1965 IEEE Intern'l Conv. Rec.*, pt. 3, p. 227.) The term encompasses both multicomputer and multiprocessor systems. A multicomputer system is a multisystem in which central processors have dedicated "private" memories and can communicate with each other via I/O channels. Multiprocessor denotes a multisystem wherein the central processors intercommunicate chiefly by access to a shared memory.

I/O interrupt requests (those interrupts which may be processed in parallel with current operations, as opposed to processor-related interrupts which are handled in series with the routine which triggers them). Several schemes now in use are:

1) send the interrupt to the processor which initiated the I/O action originally
2) have one processor declared as the one and only processor to handle I/O interrupts
3) provide hardware registers whose contents define which interrupts go to what processors.

Note that all such fixed or associative routing schemes set up the routing prior to the time of interrupt. The major disadvantage of this technique is that the associated processor may be in some high-priority procedure at the time of interrupt, and should not be interrupted. Another significant shortcoming is the difficulty of rerouting the interrupt if the preselected processor fails.

The method described in this paper involves making all processors in the system eligible for interruption by *any* I/O interrupt. In fact, this equal stature of all processors for interrupt purposes is merely a logical extension of a concept that is basic in the multiprocessor system being discussed. This system embodies totally equivalent processors, plus separate and independent I/O controllers, all sharing a common memory. In this arrangement, there is no fixed assignment of processors to any task. Even the executive program is free-floating.[4] As any processor completes a task, that same processor picks up the supervisory task assignment routine and finds a new job.

For optimum priority response, every processor in the system should be executing the most important tasks as determined by the environment.[1] Task assignment and processor selection for interrupt processing are similar problems in this regard. Both must be implemented so as to complement each other in keeping all facilities busy on the activities with the highest urgency. In this "peer" arrangement, no processor is indispensable, in contrast to master-slave systems having one processor dedicated to task assignment, or to interrupt recognition.

Whereas task assignment is handled in the executive software, processor selection for interrupt handling is implemented in a relatively simple hardware unit called the Interrupt Directory. This unit can execute the processor selection algorithm more rapidly than could be expected of the software needed to do the same job. Moreover, such software would have to run in one of the processors, which has many more components than the directory—an important reliability consideration.

The selection algorithm uses variable software-defined priority levels as parameters. The features of a multilevel priority interrupt scheme are used.[1] For example, whether or not an individual processor receives an interrupt is contingent on whether the interruption event is of greater urgency than the current processor operation. But the method described in this paper also considers the current operations in *all* the processors in the system. Because the notion of priority can be generalized to include the "priority" of idle processors, the selection method always routes I/O interrupts to idle processors in preference to busy ones.

2. System Description

2.1 System Organization

The multiprocessing system assumed for this discussion is a configuration in which two or more computing units (processors) and one or more input/output subsystems can communicate with a common modular memory. All input/output subsystems are independent of the processors and may operate in parallel with the processors (see Fig. 1).

The processor is the basic computational element within the system. All processors have identical capabilities for processing instructions, accessing memory, and communicating with the I/O subsystem. Initiation of I/O is performed by the processor.

Though the processors initiate I/O, the control and sustaining of the operation(s) is handled by the I/O controller. An I/O controller is a small processor capable of executing special instructions; an initiating processor "writes" a "program" made up of these instructions, and then orders the controller to execute the instruction sequence. Each I/O controller may supervise several I/O operations concurrently (one for each channel to external subsystems). Moreover, each I/O controller receives all interrupt signals from the external subsystems associated with it. An I/O controller can also generate interrupt signals based on conditions internal to the controller.

The memory may be one common module or it may be composed of several modules. In either case, the memory module(s) is able to communicate with every processor and input/output controller in the system. A system so connected is referred to as a fully distributed system. The memory module responds to read and write requests from both processors and I/O controllers. Each memory module has the ability to resolve queues which arise from simultaneous processor and/or I/O controller service requests.

The system control of the multiprocessor system is based on the concept of a *pool of anonymous processors*. This approach to controlling a multiprocessor system avoids *any* basic distinction between the separate processors. This method is in contradistinction to a master-slave system wherein the master processor assigns functions (task assignment, interrupt control, task execution, etc.) to itself and to the slave processors.

