Presented Oral at The Fall Joint Computer Conference, San Francisco, October 27-29, 1964. Will be published in the proceedings of this Conference.

MAY 15 1964

CONF-518-1

CONSEQUENT PROCEDURES IN CONVENTIONAL COMPUTERS*

MASTER

D. R. Fitzwater (Tel. 515-231-5055)
Institute for Atomic Research and Department of Chemistry
Iowa State University, Ames, Iowa
and E. J. Schweppe (Tel. 301-927-3800, x7600)
Computer Service Center, University of Maryland
College Park, Maryland

ABSTRACT

The use of consequent procedures in designing programming systems and languages for real time, on-line or multiprogramming applications is discussed. Consequent procedures differ from normal procedures in that the flow of control between procedures is defined by the consequences of previous procedure rather than by the programmer. The use of these concepts leads to comparatively simple system and language design for general purpose systems. Interactions with the external world of human and non-human users are simply described and implemented.

1.0 Introduction.

The complexity of the tasks performed by computer systems has been expanding rapidly throughout the short history of these machines, but only in recent years has the basic feature of sequential control of operations been seriously violated. The large

* Contribution No. 1509 work was partially supported by the Ames Laboratory of the U. S. Atomic Energy Commission and partially by the National Aeronautics and Space Administration under grant NsG-398 to the Computer Science Center of the University of Maryland.

LEGAL NOTICE

This report was prepared as an account of government sponsored work. Neither the United States nor the Commission nor any person acting on behalf of the Commission

A. Makes any warranty, representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights, or

B. Assumes any liability with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or in the performance of which such contractor

Facsimile Price \$ 3.69

Microfilm Price \$ 1.28

Available from the
Office of Technical Services
Department of Commerce
Washington 25, D. C.

computing system of the future will have multiple processing capabilities and will be operated in a shared manner in order to obtain the potential efficiencies and expansions of application areas which are possible in such a system. A shared computer system will be heavily dependent on real time interactions with people and other machines. The effectiveness of such expansion in the application areas of shared systems depends heavily upon advances in both hardware and software structures, and upon the feedback between them. In order to solve the language problems of such a system, it is not sufficient to try to find a more convenient language to describe conventional program structures.

The concepts of consequent procedure networks, discussed by Schweppe and Fitzwater 1 , provide an effective framework for considerations of hardware and software design for such systems. Although the use of such general network implementation in controlling computer complexes would be quite effective, their full implementation within a given conventional processor becomes inefficient. The purpose of this paper is to discuss the application of a related but somewhat degenerate form of consequent procedure networks inside a conventional computer.

The results of this study have been used in the design of a shared on-line computer control system as described by McFarland, Fitzwater and Stewart 2 , 3

1.1 Need for Consequent Procedure Networks.

The procedures which are carried out in a complex system must be formed in a sequence which is dictated by the demands of the

people or equipment being serviced and the existence of the necessary inputs. Since this order is determined only during execution of the procedure--and indeed may vary from one execution to another--the program structure should be an invariant of the order of execution. Clearly most present languages and systems do not allow such flexibility. The requested procedure itself may have further interaction with the requesting entity as well as other entities and such interaction may arise from a variety of causes, some of which are requirements for competitive processes, random demands for data acquisition, and indeterminate processing times. As a result the language used to describe procedures should contain, neither implicitly or explicitly, any artificial requirements as to the order in which operations are carried out.

Many ad-hoc system programs have been designed and operated successfully within real time computers. These systems have usually been dependent on a specific design for a specific set of tasks which are known at the time the system is developed. In this case any structure which will process the specific tasks is satisfactory. The problem of designing a general purpose system for processing unspecified procedures for possibly unspecified users is quite different and is much more akin to the design of a general purpose computer. Obviously, in such a system, one cannot guarantee that all of the unspecified requests of the unspecified users can be satisfied. The goal is to design a general purpose system which is capable of satisfying a large class of such requests. Such a system must recognize that the structure of the processes

which it must carry out is not sequential in nature.

An implicit recognition of non-sequentiality in an on-line system may be effected by the technique of memory swapping as discussed by McCarthy, Boilen, Fredkin and Licklider 4 . In this scheme, the currently requested programs are run for a short time each in sequence. The turnover rate is made fast enough so that the effect on the serviced entity is that of a somewhat slower computer. The major advantage of such a system is that conventional languages and coding structures may be used in each user's program.

