THE UNIVERSITY OF MICHIGAN

COMPUTER AND COMMUNICATION SCIENCES

Final Report

REAL TIME OPERATING SYSTEM IMPLEMENTATION STUDY

Bruce W. Arden
Principal Investigator

James Hamilton
James Henriksen
Ronald Srodawa

ORA Project 037220

under contract with:

U.S. ARMY SAFEGUARD SYSTEM COMMAND
CONTRACT NO. DAHC60-71-C-0028
CONTRACTS OFFICE SSC-CS
HUNTSVILLE, ALABAMA

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

March 1971

TABLE OF CONTENTS

1.0 INTRODUCTION

The authors of this report have had the opportunity, to varying degrees, to observe the development of a general purpose multiprocessor computing system [Ref. 9] and a special purpose, high performance multiprocessor system [Ref. 4]. On first observation there seems to be little in common between these two types of computer systems and, in fact, they appear to have developed quite independently. However the general purpose multiprocessor with its necessarily dynamic scheduling of tasks and assignment of storage is demonstrably successful and it is now natural to ask whether some of the same techniques would be successful in the more specialized system.

If the development of a large software system falters one can look for trouble in two broad areas. The system design (i.e., choice of operating algorithms) may be poor or the implementation procedures inadequate—or, more probably, a subtle combination of both. For the purposes of this report a clear distinction between the two areas will be made.

Whenever system performance is a vital issue, as it is in general system supervisory programs and demanding real time programs, the often asked question of whether high level programming languages can be effectively used is still an issue. In an earlier report [Ref. 2] some comparative programming of system and application type modules was carried out with the general conclusion that the specialized code that can be obtained with assemblers still produces a substantial size and speed advantage. It is the purpose of Section 2 to examine in more detail the reasons for this somewhat discouraging result. The general approach taken is to examine in detail the structure of two System/360 programs, isolate the structural properties, and then determine how well such structures can be represented in higher level languages. It is an attempt to refine the program level comparison, to determine, although not completely in this report, where the mismatch occurs. The chosen two programs are representative of the system code in the general multiprocessor system. They are known to be well coded and effective and perhaps most important, their structure is known in detail.

The preceding paragraph describes a first step in determining the feasibility of certain system implementation procedures. The other question of the suitability of the operating system algorithms seems conceptually easy to state but surprising difficult resolve. It is not hard to give a general division of supervisor function which encompasses all systems from general—time—independent to specific—with—deadlines. The following is one such division.

1. Machine interface
2. Resource allocation to top priority tasks

3. Priority determination
4. Management of time independent sequence dependencies
5. Deadlines and alternatives

In the specialized cases where the nature of the tasks is known the allocation and priority assignment are subsumed under the general heading of scheduling. If the number of tasks is fixed and there are no sequence dependencies the refinements to job shop scheduling [Ref. 5] give an algorithm for determining an optimal schedule. Unfortunately, in real systems, neither premise is valid. Since there are working systems with these general inputs it is clear that working solutions to the scheduling problems have been found. The purpose of Sections 4 and 5 are to describe the solutions for two different systems.

The radar and missile control computer systems are an interesting intermediate case. In typical process control systems the tasks are known and fixed; in general systems the tasks are not known a priori and must be dynamically scheduled. The intermediate case is where the tasks are known but the number of instances of such tasks (or load) is highly variable. This comparison suggests that such systems might be successfully implemented with the dynamic approach.

It would be presumptuous to attempt a comprehensive analysis of the systems described in Sections 4 and 5 in so brief a study. The management of a beam-switching radar is a well defined and demanding real time load; and hence it was decided to investigate the overall structure of a successful system of this type—the AN/FPS-85 [Ref. 3]. Although it is a dual processor system it is not strictly a multiprocessor system and hence permits some simplifications. The multiprocessor SAFEGUARD system is, of course, extremely complex but an attempt was made to understand the overall structure. In both cases the emphasis of the study was to determine to what extent a priori scheduling was done and how much dynamic scheduling was included in the actual real-time dispatching of tasks. With the best results, clearly defined algorithms for these functions could be abstracted. However, in both cases, the a priori scheduling included some human judgment and experience which could not be expressed as an algorithm. If it were possible to study enough instances of such designs to develop an algorithm for these parts then perhaps some study of the approximation to optimality could be carried out.