The supervisory or executive program is maintained in the shared storage (main memory) and is executed by any or all processors as required within the system.[4] Only the executive program, not a processor, can be considered as *the master!* Upon completing a task (coding which may be executed in parallel with other coding) a processor returns to the executive program, executes the task assign-

[4] J. J. Pariser, "Multiprocessing with floating executive control," *1965 IEEE Internat'l Conv. Rec.*, pt. 3, pp. 266–275.

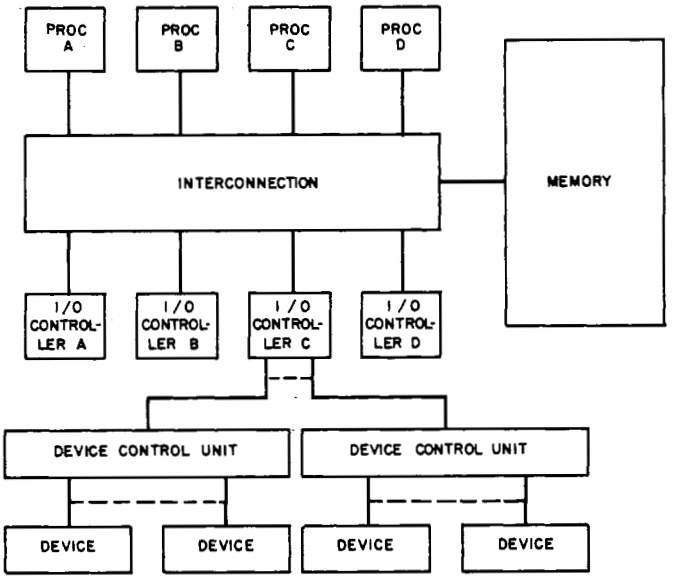

Fig. 1. Multiprocessor system.

ment routine, assigns *itself* a new task, and exits from the executive program.

2.2 Processor Selection Problem

2.2.1 Interrupt Categories: In the multiprocessor system described above, as in any system, we can distinguish four major categories of interrupts—traps, time-out clocks, machine errors, and external or I/O interrupts. These categories form two groups: those which are associated with a particular processor (dependent interrupts), and those which are independent of a processor. Traps, time-out clocks, and machine errors are directly associated with a processor and/or the current operating task. Any interrupt in one of these dependent categories halts the current task at the point where the contingency occurred and the interrupt is processed prior to completing the current task:

1) in order to retain the same logical sequence in executing the task (as with traps), or
2) to establish if the current task should be continued or terminated (as with time-out clocks), or
3) because the parameters involved are contained within the processor and task being executed (as with machine errors).

The input/output interrupts are characterized by occurring independent of the task in execution. The occurrence of this interrupt has no dependency upon a specific processor but is related only to the task associated with controlling the I/O subsystem which generated the interrupt. Essentially, an I/O interrupt is a signal to start processing an additional path of instructions. However, it is not necessary to capture a busy processor to handle the I/O interrupt unless a) there are not sufficient processors to work on all the available paths, and b) the I/O interrupt path is considered of higher priority.

2.2.2 Interrupting for Optimum Priority Response: In a multiprocessor system having m processors, optimum priority response implies that, out of n tasks (including interrupt processing) that *could* be running at a given time, the system is executing the m tasks with highest priorities. If each potential task has a weighting factor, then the tasks (including interrupt processing) must be selected so that the sum of the weights of the m tasks being run is at a maximum. We refer to this sum of m weighting factors (or priority values) as the *total system priority*.

Part of the job of maximizing total system priority is in the task control portion of the executive software. However, input-output interrupts cannot be included in the normal task control mechanism. This becomes apparent if we consider that system reaction to an I/O interrupt request must still be such that *if* a processor were to be interrupted, there must be a net gain in total system priority resulting therefrom. If no gain would result, then *no* processor should be interrupted, *not even* to evaluate a potential gain or loss in total system priority.

As a result, in order to insure optimum priority response when an I/O interrupt request occurs, there must be a means of predicting the effect of the interrupt on the total system priority, a means that does not require a processor for its implementation. Furthermore, if a net gain is foreseen, the interrupting should be done so as to obtain the *largest* net gain. Only then is the system priority truly optimized.

3. Interrupt Directory and Interrupt Request Handling

3.1 Interruptibility Index and Interrupt Priority

The optimization process utilizes two parameters for determining the net gain in system priority. These parameters are called the Interruptibility Index and the Interrupt Priority. To provide flexibility in their use, they are assignable under program control.