The major disadvantage of such a technique is that the consequent relationships of sub-procedures are not explicitly represented in the structure of the conventional language and cannot be implicitly recognized except in rather special cases. As an example, the use of priority overlapped input-output equipment in FORTRAN programs cannot be expressed explicitly and is only sometimes expressed implicitly in the structure of the systems package. The programmer describing a procedure has no direct control over such implicit structures. If the only non-sequential operations to be carried out by the system involve simultaneous reading, writing and computing it is possible to provide implicitly for such operations and some FORTRAN systems do this. When the system is operating under a hierarchy of priorities or priority interrupts the implicit recognition of non-sequential structures lays impressive demands on the system package design. These demands become oppressive if further, non-standard non-sequential operations are required in describing the procedure. Because such structures

may be peculiar to a specific task, implicit recognition of such general structures in the system package is neither feasible nor desirable.

Although, in a specific system, the decision to relinquish any part of conventional language and systems structures must be carefully considered, the inherent awkwardness of conventional system structures for general purpose real time systems leads to a hope for a more natural and effective structure which does not place such severe demands on the user, programmer and systems designer.

The use of consequent procedure networks is an alternate and effective approach to a solution of the language and system structure problem. The introduction of non-sequentiality occurs in a natural and flexible manner.

1.2 Operational Requirements.

The computer system must satisfy the needs of the user, programmer and systems designer. There is no a priori reason for assuming that these needs are identical and can be represented in the same communication languages and techniques. There is little point in re-examining language and systems structure unless these needs are re-examined and the results used in the language and systems study.

In general the user is interested in requesting the application of a specific procedure to a specific set of data to produce certain required responses. Although the nature of the data, procedure, or response may vary wildly, the requirement that the

system must accept and process such requests for the user is sufficient. The sequences of such user's requests and the data descriptions are the responsibility of the user and he must therefore be given the ability to define these as he sees fit. The computer system must supply a flexible means of communication for such requests and data and a convenient language in which they may be described. The user may also be interested in supplying a priority parameter to be used in scheduling.

At the current level of discussion there is no implied distinction between human and non-human users. The formal language or structure of a request from a non-human user may be quite different from that appropriate to a human user. This does not present any serious problems since the system must be able to accept requests in a "language" suitable for the using entity. Of course, not all interacting entities have the status of a user. Some may merely supply or receive information.

The programmer has a two fold relationship with the computer system. First, he must define the operations which will perform the desired task in the form of a procedure. Second, he has the same requirements as a user while entering the procedure into the system and during the checkout process. The system must provide an appropriate language for the definition of a procedure. There is no reason to assume that this language should be the same as for the user's requirements. Because many sequencing operations are under the control of the user or the system, the language must provide sequence independent structures. In general, the

programmer should not care who or what might call his program into action. The programmer is merely defining a procedure. The sequences of such operations may be dictated by the user or incorporated in some encompassing procedure. In addition, the execution of a procedure may involve external real time interactions of undefined sequence.

The systems designer is responsible for the choice of computer hardware, procedure languages and operating systems. The design must satisfy the requirements above for users and programmers and should be realizable in an efficient manner. The systems designer must supply translators for the various languages used and must supply an operating system that will accept requests and data. The requested programs must be obtained by the system, supplied with actual parameters, scheduled and executed under system control. Because of the special requirements of such a system program, the user and programmer languages may have to be supplemented with a system language which gives to the systems programmer complete access to all hardware capabilities. The programmer languages do not need such complete access and may be designed accordingly.

1.3 Consequent Procedure Networks.

The requirements outlined above can be met in a natural manner by the use of consequent procedure networks. The use of procedures in describing a program has a long history of effective use. The network description of processing flow provides a natural way of introducing non-sequential structures of almost any desired

type. A consequent procedure network consists of procedures, linked into a network by the programmer, which are executed only when each recognizes the appropriate conditions for its execution. The conditions rather than the sequence is defined by the network.

In the general form, the specifications of the input data entities and the connection network of the procedures involved is sufficient to define the program. The mere existence of the specified data is enough to sequence the various procedures in a network. In the general form, data entities will carry network identity and may have to be stacked in input queues. The manipulation of such entities, in a conventional computer, becomes somewhat inefficient if full generality must be maintained.

The general form of consequent procedure networks loses many of its advantages in simplicity and efficiency when implemented on a single computer without substantial multiprocessing capabilities. The logical complexities in implementing such a system on a conventional processor are severe. A degenerate form of the procedure network is needed in a conventional computer. The general procedure in a procedure network may represent the operation of a unique piece of hardware or it may represent a body of coding within a single computer. Since the first meaning above is appropriate to a computer complex or to a multiprocessor but not to a single processor we will use the second meaning. A procedure represented by a body of coding may have input queues representing various calls on the procedure or, for each call, the body of coding may be replicated as a distinct entity. For our purposes,

the inherent simplicity of the last interpretation is a very promising line of approach to the introduction of non-sequential structures in conventional computers.

We are thus lead to a dynamic tree structure in which branches are executed in non-sequential fashion. The non-sequential characteristics enter at a very fundamental level in the resulting language structure.