2.0 System Programming Language Requirements

2.1 Analysis of System Programs

We begin our task of determining system program requirements by analyzing several typical system programs. More specifically, the programs are from the MTS operating system, which executes on the IBM System/360. This particular system has been chosen for analysis because the program listings are readily available to the author and it is representative of large-scale time-sharing systems.

In the analysis of these programs, several topics will be investigated for each program. These topics are (1) the function of the program and the manner in which it performs its function, (2) the environmental considerations involved in the program, including the calling sequence conventions used by the system to call the program and by the program to call other programs in the system, the method (or methods) used by the program to aquire and release dynamic storage, the method (or methods) used by the program to perform input/output operations, and the mechanisms available to the program to alter the processor scheduling algorithms applied to it, (3) the compile-time facilities required by the program, (4) the data structure requirements of the program, (5) the computational requirements of the program, (6) the program structure requirements of the program, (7) the control requirements of the program, and (8) the efficiency requirements of the program.

The particular programs which have been chosen for analysis are the following:

(1) The MTS Paging Drum Processor. This program possesses the most demanding set of requirements of all programs in the MTS system other than the supervisor. The program has fairly severe timing constraints in that it should be capable of keeping up with the transfer rate of the paging drums. The program must build channel programs for the paging drums and discs, request that the supervisor start the devices using the channel programs which the Paging Drum Processor has built, and process device-end and PCI interrupts from these devices in close to real time.

(2) The Computer and Communication Sciences 573 Supervisor. This program is a simplification of the MTS supervisor which has been developed as a pedagogical tool for teaching the internal structure of supervisors in the course CCS 573. The program requirements of this supervisor are essentially identical to those of the MTS supervisor. The program runs under a simulator for an imaginary machine which closely resembles a System/360. The imaginary machine differs from

the System/360 only in that the memory size is smaller and the input/output system is less complex. In fact, only the privileged instructions of this imaginary machine are simulated; all other instructions are directly executed by the System/360. This means that program running times under the simulator are very nearly identical to those of the same programs run on a System/360. The fact that this supervisor runs under a simulator is to our advantage because the simulator can gather timing information and there are none of the problems attendant with scheduling the use of real hardware for testing purposes. Thus, this program is an excellent choice for testing the efficacy of the system programming language which we will develop.

Following the detailed analysis of these system programs, we will summarize their composite requirements. From this summary we will characterize system programs and determine the necessary and desirable features which a language designed for system programming should include.

2.1.1 The MTS Paging Drum Processor

In MTS, the function of moving virtual memory pages between main storage and auxiliary storage is divided between the supervisor (UMMPS) and the Paging Drum Processor (PDP). The choice of which pages to read (transfer from auxiliary storage to main storage) and which pages to write (transfer from main storage to auxiliary storage) is made by UMMPS while the PDP constructs the channel programs to read and write pages on the auxiliary storage (currently drums and discs) and notifies UMMPS when a page has been read or written. This description of the PDP is a brief summary of information from Alexander [1].

The PDP program is executed by a single, absolute task within the UMMPS task environment. (An absolute task is one whose address space is precisely the main storage of the Model 67, as opposed to a relocatable task, whose address space is a virtual memory which is paged. Many system functions are run as absolute tasks, either (1) because by their nature they must be able to reference all of real storage, or (2) because the overhead inherent in relocating their channel programs and paging their private storage would lead to inefficiencies.) The PDP must be executed by an absolute task because if it were executed by a relocatable task and directed to write part of itself onto auxiliary storage, it might not be able to read itself back into main storage.