Associated with each task in every processor is an Interruptibility Index (II). This is a number which is a measure of the relative urgency or priority of the task during its execution by a processor. This number or index is held in a hardware register in the processor. The index can assume any value, in order of increasing urgency, say from zero to fifteen. An II of zero represents a completely interruptible processor and can be loosely associated with the idle state of a processor. The value of the II is provided for each task in process, independent of any other tasks in process, by instructions available only to supervisory routines.

An Interrupt Priority (IP) number is associated with each possible external interrupt condition. This number is a measure of the relative response time required by the I/O interrupt with respect to the priority of other interrupts and the Interruptibility Indices of current processor tasks. This priority number may also assume any value, in order of increasing urgency, say from zero to fifteen. Should a device require a short response time, a high IP value is assigned to that I/O operation. If the interrupt may be deferred, indicating a very low urgency of response, the external operation may be assigned an IP of zero. In such a case, the interrupt status word would be stored without generating an

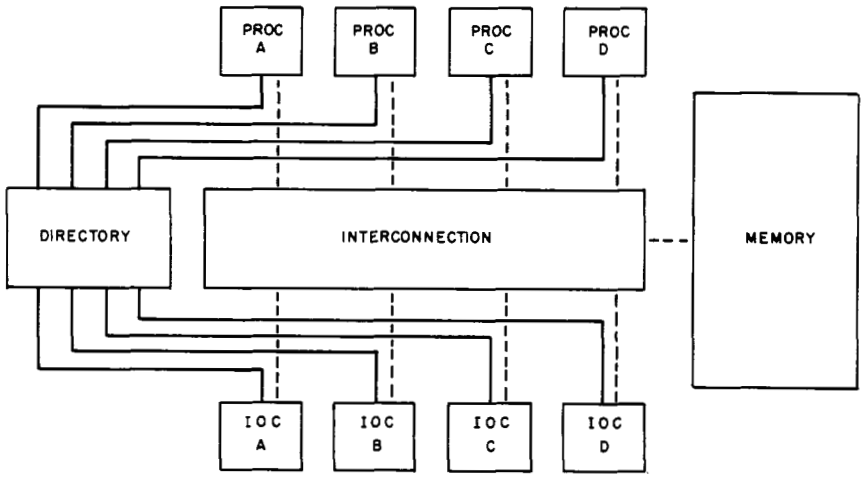

Fig. 2. Multiprocessor system with Interrupt Directory.
——— interrupt control lines. - - - - - data paths.

interrupt signal. This feature is referred to as the "queueing of interrupts." The value of the IP is provided for each I/O operation by the supervisory input/output routines. Since the values of II and IP for tasks and I/O operations are defined under program control, priority allocation is truly dynamic.

3.2 Directory Functions

The algorithm for processor and interrupt selection is implemented in a hardware unit called the Interrupt Directory —see Fig. 2. The unit is a logic network about as complex as a simple parallel adder. This device a) automatically assigns processors to interrupt-related tasks, and b) selects the external interrupt to be processed. The assignment is performed on the basis of processor status *at the time* of interrupt.

In general, the Interrupt Directory accepts as inputs the Interruptibility Index from each processor task and the Interrupt Priority from each I/O subsystem. The Directory then performs three basic operations:

1) it finds the processor with the lowest index
2) it chooses the interrupt request with the highest priority
3) it interrogates the selected processor *if* the interrupt priority value is greater than the index of the processor.

3.3 Overall Interrupt Request Handling

The Interrupt Directory forms a part of the overall interrupt request handling system. The essential components of this system are shown in block form in Fig. 3. Their operation is depicted in flow chart form in Fig. 4.

All I/O interrupt requests are channeled through, or are initiated by, the I/O controllers. As each request is generated, the pertinent I/O controller stores the related interrupt status word in a buffer in main memory—see Fig. 5. Each I/O controller is assigned its own buffer, consisting of 16 lists, one for each IP level. The status word is stored in the appropriate list (according to the IP of the request) in the buffer for the related I/O controller. This storing always occurs, and should there be several interrupts with the same IP from the same I/O controller, the status words are queued in the list. The list can be of any length required by the application; a typical variation may be from 0 to 1024 words.

A single IP buffer list for all I/O controllers could be used rather than one list per I/O controller as mentioned. However, the indexing control for a common list complicates the hardware of the I/O controller, causing longer reaction time to the interrupt. Furthermore, the possibility of memory queueing is greatly increased.

