AN INTERRUPT BASED ORGANIZATION FOR MANAGEMENT INFORMATION SYSTEMS

Howard Lee Morgan

Technical Report

No. 69-33

March 1969

Department of Computer Science
Cornell University
Ithaca, New York 14850

AN INTERRUPT BASED ORGANIZATION FOR MANAGEMENT INFORMATION SYSTEMS*

Howard Lee Morgan

Cornell University[1]

March 1969

Abstract: A programming structure, language constructs, and supervisory system organization are proposed for the design and coding of large shared data base systems. The bases for this organization are a generalized interrupt structure and the newly introduced concept of "file tagging," which is the process of associating program structures and interrupt generating conditions with items in the data base. An algorithm for resolving conflicts which arise in scheduling the interrupt processing routines is presented. DPL, a programming language and supervisory system in which these concepts are implemented, is used to illustrate the new organization which is proposed for management information systems.

CR categories: 4.32 (Supervisory systems, multiprogramming), 3.51 (Management data processing: education and research), 4.22 (Procedure and Problem oriented languages).

Keywords and phrases: management information systems, integrated data processing, supervisors, interrupts, monitoring systems, supervisory systems, interrupt scheduling, parallel processing.

---

[1] Department of Operations Research and Department of Computer Science, Upson Hall, Ithaca, New York 14850.

* Research supported in part by the National Science Foundation under Grant GP-6827.

AN INTERRUPT BASED ORGANIZATION FOR MANAGEMENT INFORMATION SYSTEMS*

Howard Lee Morgan

Cornell University[1]

1. Introduction

Large scale integrated information systems, of which management information systems are a special case, are coming into widespread use. This paper presents an organization which will aid in the design and programming of such systems. For the purposes of this paper, a management information system may be thought of as composed of four parts:

1. Data base — all of the information which is available to the rest of the system.
2. Data entry and updating — programs which are used to keep the information in the data base current.
3. Inquiry — programs which utilize the data base in a read only manner, e.g., online inquiry systems.
4. Supervisor — program which schedules the execution of all other programs in the system on a "when needed" basis.

In most current systems, the data base is represented as a group of files, and the programs for data entry, updating and inquiry are written in a conventional manner, calling on some operating system for input/output functions. The supervisor functions, however, are often distributed among several of the user's programs, the operating system, and even the human operators of the system. It is to the programming of a supervisor that this paper is directed.

---

[1] Department of Operations Research and Department of Computer Science, Upson Hall, Ithaca, New York 14850.

* Research supported in part by the National Science Foundation under Grant GP-6827

2. Systems Organization

In a typical management information system, the data entry and updating programs may include several functional groups of programs, each of which may require online access to the data base, and each of which may interact with any of the other groups. For example, there may be one group of programs to process payroll, another for order entry, another for inventory control, and still another for general ledger accounting, all as parts of a single, integrated system.

The supervisor must know what conditions require the execu tion of each program, and must have some means of detecting whe these conditions occur so that it can schedule the running of any needed program. An order entry, for example, may reduce stock, requiring the inventory control program to be run. This program may, in turn, order more stock which may require the purchasing program to be run, and so on.

In the DPL system, interrupts are used to signal to the supervisor the occurrence of conditions which require a progra to be executed, and the interrupt block structure discussed in Section 3 is used to indicate which blocks process which inter- rupts. The interrupt generating conditions are either Boolean conditions on variables or items in the data base, or involve hardware events.

Some of the items which are needed to evaluate the inter- rupt generating conditions may be in the out-of-core part of t data base. DPL uses a method called "file tagging," discussed in Section 4 to handle this case. The implications of the interrupt structure together with file tagging is a new, parallel organization for management information systems.

3. Interrupts

The growth in the use of interrupt mechanisms in computer hardware technology has paralleled the growth in sophistication of the software technology associated with monitor or operating systems. Interrupts were first used to provide a convenient means for the hardware to inform the monitor that certain events had occurred, e.g., completion of an I/O event [2]. The second stage in the use of interrupts saw the class of monitored events broadened to include what were essentially software errors, e.g., division by zero or attempts to execute illegal operation codes [3]. When the third generation of computers was introduced, with plans for still more comprehensive operating systems, a third stage in the use of interrupts was initiated [8,12]. This third stage added a new class of events which could cause interrupts to be generated, namely, the execution of a special instruction, the "supervisor call." The IBM 360 series also allowed many more of the software error (program check) interrupts than did earlier computers.