The most important unit of information communicated between UMMPS and the PDP is the Page Control Block (PCB), which describes the status of one virtual memory page. Each PCB is a member of at most one linked list (called a queue) of PCB's. These queues of PCB's are passed back and forth between UMMPS and the PDP. There are four such queues, (1) the Page-In-Queue (PIQ) which consists of the PCB's for all pages which have been requested to be read into main storage but which the PDP has not started reading yet, (2) the Page-In-Complete-Queue (PICQ) which consists of the PCB's for all pages which the PDP has completed reading but of which UMMPS has not been notified, (3) the Page-Out-Queue (POQ) which consists of the PCB's for all pages which are in main storage and which could be removed if necessary to make space for more pages, and (4) the Release-Page-Queue (RPQ) which consists of the PCB's for all pages which have been released by their owning tasks but which the PDP has not released yet.

The fact that the PDP program is executed by an absolute task complicates the passing of information between UMMPS and the PDP since a task can be interrupted between any two instructions, with control being passed to UMMPS by the interrupt. Thus the PDP cannot safely alter any information which UMMPS might also alter. In particular, the PDP cannot remove PCB's from the PIQ or the RPQ. To

avoid this problem and to communicate other information between UMMPS and the PDP, five SVC subroutines have been added to UMMPS for exclusive use by the PDP. These five SVC's are (1) Get-Queues (GETQS) which is used to return the PIQ and the RPQ to the PDP, (2) Get-Write-Pages (GETWP) which is used to request one or more pages from the POQ which will be removed from main storage, (3) Get-Real-Page (GETRP) which is used to request a main storage page into which to read a page that must be brought into main storage, (4) Free-Real-Page (FREERP) which is used to notify UMMPS that a main storage page that was allocated to a PCB is now available for reallocation and to notify the PDP that the page was reclaimed while being written by the PDP, and (5) PDP-Wait (PDPWAIT) which is used to notify UMMPS that the PDP temporarily has no work to do.

The Paging Drum Processor begins execution at its initialization entry-point when the task which is executing it is created. At this time the PDP acquires the drums and discs which it will use as auxiliary storage devices and initializes its data structures. The PDP then enters a cyclic behavior as described below.

At the beginning of each cycle, the PDP calls the GETQS SVC subroutine to acquire the current contents of the PIQ and the RPQ. For each PCB in the RPQ the PDP releases the corresponding page in auxiliary storage (if any) and frees the main storage making up the PCB. The PDP then puts each PCB in the PIQ onto a local PIQ corresponding to the auxiliary storage device on which the page is stored and the angle of rotation of the page on that auxiliary storage device. This is referred to as "slot sorting" or "sector queueing". The PDP then constructs one channel program per auxiliary device, the channel program being sufficient to perform a mixture of read and write requests during one rotation of the auxiliary device. Channel program construction begins by calling the GETRP SVC for each PCB which is at the head of a local PIQ for the device. For each successful GETRP call, the appropriate commands are built at the corresponding slot index in the channel program to read the page into the allocated main memory page. Then the GETWP SVC is called to acquire one PCB for each still-vacant slot in the channel program and the remainder of the channel program is filled with commands to write these pages. UMMPS will refuse to allocate a page on a GETRP call if main storage is nearly full. In this case the remainder of the channel program will be filled with write requests. Likewise, UMMPS will refuse to return PCB's on a GETWP call if main storage is fairly empty. In this case, the remainder of the channel program will remain vacant.

Following the completion of the channel program, the PDP determines if an operation is in progress on the auxiliary device. If there is, the new channel program is added to the end of the last channel program constructed for

the device so that it will continue with the new channel program when it has completed all previous channel programs. On the other hand, if the device is not running it is started by issuing a SIO SVC. In either case the device will cause a PCI interrupt at the end of each revolution and a device-end interrupt when it stops running.

After channel programs have been built for one revolution of each auxiliary device, the PDP performs the cycle again. When either there are no more read or write requests pending or channel programs have been built for the next three revolutions of each auxiliary device the PDP calls the PDPWAIT SVC which causes UMMPS to defer scheduling the PDP task on the CPU until more read or write requests arrive or an interrupt must be processed by the PDP.