Each I/O controller monitors all interrupt requests from its associated I/O subsystems and presents to the Interrupt Directory the highest IP it finds. The Interrupt Directory in turn monitors all I/O controllers, looking for the highest IP. As the Directory performs the scan, each I/O controller is instructed to hold its IP lines static while being sampled by the Directory. Once the Directory determines the I/O controller with highest submitted IP, it releases the other I/O controllers, but continues to command a lockout in the selected I/O controller. If several I/O controllers have the same highest IP, one of these is selected based on its physical connection to the system according to some ordering scheme.

The Directory then performs the II/IP comparison already described. If several processors have the same II, a processor is selected on the basis of its physical connection to the system according to some ordering scheme. The Directory forwards the interrupt request, the Interrupt Priority and the I/O controller number to the selected processor. This information is sufficient to define the table and list containing the related status words. Because of the asynchronous relationship between processor and Directory, the final decision to interrupt is made by the processor itself. The processor could have changed its II in the brief time interval between processor selection and submission of the interrupt request to the processor. The processor makes this final decision by comparing the submitted IP with its own current II and returns an acceptance or rejection to the Directory.

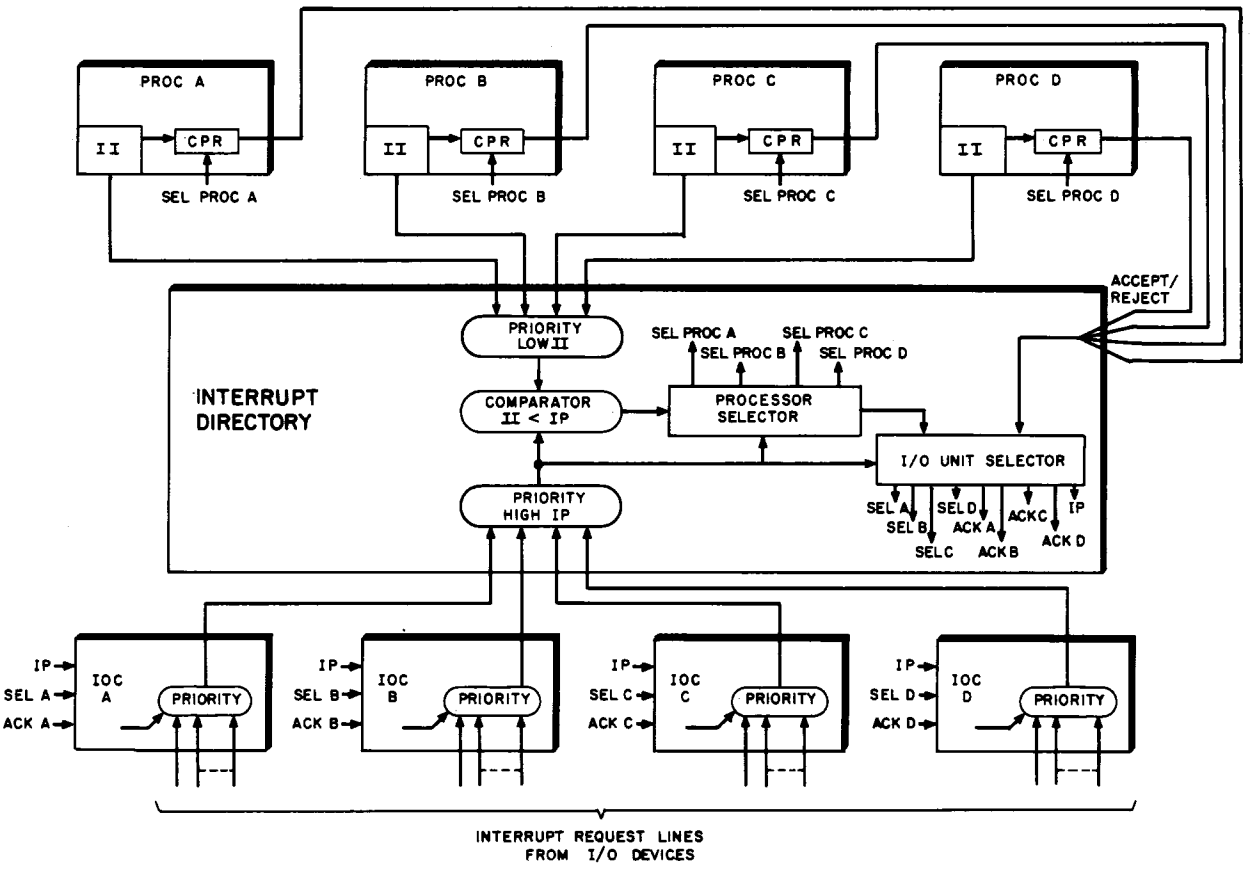

Fig. 3. Functional block diagram of interrupt request handling.