Until the third stage in the use of interrupts began, essentially all interrupt processing routines were part of the monitor system, and were inaccessible to the programmer who was not writing in assembly level language. With the increased sophistication provided in third generation hardware and software the PL/I language designers were able to allow the programmer to define his own interrupt processing routines for most of the conditions which the 360 hardware monitors, and some additional conditions which were monitored by software [10]. The user writes statements of the form: ON FLOATINGDIVIDE <procedurename> This procedure is then executed whenever a floating point divide exception occurs. These procedures, called interrupt function modules [13] may also be entered by executing a SIGNAL statement, which simulates the occurrence of an interrupt.

A fourth stage in the use of interrupt mechanisms is proposed here, and has been implemented in an experimental language called DPL (Data Processing Language) [7]. The programmer is now allowed to specify rather complex events whose occurrence will cause interrupts to be generated. There are two classes of interrupt causing events which are felt to be most useful in designing management information systems. These are the device interrupt class, e.g., the "attention" key on a teletype, and the program generated interrupt, which occurs when a specific Boolean condition on some combination of expressions occurs. The programmer indicates which block (subroutine) is to be used to process which interrupt be writing statements of the form

PERFORM <blockname> WHEN <condition> .

Some examples of the two classes of interrupts are:

PERFORM ORDER WHEN STOKLEV LE REORDER

PERFORM SERVICE WHEN TELTYP(3) INTERRUPTS .

The first statement would cause an interrupt to be generated whenever the value of the variable STOKLEV is less than or equal to the value of the variable REORDER. Of course, the condition could be far more complex and could involve any Boolean condition which could appear in an IF statement. The second statement would cause an interrupt to be generated whenever the attention key on TELTYP(3) was depressed, and would transfer control to the block named SERVICE.

The problems which can arise when several interrupts are generated as the result of a single assignment statement are treated in the next section.

Monitoring for these interrupts is performed through software by the DPL system, but conceivably this could be done by some form of microprogrammed hardware or firmware [9]. The new SIMSCRIPT 2 programming language includes a feature called "monitored variables" which could be used to implement a software checking structure, but has been designed with simulation in mind [5].

A similar generalized interrupt structure was proposed in some early work on the BCL language, developed in England, but was not carried through into implementation, probably because of the problems which arise in scheduling these interrupt blocks for execution [1].

3.1. Interrupt Block Scheduling

The execution of a PERFORM...WHEN statement causes the system to watch for the condition mentioned in the WHEN clause and, when that condition becomes true, to execute the interrupt block named in the statement. This is the program controlled interrupt feature of DPL.

When a PERFORM...WHEN is executed, the pair composed of the interrupt block name and the condition (denoted as the (b,c) pair) is placed on the "pending block" list (PB list). At the same time, a flag is set in the main symbol table for all variables which are used in the condition. For example, if the statement:

PERFORM BLKC WHEN I-J+3=K

were executed, the pair (BLKC,I-J+3=K) would be placed on the PB list and the variables I , J , and K each would have a flag set in the symbol table.

The execution of a CANCEL...WHEN statement would remove the pair designated in the CANCEL statement from the PB list. If the indicated pair is not on the PB list, an error message is generated.

Whenever a statement which can assign a value to a variabl is executed, e.g., LET, READ, PERFORM...FOR, the symbol table entry for that variable is examined. If that variable is flagg as being involved in some condition which is on the PB list, that condition is evaluated, along with any other conditions on the PB list in which that variable is involved. If any of the conditions have the value "true", the corresponding (b,c) pai is placed on the "to be executed" list (TBE list), and removed from the PB list.

The actual checking of conditions and generation of interrupts takes place upon the completion of the statement which performed the assignment. If a statement performs multiple assignments, e.g., READ X, Y , the evaluation of the affected conditions reflects all of the assignments. This is comparable to the doctrine on most machines that interrupts may only be accepted between instructions. At the level at which the DPL programmer writes, a DPL statement is equivalent to a machine instruction.