2.0 Task Tree Structures.

A procedure is a description of a body of coding designed to operate upon the formal parameters of the procedure. When such a procedure is called and supplied with actual parameters, a task is formed. The process of creating a task is carried out by the system in response to a request from an existing task. Although the structure of a procedure and of the system is unusual in this system, the creation of a task is essentially that of retrieval of a subprogram from secondary storage, allocation and relocation into primary storage, and of replacement of formal parameters by actual parameters. Having created a task, the system will then schedule the task for execution. The introduction of non-sequential characteristics will be discussed in a latter section on the structure of a procedure. For the present, let us assume that when the appropriate activation conditions for a task are met, a request for the creation of that task is made of the system. The recognition of such conditions is made in the requesting task. A task is erased from primary storage by the system upon request from the

task. The completion of the task is recognized within the task. Note that a task has only a transient existence while it is actually operating upon a specific set of actual parameters. A given procedure may be used simultaneously in several separate tasks since each call for the procedure causes the creation of a unique task. Because of this, recursive structures may be implemented in recursive calls for procedures with no further considerations.

A task tree consists of a task and its associated subtasks as shown in Figure 1. Although a certain task tree may exist implicitly in a set of library procedures, it exists explicitly only in the primary memory as a task and a set of subtasks which may in turn have subtasks. The task tree will grow in response to requests for subtasks and will shrink in response to requests for erasures of tasks.

2.1 The Primary Task.

A mechanism for the creation of task trees in response to a request from a user is distinct from the mechanism which is built into the system to process requests from existing tasks for new subtasks. The language of such a request from the user must be designed for the convenience of the user. This requires a somewhat more sophisticated analysis by the computer than is required to process internal requests from existing tasks. In addition, the task tree creation process involves certain housekeeping which is not required for extensions of an existing task tree. Consequently a special task, called the primary task, must exist as a

Figure 1. An Example of a Task Tree.

part of the system. There is only one primary task in the system and it forms the head of all user requested task trees as shown in Figure 1. The primary task will create a main task of a task tree in response to each user request. Since the requests may come from different users, many users may have separate task trees in operation at any one time.

The services offered to the user by the primary task depend upon the requirements of a specific implementation. Since the primary task is the only place the system interacts with the user (as distinct from the programmer) much thought must be given to user language requirements. Many compromises, due to local conditions, would be expected here. In what follows, only suggestions of possible features to be provided by the primary task are made. Others may be suggested by local requirements.

The primary task must accept a user request in the external user language and analyze the meaning of the request. The primary task will keep track of current (or anticipated) hardware requirements and if the current request cannot be satisfied because of hardware limitations, it can be stacked into secondary storage for later response. If a priority user request is permitted, and cannot be satisfied because of hardware limitations, the primary task may request (via a global variable) a lower priority task to terminate its operation and lodge a request for its later continuation. The priority request may then be handled in a normal fashion. The low priority tasks would be programmed in such a manner as to recognize the suspension of operation request and relinquish

its hardware requirements, leaving only a restart task in memory.

If there are hardware facilities which are currently unused, the primary task will examine its pool of unprocessed user requests for suitable tasks. A scheduling algorithm may be used to prevent the more demanding requests from being indefinitely postponed.

The system must be stable to transient overloads in the sense that it must not saturate in a logical blockage of processing. A transient overload may impair, temporarily, the reflex time of the system but the system must continue to work on its backlog. One form of logical blockage which is quite disastrous is to find that none of the task trees can be completed unless a new subtask can be created and there is no more available core space. This might be handled by removing, temporarily, a portion of the task trees in order to finish the remainder. This procedure is fraught with substantial complications. A simpler and more satisfactory procedure is to have the primary task monitor the current core requirements and reject or stack requests which could result in this condition. This may be done by supplying, with each task, a parameter which defines the minimum core space required for completion. This minimum may be much less than the maximum because, under crowded conditions, some subtasks which might be done in parallel are done in sequence without intervention by the user or programmer.

Once the primary task has determined that the current request can be satisfied by the available hardware, a normal request of the system is made for the creation of a task which will become

the main task of the task tree designed to answer the request. The parameters for this task are provided by the user as a part of his original request.

The user may wish to monitor or alter the progress of the program. The primary task will therefore accept requests for display or modification of the parameters of the original user request. Some of these parameters may be programmed to provide the desired information.

The primary task may also provide notification to the user of the real time suspension or completion of the requested task if desired.

2.2 Main Task.

A main task is a task which has been created by a request from the primary task in response to a request from a user. There will be a main task for each user request currently being serviced. The main tasks may arise from different users, thus providing for sharing of the system among users.