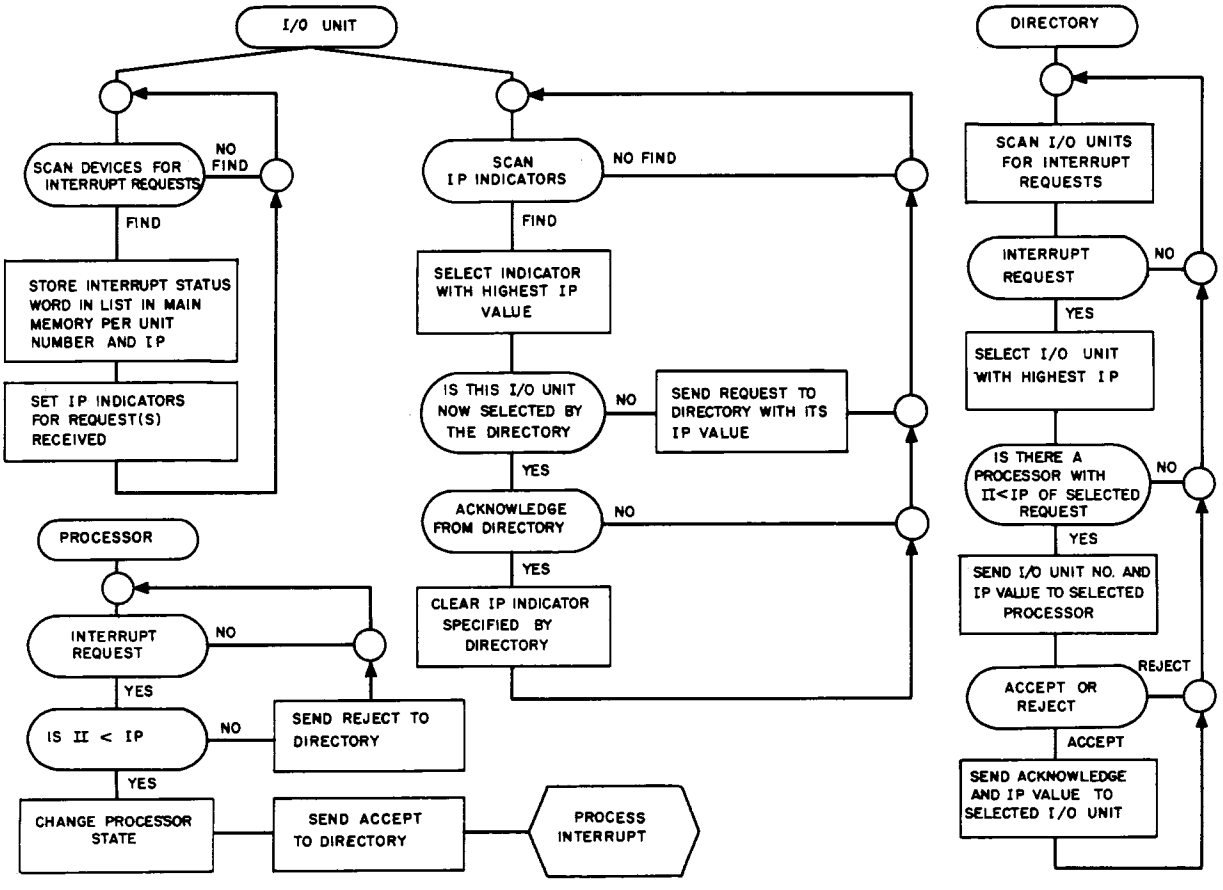

Fig. 4. Flow chart of IOC, Directory, and processor interrupt functions.