Note that the condition is checked only on store operations and is not checked at the time the PERFORM...WHEN is executed and the (b,c) pair is added to the PB list. This is similar to the convention followed in some computer systems, namely: The execution of an interrupt enable command does not enable the interrupts until after the execution of the instruction following the interrupt enable instruction [11]. The reason for doing this in DPL is simple. It allows the last statement in an interrupt block to be a PERFORM...WHEN which will put the (b,c) pair for that block back on the PB list. Presumably, if the condition were checked upon execution of the PERFORM...WHEN statement, the condition would be true and a nesting problem would arise.

If there is only one (b,c) pair on the TBE list after an assignment statement has been executed, and the interrupt block named in it does not issue any PERFORM...WHEN statements, the block is performed and control is returned to the statement following the assignment statement. When there is more than one pair on the list, however, or some of the interrupt blocks issue PERFORM...WHEN statements, thus adding pairs to the PB list while an interrupt block is executing, the situation becomes quite complicated. The following examples may help to illustrate some of the problems which can arise, and will be used to show the rationale for the scheduling algorithm which was chosen.

1. Suppose the PB list contains the two entries (B1, X LE 5) and (B2, X LE 10) . The statement: LET X=4 is executed. Which block should be executed first?

2. Suppose the PB list contains the two entries (B1,X=2) and (B4, X=2) . Suppose further that execution of block B3 will not change the value of X , but execution of B4 will set X to 3 before exiting. Again, which block should be executed first?

3. In this example the PB list consists of the single entry (B5, X=3) , and the first two statements in B5 are:

   PERFORM B6 WHEN X=2
   LET X=2

   Should the execution of B5 continue after these two statements are executed, or should B6 be entered after execution of the LET statement?

Some of the problems arise from interaction of one interrupt block with the conditions on the PB list, and others arise from interactions between the PERFORM...WHEN statements, which may not all be consistent with each other. The execution of one block when more than one has been placed on the TBE list may change the condition which caused other blocks to be placed on the TBE list. Furthermore, the ordering of the execution of the blocks on the TBE list may affect the number of blocks which can be executed as a result of a single assignment statement.

The criteria used in developing the algorithm which scheduled the execution of interrupt blocks were:

1. When an interrupt block is entered for execution, the associated condition must be true. (This may have been assumed when the block was written.)
2. A unique ordering for the execution of the blocks must be guaranteed.
3. When several pending blocks are scheduled as the result of a single assignment statement, the execution of any one of these blocks should be transparent to all of the other blocks, i.e., each block may be written as if it will be the only block executed when an assignment is performed.
4. Nested interrupts should be treated with lower priority than those interrupts which are generated as a result of the initial store operation.
5. As far as possible, as many of the blocks which are initially placed on the TBE list should be executed.
6. The algorithms should be aware of any conflict situations which it cannot handle, and should report these to the programmer.

Algorithm A, presented below, meets all of these criteria and is the one used in DPL to schedule interrupt blocks.

ALGORITHM A

Step 1. Label all (b,c) pairs on the TBE list as level 1.

Step 2. Select from the TBE list all interrupt blocks whose execution will not change any of the conditions associated with blocks on the TBE list. That is, select those blocks which make read only accesses to variable which are involved in conditions on the TBE list. Execute the selected blocks in any order and remove from the TBE list.

Note 1: If any new interrupts are generated during the execution of any of the blocks executed in Step 2, add the new (b,c) pairs to the TBE list and mark the new blocks as level 2. Also remove these blocks from the PB list.

Note 2: If any PERFORM...WHEN statements are issued by any of the blocks executed, add the (b,c) pairs to the "reschedule" list, and not to the PB list.

Step 3. Test the condition, c, for the next (b,c) pair marked as level 1 on the TBE list. If c is true, go to Step 4. If c is false, go to Step 3. If the end of the TBE list has been reached, go to Step 5.

Step 4. Execute the selected block and remove the associated (b,c) pair from the TBE list. Notes 1 and 2 above apply to this execution. When execution is completed, go to Step 3.