The distinction between a task and a main task is in its position in a task tree and not in its program. Any task may be requested explicitly by the user, thus forming a main task, or implicitly by the user through the task tree, thus forming a subtask. Any procedure in the library may be requested as a main task or as a subtask as is currently required by the user. For example, a subtask designed to invert a matrix may be called as a main task by a user with a specific matrix or as a subtask to a main task which is to solve a specific set of equations. This

property has important consequences in code debugging and in real time control situations which will be discussed in a later section.

A discussion of the internal structure of a main task is identical to that for a subtask and is given in the next section.

2.3 Subtask.

With the exception of priority tasks (which will be discussed in a later section) all tasks have the same structure. A subtask is distinguished only by its position on a task tree as other than a main task. A simple subtask is a task which has no associated subtasks. A task which has potential subtasks but does not in fact request their creation is not a simple subtask.

Before we can discuss the structure of a task we must first discuss the structure of a procedure which can form a task. A formal definition and examples of such procedures must wait until a possible language to describe such procedures is developed. A general discussion of such procedures is required here in order to introduce the non-sequentiality concepts which must be explored before the definition of a formal language.

A procedure body consists of a sequence of statements which are executed in a normal fashion. One additional type of statement, a subtask call, is provided. A subtask call has three components. The first component is the name of a procedure in the library. The second component is a list of actual parameters which, in conjunction with the name of a procedure, provides the information to be used in a request for the creation of a subtask.

The third component is an activation parameter. The activation parameter is a Boolean variable. A TRUE value of the activation parameter implies that the appropriate conditions for creation of the specified subtask do now exist, and that it is appropriate to request the creation of such a subtask.

If, during the execution of a subtask call, the activator is FALSE or if the subtask already exists, no further action is required and the next statement will be executed. If, however, the activator is TRUE and the subtask does not exist, a request for the creation of the subtask is lodged with the system and the next instruction is then executed.

The system will subsequently request the appropriate procedure from secondary storage and go on about its business. When the specified procedure is available, allocation of storage is made, the procedure is relocated, actual parameters from the call are supplied and a request for execution is made of the task scheduler. At the appropriate time, the scheduler will make an actual call for the task and cause its execution.

When the system scheduler executes a task it does so by turning control over to the coding body of the task. This coding body acquires the references to the actual parameters and then executes the set of sequential instructions referred to above. The process described above is then repeated. The set of sequential instructions act primarily as an activation scan of the potential subtask. Statements to perform other useful operations may also be embedded in the activation scan. Since activation conditions are changing

dynamically, further activation scans must be made at later times. In the absence of hardware multiprocessing or associative memories for the storage of activation parameters, efficiency dictates that the system must not continually monitor changes in activation parameters. An effective and economical compromise is to stimulate an activation scan of a parent task each time one of its subtasks either completes or requests an activation scan of its parent.

The use of an associative memory which would eliminate the need for scanning activators would make it possible to reduce the housekeeping and increase the flexibility of the activation concept. An alternative approach would have a second central processing unit carry out the scanning, allocation and other housekeeping. In the later form, queue manipulation becomes feasible and the very powerful general concepts could be implemented effectively.

A task is considered completed if, at the end of the execution of its sequential instructions, no subtasks are currently active. Upon completion of a task the system will make the hardware components involved in that task available for use of other tasks and the task will cease to exist.

2.4 Simple Tasks.

If the sequence of statements in the body of a procedure does not contain a subtask call statement, the statements will be executed sequentially and the task is ended. Such a task is called

a simple task. The use of simple tasks on any level of a task tree is exactly the same as a conventional subroutine, except for its activation and scheduling. Indeed, most conventional programs could be considered to be simple subtasks to the conventional monitor.

2.5 Priority Tasks.

One of the weaknesses of conventional languages is the difficulty of describing hierarchical priority interrupt routines. In the present structure, this is accomplished by a simple task. There is very little reason for explicit non-sequential operation below the priority interrupt level and so the use of a simple task for priority routines is quite appropriate.

A priority task differs from a simple task in that it has three coding bodies associated with it. The first coding body functions just as a simple task whose job it is to create an entrance to the second coding body which defines the appropriate process to follow a priority interrupt. The second coding body then awaits one or more priority interrupts and makes its own decision on when its task is completed. The second coding body, when the task is completed, makes a special request to the scheduler for the execution of its third coding body. The reason for the third coding body is that a task may not relinquish control in the priority mode to anything except the point at which the priority interrupt occurred. The third coding body then releases control to the parent task.

The first and third coding bodies may be used also for initiation and termination respectively of external operations associated with the interrupt.