Whenever any of the auxiliary storage devices completes a revolution or reaches the end of the channel programs presented to it, the PDP is entered at one of two entry-points. The PDP then processes each entry in the channel program which the device has finished. For those pages which have been read into main storage, the PDP places the PCB on a local PICQ, which will eventually be placed on the supervisor PICQ. For those pages that have been written onto auxiliary storage, the PDP calls the FREERP SVC to indicate to UMMPS that the main storage page is available for reallocation. If the page has been reclaimed or released while being written, UMMPS returns an indication of this and the PDP frees the auxiliary storage just written. After processing each completed entry in the channel program the PDP returns control to the point of the interrupt, if the PDP was in the cycle at the time of the interrupt, or to the beginning of the cycle, if the PDP had called the PDPWAIT SVC.

There are two further considerations which complicate the interrupt processors. First, if the operation on the auxiliary device terminated in an error, the PDP must call the standard direct-access device error recovery routine to retry the operation. second, there are certain sensitive sections of the PDP cycle which cannot be interrupted. If an interrupt occurs in these sections the fact that the interrupt occurred is noted and control is returned to the point of the interrupt. At the end of the sensitive section a test is made for interrupts noted while in the sensitive section. Any interrupts which were noted are processed at that time.

ENVIRONMENTAL CONSIDERATIONS

The Paging Drum Processor runs in the absolute task environment as defined by UMMPS. As such, the PDP can

execute all non-privileged instructions of the System/360 and reference or change all of main storage. Regions of main storage can be acquired by the PDP for its exclusive use through a variety of SVC subroutines. Input/Output operations and the enabling of input/output interrupts also are requested by the PDP through SVC subroutines.

Calling Sequences

The Paging Drum Processor is called at its initialization entry-point with a non-standard calling sequence. General-Register 15 contains the address of the entry-point, General-Register 2 contains the address of a vector of fixed-length character items of length 4, and all other register contents are arbitrary. The PDP must return via an SVC, rather than by transferring control back to the routine which called it. This is typical of programs which are entered by the creation of a task.

The Paging Drum Processor must call the standard direct-access device error recovery routine to retry operations which abnormally terminate on auxiliary devices. This error recovery routine also follows non-standard calling sequence conventions. General-Register 15 contains the address of the entry-point of the error recovery routine, General-Register 13 contains the address of a control block, General-Register 0 contains the logical device number of the device on which the error was detected, General-Register 1 contains the address of the channel program, and General-Register 2 contains the address of another control block. The direct-access device error recovery routine will never return control to the instruction following the call upon it. Control will be transferred to one of several entry-points whose addresses are specified in the control blocks. If the error condition is fatal, control is transferred to an instruction contained in one of the control blocks. In all cases, all registers are restored to their contents at the time of the call.

The Paging Drum Processor must use the SVC subroutines provided by UMMPS to obtain supervisor services. These subroutines are called by issuing an SVC instruction. Arguments are generally passed in general-registers 0, 1, 2, and 3. Results are returned in the same registers and in the condition code. Although most SVC subroutines return to the point of call, others (for example, PDPWAIT) return control to some entry-point specified as an argument. Still others enable interrupts in such a manner that the occurence of some asynchronous event will cause the PDP to be interrupted with control being passed to the entry-point of the appropriate interrupt routine at that time. Specific supervisor services obtained by the PDP will be discussed in the remainder of this section.

The Paging Drum Processor is called at one or the other of its two interrupt routine entry-points whenever a PCI or device-end interrupt is received from one of its auxiliary devices. General-Register 0 contains the right-hand half of the PSW of the PDP at the point of the interrupt or zero if it had executed the PDPWAIT SVC. General-Register 1 contains the address of the interrupt control block. General-Registers 3 through 15 and the floating-point registers contain their contents at the time of the interrupt and must be restored when the interrupt routine returns. The interrupt control block contains the device status byte from the Channel Status Word stored at the time of the interrupt plus the contents of General-Registers 0, 1, and 2 at the time of the interrupt. The interrupt routine may return to the point of the interrupt by executing one of two SVC's, one simply returns, the other also enables the interrupt again. (The interrupt is automatically disabled at the time of the first interrupt.) It also is possible to return from the interrupt to some point other than the point of the interrupt by using a third SVC. Finally, if the interrupt routine does not intend to return at all, a fourth SVC causes the CPU status saved by UMMPS as a part of the interrupt to be released.