Step 5. Flag all blocks remaining on the TBE list at level as "in conflict," and remove these pairs from the TBE list, placing the pairs on the "reschedule" list. If the TBE list is now empty, go to Step 6. If not, go to Step 1.

Step 6. Add the (b,c) pairs on the "reschedule" list to the PB list and delete the "reschedule" list. Return control to the main program at the point where the first interrupt occurred.

It is instructive to examine the performance of this algorithm on the examples given above. In example 1, the pair which had been placed on the list first would be executed first. In the second example, the algorithm would first execute block B3, which desires only to read the value of X , and then would execute B4. B3 might, for example, be printing an exception report while B4 might actually take action on the exceptional condition. In example 3, the new (b,c) pair would not be placed on the PB list until B5 was exited, thereby eliminating the nesting problem.

Step 5 in Algorithm A, which flags conflicts in accordance with criterion 6, is a rather important feature of DPL's handling of interrupt blocks. If there is more than one programmer at work designing and programming parts of a large system, ambiguities in the description of the responses of the system will often arise. The algorithm actually tries to cope with these ambiguities, and only after all means of coping with the ambiguity have failed will it give up. Another example of this type of algorithm has recently appeared in the context of decision tables [4]. This bears a close relation of DPL's problems since the entire program controlled interrupt structure may be thought of as an asynchronous decision table processor.

4. File Tagging

When a data base is shared among several users, one must be careful to assure that when a user is modifying part of the data base, he is really working on an element of this data base, and not a copy of this element. For example, if two parallel processors each try to fetch an element, add 1 to it, and store it back at the same time, the result will be original value plus 1, and not plus 2 as is required. The DPL system takes pains to keep track of the current location of all elements of the data base, so that when a condition is evaluated, all values used in the evaluation are current. This may sometimes require access to secondary storage. If a file containing a value of a variabl which is on the PB list is closed, file tagging will take plae .

When a file is tagged, the conditions on the PB list which contain elements of that file, along with the associated interru processing blocks, are attached to the file, i.e., pointers to these items are added to the file. When that file is next opened for access, either by the user who tagged the file or any other, any conditions which are attached to the file are added to the current PB list. The initial user has, so to speak, put his tags on the file specifying conditions on some elements of the file and, if those conditions should become true, appropriate processing action.

Several successive programmers may all put their tags on a file, the effect being cumulative. Whenever the file is opened, all of the associated conditions are added to the current users PB list. Of course, any tag may be removed from a file by opening it, issuing a CANCEL, and closing it.

One previous attempt to connect programs and files was made by Lombardi [6]. His attempt was based on static decision table logic and was more suited to production control than for use as a building block for management information systems.

When several different programmers wish to share the data base, which is the normal case, certain naming conventions must be followed by these programmers so that the scheduling mechanism can work properly. These conventions could be eliminated in a production system, but would leave the system open to more errors than can occur when the conventions are in force. They insure that any field of a file will always be given the same name by all programs accessing that field.

5. A New Organization for Management Information Systems

The use of the file tagging and program controlled inter-tupe features allows a new organization for management systems. Rather than the four parts mentioned at the start of this paper, the management information system would consist of two parts; a tagged data base and a supervisor program. The contrast between this form of organization and current methods of organizing systems is shown graphically in Figure 1. The new organization is closely related to parallel processing while the old reflects years of sequential processing experience.

In the parallel system, each program is written independently of all others, and is then attached to a file. The programmer also provides the supervisor with the PERFORM...WHEN statements needed to start the execution of his program. For example, the inventory file might be tagged with one set of programs by which the order entry requests are handled and a set of tags which would ensure that inventory control runs were made when the stock level dropped below a preset order point. In the sequential system, the order entry programmers would have to work intimately with the inventory control programmers to ensure everyone's satisfaction.