In the normal mode of operation, the conditions giving rise to an interrupt are not created until after the corresponding priority task has been requested and created. Thus the priority reflex time is not dependent on secondary storage or transfer rates. This does imply that all priority tasks must exist whenever their priority interrupts are currently expected. The priority tasks handle the high speed reflex requirements of the system and the non-priority routines handle housekeeping and on-line processing. A given task may, of course, involve subtasks of both types.

A very usefull priority task is one which recognizes a clock pulse as its interrupt entrance condition. Control is given to the task in the priority mode and it releases control to the next clock priority routine in the priority mode. The system is responsible for recognizing the clock pulse interrupt and for scheduling the pending clock priority tasks. Such routines may be used to monitor control operations or for periodic data logging.

2.6 Activation.

We can associate a status with each variable. The status of a variable has two values, active or inactive. For example a matrix M with inactive status may be given active status when a particular set of values have been assigned to its elements. However,

unless we wish to introduce queues, we must not assign a new set of values to the matrix $\underline{M}$ until it returns to the inactive state.

The activation parameter for a subtask call may be the state of a variable or a Boolean variable derived by logical manipulations of other variables or states. The concept of the consequent subtask call requires that a request for the subtask be made of the system each time the activator goes active. Again, in order to prevent queuing, we must ignore any activator for a subtask which has already been requested until that subtask completes. This is where we lose the major portion of the generality of consequent networks with queues.

The activator is reset to inactive upon completion of the corresponding subtask and a new call is permitted if the activator should subsequently become active.

The more general concept of activation results in fascinating program simplicity and in almost prohibitive housekeeping requirements in a conventional computer system. We will consider only the more restricted form which does not generate queues in this paper. Unfortunately, this restricts the programming use of activators almost to that of binary switches.

3.0 Use of Task Tree.

The use of this non-sequential task tree has all of the advantages of subprogram structures with the added flexibility that the programmer need specify sequential relationships only where they are known. Non-sequential relationships may be controlled by the

system. Because these relationships are recognized in the structure of the programming language, the programmer is relieved of the responsibility for them and the system is given the information it needs in order to handle them, thus simplifying everyone's job. Storage allocation and scheduling may be handled by the system. The programmer need not care when his routine is to be used or where.

3.1 Natural Description.

The consequent structure is well suited to describe what systems do whether they be corporations, job shops or programs. Indeed, this type of structure could be used to describe, in recognizable form, the operations of a business using programming structures which have their counterparts in administrative structures. The restriction to not permitting queues is similar to abolishing in and out baskets in an office. This restriction is almost unavoidable until we can get effective computer hardware for such "baskets".

The generalized procedure network is an effective model of the way human organizations operate and the degenerate form discussed here is a compromise which retains substantial power. This natural organization suggests the basic power and simplicity which is possible in such a computer system. Many of the burdens of organization and sequencing are assumed by the system. Because the language is designed for this, the system becomes relatively simple.

3.2 Language Structures.

We will consider only the language appropriate to the programmer. The language in which a user would make a request is simple but quite dependent on local boundary conditions. The language used in writing the system might be any which offers appropriate flexibility in controlling the specific computer.

The object program structure produced by a source language of the type under discussion is substantially different from normal usage and is closely integrated with the operating system. The source language itself is quite similar to conventional languages and is easily defined.

In FORTRAN, the addition of a subtask statement (similar to subroutine) and a call subtask statement (similar to call) will suffice. Because of the demand for Boolean variables, a FORTRAN version permitting such variables should be used. The ordinary call and subroutine statements could be retained if desired. Note that, although the new language is quite similar to the old, the method of use of the language statements is quite different.

In many respects, Algol forms an excellent vehicle for such non-sequential structures. In this case the procedure would be modified to include the activator and the subtask would replace both the block and the procedure concepts. This results in a much simpler language which retains the advantages of Algol and the power of the non-sequential structures.

3.3 Multilevel Control and Debugging.

Since all tasks have the same calling structure, any task may

be used as a main task or as a subtask. For example, one might have a task which controls some external equipment through its use of various subtasks. If the current control program is unsatisfactory for some special purpose, the user may assume control on a lower level of the task tree and determine his own sequencing of the rudimentary control functions. This is similar to the operation of a space satellite in the "fly by wire" mode.

A further advantage of this mode is in debugging. A task tree may be debugged from the end of its branches inward by calling each subtask as a main task and supplying the appropriate test parameters with his request. No special debugging main program need be written or debugged.

4.0 An Example of Usage.

In order to give an example of the usage of consequential procedures it is necessary to introduce a language in which to describe them. Although the syntax of the language used does not differ very much from conventional ones, the way in which the language is used is quite different and can best be appreciated by study of an example. The following casual definition of a language based on Algol may suffice for our purposes but is certainly not a sufficient definition for the design of a syntactic analyzer. The object program structures are dependent on a specific computer and are not discussed here. More details as to a language and system structure on a specific computer (the SDS 910) are to be found in the paper by McFarland, Fitzwater and Stewart 3 .