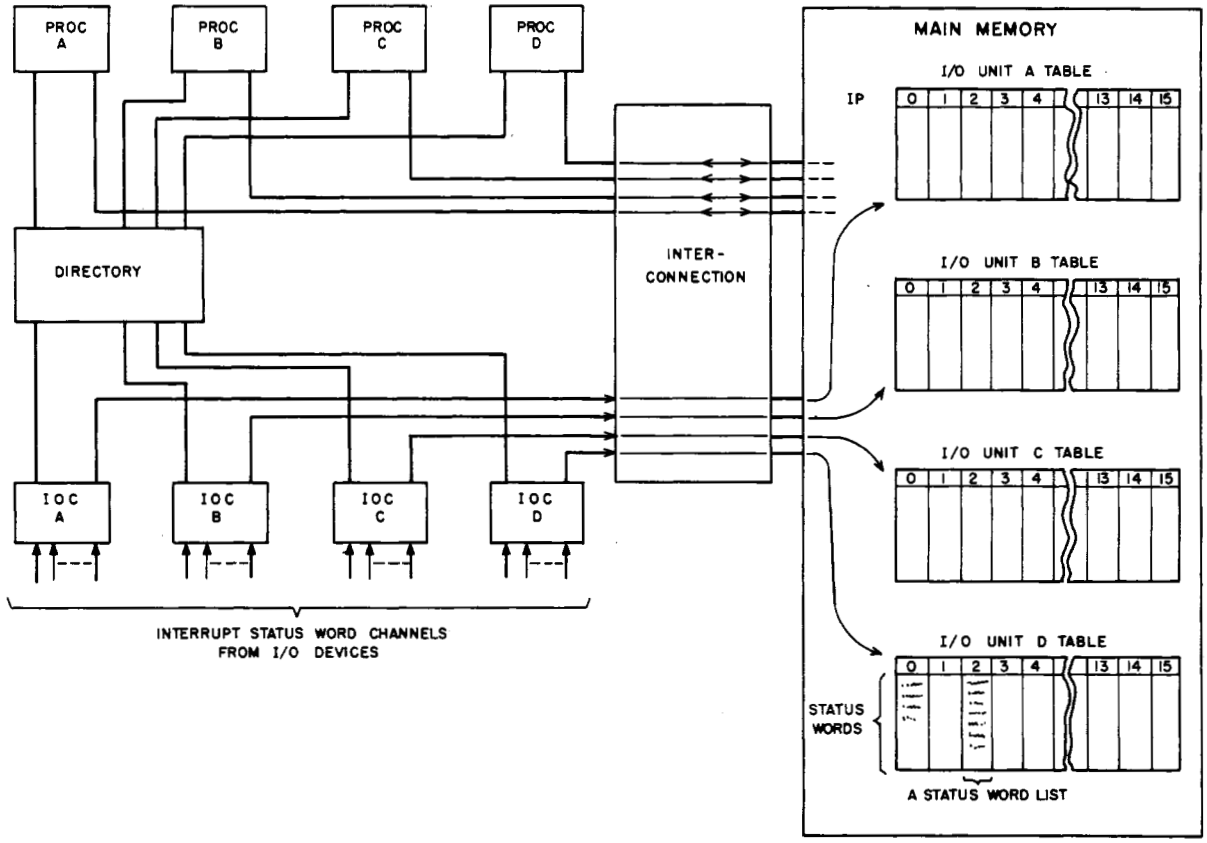

Fig. 5. Interrupt status word buffer in main memory.

If an acceptance is made, the Directory acknowledges the originating I/O controller, and only at this time is the staticizing command to that I/O controller released. While the I/O controller was held static, the staticizing applied only to its output lines to the Directory. Internally the I/O controller is free to accept additional interrupt requests and store the related status words in the appropriate list. Moreover, the I/O controller remembers which additional request had the highest IP in this period. As soon as the acceptance acknowledge is received from the Directory, this request is forwarded to the Directory as the candidate of this I/O controller for selection by the Directory in its next selection cycle.

If the processor returns a rejection, this same sequence of events occurs, and the Directory starts another selection cycle. If the original request still has top priority, the selection cycle amounts to a search for a different processor.

3.4 Processor Functions After Acceptance

Within an interrupted processor, the interruption sequences cause the current task to be suspended. All pertinent information associated with the task, including its II, is saved. This process is facilitated by having separate sets of machine registers for each major interrupt category plus the running task. (These registers are maintained in a very high speed local memory within each processor.) A new value of II is automatically instituted for the I/O interrupt processing. This processing is comprised of two parts. The first part is the preprocessing of the interruption. This handles the housekeeping required for allowing the nesting of interrupts if such is desired. The II for the preprocessing might be 15, making the processor noninterruptible or disarmed. The second part is the routine which actually handles the information associated with the interrupt. This handling routine may set the prevailing II to any value. Normally, the II would be set equal to the IP of the current interrupt. This allows the processor to be interrupted by higher priority I/O interrupts should they occur.