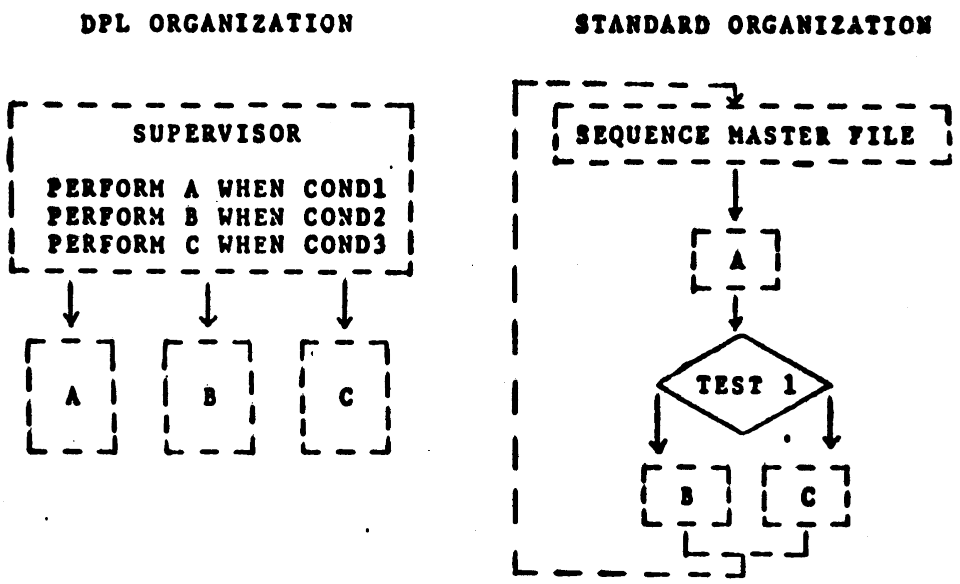

Figure 1. Information Systems Organization.

The supervisor program would first open all of the files in the data base, thereby setting up a large PB list with all of the conditions which must be monitored. It would then open the remote terminals and either begin to poll them or accept device interrupts. Finally, the supervisor would begin the execution of some background task.

If one of the terminals interrupted and requested the program to update a particular file, that program would be executed, possibly setting off a chain of execution of other interrupt blocks. After processing was finished, the system would return to its background task.

The conflict recording mechanism of Algorithm A would indicate to a programmer when he was in conflict with some other programmer's requests for a file. These conflicts could then be handled manually between the programmers. What this amounts to is using the files as the main interface between programmers.

6. Summary and Conclusions

An organization has been proposed for management information systems. This organization is based on a generalized interrupt structure and scheduler, and a method of associating programs with elements of the data base, called file tagging. This organization is valuable for two reasons:

1. It makes the division between those parts of a system which perform processing and those parts which schedule that processing clear. This may allow programmers of varying ability to be assigned to that section for which they are best suited.
2. It allows the large number of programmers usually involved in writing a system to communicate and resolve conflicts regarding the data base in an automated manner, through the system. This may tend to reduce the bugs which arise when this communications and conflict resolution is done manually.

It is hoped that the advent of systems which allow easy microprogramming of special functions will permit an implementation of these ideas in the construction of a real management information system.

REFERENCES

1. Hendry, D. Provisional BCL Manual. Institute of Computer Sciences, London, England, 1966.

2. IBM 709 Data Processing System. IBM Manual No. A22-6536, 1959.

3. IBM 7040 Principles of Operation. IBM Manual No. A22-6649, 1962.

4. Jackson, M.A. The need for imprecision. Datamation 14 (February 1968), 143-144.

5. Kiviat, P.J., Villenveve, D., and H. Markowitz. The SIMSCRIPT II Programming Language. Englewood Cliffs, N.J.: Prentice Hall, 1969.

6. Lombardi, L. A general business oriented language based on decision expressions. Comm. ACM 7 (February 1964), 104-111.

7. Morgan, H.L. DPL: A language for instruction in contemporary data processing concepts.' Technical Report No. 53, Department of Operations Research, Cornell University, Ithaca, New York, 1968.

8. OS/360: Concepts and Facilities. IBM Manual No. C28-6535- 1965.

9. Opler, A. Fourth generation software. Datamation 13 (January 1967), 22-24.

10. PL/I Language Specifications. IBM Manual No. C28-6571-4, January 1965.

11. Small Computer Handbook. Digital Equipment Corporation, Maynard, Mass., 1967.

12. System 360 Principles of Operation. IBM Manual No. A22-6826-6, 1967.

13. Wagner, P. Programming Language, Information Structures, and Machine Organization. New York: McGraw Hill Book Company, 1968.