4.1 Consequent Language Description.

Let a consequent procedure have the form of an Algol procedure whose main program is the parent task. The procedure body is a block and contains no further blocks. A procedure statement is a subtask call with the following syntax:

$$\begin{aligned} \langle \text{Procedure Statement} \rangle &::= \langle \text{Procedure Identifier} \rangle \\ &\quad (\langle \text{Activator} \rangle; \langle \text{Actual Parameter List} \rangle) \\ \langle \text{Activator} \rangle &::= \langle \text{Boolean Variable} \rangle \\ &\quad | \langle \text{State of a Variable} \rangle \\ \langle \text{State of a Variable} \rangle &::= [\text{Variable}] \end{aligned}$$

The state of a variable (active or inactive) is a Boolean variable associated with each variable. We will place the further restriction that a procedure statement may not be used in a compound statement or in a simple or priority subtask.

The coding body of a simple task may be written in a machine language code which is otherwise unspecified.

Two pseudo blocks which are executed only once each may be included as the first and last statements in the body of a procedure. These are defined by enter...end or exit...end with enter or exit replacing a begin. An enter block is executed only on first entrance and in a non-priority mode. An exit block is executed just prior to the subtask completion exit and is always executed in a non-priority mode.

In order to simplify the language description given here, we will use descriptions of statements in place of the statement when

the explicit syntax for such a statement has not been given. Such descriptions will be placed between parenthesis to prevent confusion with comments.

4.2 Example Description.

The example chosen will illustrate the interactions of various priority and non-priority tasks. This example is of the control and logging of data from a simple external device. The same types of interactions, although there might be no need for priority tasks, would occur in a data processing application.

Let us suppose the external device has three asynchronous processes. Process one is to set a value of a variable and hold it during the remaining two processes. We may modify certain device parameters so as to cause the variable value to assume the desired value. The second two processes are very similar but are not sequentially related. We may initiate either process and at a later time (signalled by an interrupt) we may read in a vector of 3 values for each process. Computations using these values then produce a new setpoint for the first process which is set and held dynamically for the remaining two processes. The value of each setpoint is typed out and a wait, if necessary, is made to prevent stacking demands on the typewriter.

In case of failure, for any reason, to complete the above cycle within a specified time limit, we will assume some disaster has occurred, shut down the external device and remove the task tree from storage as an emergency measure. The system will so

notify the user.

The operator may change the value of the Boolean variable QUIT to true and cause the task tree to finish normally.

4.3 Coding for Example.

The user would request the task called measure and supply appropriate values for its parameters. The meaning of the parameters in MEASURE is as follows:

PA, PB	define the priority interrupt entrance. If an interrupt associated with PA occurs, the priority subtask associated with PA will be entered in the priority mode. If the subtask does not exist, it will be ignored.
LIMIT	is the integral number of clock interrupts which must be received before disaster is assumed.
I	is a counter for clock interrupts. It may be reset to zero at any time.
QUIT	is a Boolean variable which, if true, request cessation of operation.
D	is a range bound on the value of the variable C.
NA, NB	are the norms of the vectors A, B.
A, B	are vectors whose values are obtained from the external device.
C	is the value to be held in the external device by the first process.
READY	is a dummy Boolean variable denoting that the set-point has been reached.
SYSTEM	is a global label which returns control to the system scheduler.

The ENTER block will establish the priority subtasks HOLD and TIME which will remain active for the duration of the control task. This block is executed only on first entrance to the task MEASURE.

The subsequent entries to the task, MEASURE, will be made at the first statement following the enter block. Subsequent entries will be made at the completion of any subtasks of MEASURE.

The calling of the evaluate subtask will be considered only if its input data, NA and NB, are available and the last value of C has been logged.

The remaining subtask calls will be made if during execution of the task body (an activation scan) their activators are true. The activator for a subtask will be set false at completion of that subtask. Note that the order is not important since any order of the remaining statements will produce the same results.

Since the scheduling of the subtask calls is a function of currently available storage and of the occurrence of the priority interrupts the programmer does not know how much overlap of subtasks will actually occur on each measurement cycle. If storage is temporarily restricted, the subtasks would be executed one at a time in sequence. If the response to INPUT1 is very fast, NORM (A; A, NA) may be executed during INPUT2 and while TYPE is operating. The system will automatically take advantage of such overlapping and will give the best response, on each cycle, that is possible under the conditions then prevailing.