The IP value of the interrupt and the I/O controller number sent to the processor by the Interrupt Directory serve as a list pointer for use by the interrupt handling routine. This pointer defines the status word list where the interrupt status words corresponding to the current interruption are stored. Each status word contains all the information necessary to identify the source and reason for the interruption, as well as other data needed for proper reaction to the event.

The interrupt processing routine normally would process only the interrupt defined interrupt by the pointer. However, the program has the option of processing status word lists having IP's lower than indicated by the current pointer. Such an action effectively upgrades the priority of these lower level interrupts. Independent of the option, the fact that interrupts are outstanding for lower level lists is remembered in the form of the interrupt request indicators for those lists. Thus, interruptions for the lower level lists will occur at some later time when the proper II-IP conditions permit.

To cover the case of additional status words put in the

current priority list after processor acceptance, we must recap the effects of acceptance. The acceptance message from a processor is relayed by the Directory to the selected I/O controller. In the I/O controller, the acceptance signal "clears" the interrupt request indicator. All status words put in the list up to this time can be batch-processed by the interrupted processor. (Note that one processor interruption can handle many individual interrupt messages.)

Status words of the same priority stored after the acceptance is received *may* be handled by the currently interrupted processor. But the I/O controller's request indicator has been cleared since the receipt of acceptance; any interrupt signals will, therefore, reset the request indicator. This may cause the interruption of another processor to work in the same status word list. (Note that this provides a sharing of the interrupt processing load.) By means of a flag convention, processors are prevented from working on the same word. Special instructions exist for flag checking and manipulation.

When no other processor is interruptible, the processor currently working in this list need *not* "empty" the list. That is, the interrupted processor is under no constraint to handle the additional interrupt messages, because the interrupt system *guarantees* a later interruption should there be additional interrupt status words not handled by the currently interrupted processor.

4. Conclusions

The multiprocessor system described in Section 2 achieves complete processor anonymity by eliminating any processor specialization for arithmetic processing, input/output command, and interrupt handling. To this system is added the Interrupt Directory, which selects processors for interrupt handling based on the relative values of II's and IP's in the system, rather than associative connections between specific processors and I/O controllers. In the following paragraphs we point out several advantages of the resulting multiprocessor, in the area of optimum priority response, flexibility, and graceful degradation.

4.1 Optimum Priority Response

The mechanized processor selection algorithm provides a considerable reduction in the time from interrupt presentation to the system to the start of useful processing of the interrupt. The algorithm is performed at hardware speeds which are considerably faster than the execution times of any equivalent software. The job is complex and time-consuming when done in software even with the faster instruction cycles now available. Consider the software problem in determining:

1) if an interrupt exists
2) should it be processed at all
3) what processors are available
4) which of the available processors should get the job
5) reselection if processor status changes at the last instant.

Done in hardware, all these operations not only take less time but leave all processors at current tasks until the point of final acceptance. At this point the nonassociative aspect provides that only one processor need turn from its current task. Contrast this to a master-slave arrangement where the master processor may assign an interrupt to a slave processor and hence, two processors become involved in handling one interrupt request.

The nonassociative or peer arrangement also provides a faster response than a master-slave arrangement in the situation where two interrupts occur in quick succession. With the directory and a peer arrangement, the first interrupt undergoes a directory cycle time and a processor is selected and accepts the interrupt. The second interrupt has, as a worse case, a directory delay of two directory cycle times and then is picked up by a processor (assuming several available). In the master-slave system where the master is designated as *the* interrupt handling processor, the second interrupt undergoes a significant wait while the master handles the first interrupt, even if the master hands one interrupt over to an available slave.

4.2 Flexibility

Implementing functions in hardware will always restrict flexibility somewhat because one cannot dynamically modify the function. However, the II/IP scheme, even with a hardware directory, provides a high degree of freedom and flexibility in control methods without a high cost in the form of program complexity.

First of all, the II/IP scheme is essentially independent of task assignment operations performed by supervisory software. Such software is left free and unrestricted to perform task assignment on whatever basis is deemed advantageous. What the II/IP scheme governs is the relationship between an interrupt task and a problem task while the problem task is running. Executive software could attempt to govern this but at the expense of program complexity and longer response time. The Directory relieves the executive of a knotty problem in task selection, a problem that is best done in hardware anyway.