COMPLEX MEASURE (PA, PB, D, LIMIT, QUIT);

PRIORITY PA, PB; REAL D; INTEGER LIMIT; BOOLEAN QUIT;

BEGIN REAL NA, NB, C; REAL ARRAY A, B 1:3 ;

SUBTASK EVALUATE, HOLD, TIME, INPUT 1, INPUT 2, TYPE,

```

NORM; BOOLEAN P, READY;
INTEGER I;
ENTER C: = 0.0; C : = TRUE; READY: = FALSE;
      HOLD ([C]; C, READY, QUIT, I);
      TIME ([C]; LIMIT, I, QUIT);
END ENTRANCE BLOCK;
IF QUIT THEN GO TO SYSTEM;
IF NA AND NB AND C THEN P: = TRUE ELSE GO TO SKIP;
EVALUATE (P; NA, NB, C, D);
SKIP: NORM ([A]; A, NA);
      NORM ([B]; B, NB);
      INPUT 1 (READY; A, PA);
      INPUT 2 (READY; B, PB);
      TYPE ([C]; C);
END MEASURE;

```

The subtask HOLD is a clock priority routine. This implies that an entrance to the body of the subtask will be made in the priority mode each time a clock interrupt occurs. The ENTER and EXIT blocks are executed only once and in the non-priority mode. The variables declared in hold have the following meaning;

X is assigned the value of the external address dial.

DEAL is an address in the external device which selects the register holding the value to be assigned to X.

ERROR is the discrepancy between the setpoint and the current value of dial.

The ENTER block initializes the mode switch to hold and requests

an entrance on the next clock interrupt. This is executed in the non-priority mode. Subsequent priority mode entrances are made at the first statement after the ENTER block.

If the error is large, HOLD assumes that a new setpoint has been requested and READY is set FALSE. When the error has been reduced to an acceptable value, READY is set TRUE to initiate the appropriate subtask calls in the parent task. An activation scan of the parent task is requested to give the parent the opportunity to make such calls.

The normal exit from the priority mode section of HOLD is made by requesting another clock interrupt and releasing control.

This subtask will complete only when QUIT is TRUE by requesting its removal at a later time in the non-priority mode and releasing control to the interrupted coding. The system will then enter the coding body and execute EXIT block in the non-priority mode. This results in a return to the parent task and the destruction of this subtask.

CLOCK PRIORITY HOLD (A1, A2, A3, I);

REAL A1; BOOLEAN A2, A3; INTEGER I;

BEGIN REAL X; ERROR; EXTERNAL ADDRESS DIAL: = 1234;

SWITCH MODE, SET, HOLD ; INTEGER N;

ENTER N: = 2; (REQUEST CLOCK INTERRUPT)) END

IF QUIT THEN GO TO DONE;

(INPUT DIAL VALUE AND ASSIGN TO X);

ERROR: = X - A1; GO TO MODE N ;

SET: IF ABS (ERROR) < 0.01 THEN

```

BEGIN [A2] := TRUE; N: = 2; I: = 0;
COMMENT THIS RESETS THE TIME LIMIT CLOCK COUNT;
(REQUEST ACTIVATION SCAN OF PARENT) END

CORRECT: (INITIATE APPROPRIATE CORRECTIVE ACTION TO HOLD
          ERROR AT A MINIMUM); GO TO REPEAT;

HOLD: IF ABS (ERROR) > 0.5 THEN
      BEGIN [A2] := FALSE; N: = 1 END
      GO TO CORRECT;

REPEAT: (REQUEST CLOCK INTERRUPT);
        (RELEASE PRIORITY);

DONE: (REQUEST PRIORITY TASK EXIT);
      (RELEASE PRIORITY);
      EXIT (SHUT DOWN HOLDING ACTION) END

END HOLD;

```

The subtask TIME is a clock priority routine which counts clock interrupts. If the limit is reached emergency action is taken. This subtask is active during the operation of MEASURE. The variables used in time have the following meaning:

- LIMIT is the number of clock pulses which are allowed before assuming some uncontrollable disaster has occurred.
- I is a counter used for counting clock interrupts. It may be reset on each complete cycle of measure. This resetting is done in subtask evaluate.

This subtask does nothing but count unless QUIT is TRUE or LIMIT is reached. If QUIT is TRUE, the subtask completes in the same fashion as hold. If the limit is reached, something is beyond

control and the entire device is shut down and a request to eliminate the entire task tree is made of the system. When this is accomplished, the system will so notify the user.

CLOCK PRIORITY TIME (LIMIT, I, QUIT)

INTEGER LIMIT, I; BOOLEAN QUIT;

BEGIN

ENTER I: = 0; (REQUEST CLOCK INTERRUPT) END

IF QUIT THEN GO TO DONE;

I: = I + 1; IF I = LIMIT THEN

BEGIN (SHUT DOWN EXTERNAL EQUIPMENT);

(REQUEST EMERGENCY ERASURE OF TASK TREE);

GO TO OUT

END LIMIT REACHED;

(REQUEST CLOCK INTERRUPT);

OUT: (RELEASE PRIORITY);

DONE: (REQUEST PRIORITY TASK EXIT):

(RELEASE PRIORITY);

END TIME LIMIT;

The INPUT1 and INPUT2 subtasks are identical, at least to the degree of coding appropriate here, and only differ in the external device addresses which are used to obtain information. We will discuss only INPUT1 which uses formal parameters corresponding to actual parameters which have been discussed.

INPUT1 is a priority routine whose body is executed only once. The ENTER block sets up the priority entrance and initiates

the external process which will, at a later time, give the desired priority interrupt which signals that the values of A1 are ready to be read.

PRIORITY INPUT 1 (A1, A2);

PRIORITY A2; REAL ARRAY A1;

BEGIN

ENTER (REQUEST PRIORITY ENTRANCE A2);

(INITIATE ACTION WHICH GIVES PRIORITY ENTRANCE
A2 FROM INPUT 1)

END ENTRANCE SET UP;

(INPUT EXTERNAL VALUES OF A1 AND ASSIGN TO A1);

[A1] : = TRUE;

(RELEASE PRIORITY)

END INPUT 1;

PRIORITY INPUT 2 (A1, A2)

PRIORITY A2; REAL ARRAY A1;

BEGIN

ENTER (REQUEST PRIORITY ENTRANCE A2);

(INITIATE ACTION WHICH GIVES PRIORITY ENTRANCE
A2 FROM INPUT 2)

END ENTRANCE SETUP;

(INPUT EXTERNAL VALUES OF A1 AND ASSIGN TO A1);

(REQUEST PRIORITY TASK EXIT); [A1] : = TRUE;

(RELEASE PRIORITY)

END INPUT 2;

The remaining subtasks, EVALUATE, NORM and TYPE, are simple non-priority routines that are executed only once per call and return control to the parent task. The actual parameters used in the calls have been discussed and there is little more to say about these subtasks.

SIMPLE EVALUATE (A1, A2, A3, A4);

REAL A1, A2, A3, A4;

BEGIN A3: = (A1 + A2)/(A1+A2) + A3 + 0.5;

IF A3 > A4 THEN A3 := A4;

[A3]: = TRUE

END EVALUATE

SIMPLE NORM (A1, A2)

REAL A2; REAL ARRAY A1;

BEGIN A2: = (A1[1]² + A1[2]² + A1[3]²)^{0.5}; [A2]: = TRUE

END NORM;

SIMPLE TYPE (A1);

REAL A1;

BEGIN (TYPE VALUE OF A1) END TYPE

5.0 System Requirements.

A system of the structure described may be operated on many levels. It might be used as shown in the example within a single processor or it might be used in such a fashion that the activation of a subtask calls into play a whole subset to a man-computer-machine complex. Some of the subtasks might be performed by

people while other people controlled the overall tree structure by making appropriate requests of the system. In a multiprocessor system it is not necessary but it may be desirable to specialize one of the CPU to perform the allocation and scheduling operations. Such specialization could also allow queue control and the subsequent use of the more general activation concept previously discussed. A small associative memory would be advantageous for storage of activator parameters. Indeed, this latter organization is a very intriguing way to consider multi-processor systems. We will, however, confine our subsequent remarks to a single processor system.

We have implicitly considered that the single processor has a secondary (or multi-level) store which has a size sufficient to maintain total library control and transient data storage. We are not so concerned about transfer rates between primary and secondary storage so long as average rates are satisfactory. Critical response loops may be closed through priority routines which have been installed before the high speed loop is initiated.

The most significant demand on core space arises from currently active priority tasks which must occupy storage until their task is done in order to give short reflex times. If longer reflex times are suitable, small interrupt routine can activate a non-priority task to do the job.

Dynamic allocation of primary storage is a critical factor and can easily become the most time consuming part of the system

which otherwise has a very low duty cycle that is optimumly phased with the workload. It is still feasible to relocate subtasks as they are entered but it would be much more desirable if routines could run wherever they happen to be without relocation.

The system itself requires only a small amount of core storage for the control program and tables and may itself extend into subtasks for auxiliary operations or extension of facilities.

- 1 E. J. Schweppe and D. R. Fitzwater, to be published.
- 2 D. E. McFarland, D. R. Fitzwater, R. M. Stewart, presented at Fall SDS Users Conference (1963).
- 3 D. E. McFarland, D. R. Fitzwater, R. M. Stewart, to be published.
- 4 J. McCarthy, S. Boilen, E. Fredkin and J. C. R. Licklider, AFIPS Conference Proceeding 23, Spring Joint Computer Conference (1963).

END