The ability to manipulate II and IP values under program control is the source of the flexibility in the scheme. There is complete freedom in the choice of algorithm for setting and varying these parameters within the framework of the requirements of the particular system application. The importance of a task relative to possible interrupts is but one criterion in setting an II. The II could be based on length of program (either program size or running time). Furthermore the parameter values could be adjusted at different points in a task to have greater or less interruptibility near the beginning or end of a task.

The IP of an I/O event is assignable by the program when the I/O operation is initiated that creates the potential for the interrupt. Or the system could be designed so that IP values are a function of both program control and a user action or external contingency.

A task or interrupt routine can be guaranteed freedom

from interruption, with the assurance that interrupt status words continue to be saved in main memory, and that some processor will be alerted if there is one "available." I/O events, which normally interrupt processors, can be set to queue status words only. One or more processors may inspect the status lists periodically as a function of an internal (processor-related) time clock. In fact, processor interrupts occasioned by I/O events may be eliminated completely. In applications where the time allowed for answering interrupt signals is long or matures at regular intervals, the interruptless system permits a simpler executive control scheme. Interrupt status word processing could be done entirely on a batch basis, if this is permitted by the priority considerations of the application. Note that the automatic queueing of status words in interrupt status buffers is essential to this type of operation.

Inherent in the II/IP scheme are certain arm/disarm capabilities. Assigning an IP value of zero certainly disarms the interrupt request as regards causing an interrupt. Any nonzero value constitutes either arming with variable priority, or multilevel arming, depending on your point of view. The system certainly has a generalized arm/disarm capability of being able to effectively disarm all interrupts (by raising processor II's) and subsequently restoring the same selection of armed lines merely by lowering the II's.[2]

Although the Interrupt Directory has been described above in terms of its role in routing I/O interrupts, it can also contribute to priority-response where I/O is not essentially involved. Consider a processor encountering a request for a branch, wherein each leg may be executed concurrently. The processor can initiate in any I/O controller a command which merely causes an interrupt at an IP specified to be equal to the II associated with the branching program. A parameter in the I/O command is copied in the resulting interrupt status word. Any processor whose II was lower than the branching program II, will be interrupted and, upon interpreting the interrupt status word, will find a request to assist in processing the more urgent branch.

4.3 Graceful Degradation

Graceful degradation, or fail-soft operation, is the ability of a computing system, upon failure of one or more of its component units, to continue the timely processing of tasks most critical to the mission, at the expense of neglecting tasks least critical to the mission. In order for a system to degrade gracefully, either its various units must be duplicated, so that when one fails its function may be assumed by another; or the units must be to some degree dispensable, so that when one fails it can be programmed around; or the units must be failproof. All three approaches are represented in the system described here.

The processors in the system described in this paper are functionally equivalent even with respect to the interrupts they can handle. The system control scheme automatically insures that tasks uppermost in the priority-ordered task list have preferred service from the pool of available processors. Processors executing the task selection portion of the executive need not be aware of the identity or number of other processors. Whatever processor is executing the task selection routine is steered to the highest priority task which is able to run. If one or more processors fail and turn themselves off just after ending or before beginning a task, the above described provisions are all that are necessary for the desired redistribution of work over the remaining processors. Of course, it is impossibly idealistic to count on this mode of processor failure. What must be done is to detect the failure, eliminate the failed processor from an active role in the system (until it is restored to operating condition), and recover from any indeterminate state introduced into the task by the defective processor.

Detection of processor failure is done by 1) conventional hardware failure detection logic, 2) execution of diagnostic procedures, possibly built into tasks, and 3) inspection by processors at task selection time, to insure that tasks which are supposed to be running have properly updated a "sign-in" log. The last provision takes into account failures in the form of stalling in the middle of a task or an instruction.

The actions taken in case of a known failure are to 1) raise the current processor II to 15, to prevent further I/O interrupts of the processor; and 2) stop the processor. These actions are performed automatically in the case of a processor stall.

The Interrupt Directory represents the one centralized unit in the system; therefore, it may be desired to construct all of its circuits of triply redundant majority logic, in order to make it "almost" failproof. When an Interrupt Directory failure does occur, all is not lost. Since transmission and storing of interrupt status words is entirely independent of Interrupt Directory operation, processors may continue to interpret information in the status buffers in the same way as before. The source of the stimulus for interpreting the status must then be an internal interrupt timer inside each processor, rather than I/O interrupt signals from the Interrupt Directory.

Acknowledgment

The authors wish to acknowledge the assistance of J. J. Dèch in the preparation of this paper.