Dynamic Storage Allocation

The Paging Drum Processor allocates and frees three different types of dynamic storage.

First, dynamic storage is allocated to the PDP by the GTBUF SVC and freed by the FREEBF SVC. This storage is automatically released when the task which is running the PDP is terminated, either abnormally or normally. The PDP uses this mechanism to acquire storage into which to read information from each auxiliary disc during initialization. This storage is released at the end of initialization, and is the normal type of dynamic storage which is allocated to tasks.

Second, dynamic storage is allocated to the PDP by the GETSC SVC and freed by the FREESC SVC. This storage is allocated from the areas used by the supervisor for its own dynamic storage needs and is never implicitly released. The PDP uses this mechanism to acquire storage for its device data structures, one data structure item for each auxiliary device. Since this storage is not released when the task running the PDP is terminated, this information is not lost if the PDP should terminate. Thus, a new task can be created and the PDP will continue. The PCB's also are allocated by UMMPS in supervisor storage. Thus, the PDP uses the FREESC SVC to release PCB's after processing from the RPQ.

Finally, dynamic storage is allocated to PCB's and freed from PCB's through the GETRP and FREERP SVC's. This memory is used to contain the user's virtual storage page when it is located in main storage. The PDP allocates a page to the PCB prior to reading the page into main storage and frees the page from the PCB after writing the page onto auxiliary storage.

Input/Output

The Paging Drum Processor uses two distinct supervisor services for input/output.

First, the PDP types information to the operator on the operator's console through the WTO SVC. This supervisor service requires only the address and length of the message.

Second, the PDP performs input/output operations on the auxiliary devices through the SIO SVC. This supervisor service requires the address of the channel program, the logical device number of the device, and the address of a control block used during error recovery. The PDP generally enables device-end and PCI interrupts through other SVC's, as described above under calling sequences. At other times the PDP issues the WAIT SVC, which causes execution of the PDP to be suspended until the device-end interrupt is received from the device.

The fact that the exact channel program presented to the auxiliary device is specified when using the SIO SVC is used to great advantage by the PDP. By taking advantage of this the PDP is able to link new channel programs to the end of old channel programs while an auxiliary device is running and process PCI and device-end interrupts in a meaningful way.

Other SVC's are used to acquire and release devices, obtain sense data from a device when errors occur, discover the address in the channel program of the last command executed by the device, and manipulate the UMMPS data structure for the device as a part of error recovery attempts.

CPU Scheduling

The Paging Drum Processor uses two SVC's to affect the UMMPS CPU scheduling policies.

First, the PDP issues the DORMANT SVC to relinquish the remainder of its time-slice during initialization when it is attempting to acquire an auxiliary device which some other task owns for the moment. A contention problem develops because several programs, including the PDP, acquire and

release each disc in succession until they find the discs whose labels match those which they are to use.

Second, the PDP issues the PDPWAIT SVC when it temporarily has no work to do. This causes UMMPS to defer scheduling the PDP task on a CPU until more read or write requests arrive or an enabled interrupt occurs on one of the auxiliary devices.

COMPILE-TIME FACILITY REQUIREMENTS

Inclusion of Data Structure Definitions

The PDP copies the definition of the PCB data structure from a file. This file allows both UMMPS and the PDP to reference a common definition of that data structure.

Compile-Time Variables

The PDP requires several compile-time variables whose values affect the generation of code. For example, two compile-time variables have as their values the maximum number of discs and drums, respectively. Other compile-time variables have as their values the numbers associated with the SVC subroutines and the operation codes for channel programs. The value of a compile-time variable should be accessible in other compile-time statements and in place of constants in normal statements. For example, the PDP uses compile-time variables to designate the number of entries in some data structures.

Compile-Time Expressions

The PDP requires that simple arithmetic and Boolean expressions involving constants and compile-time variables be evaluated at compile-time. For example, the sum of the two compile-time variables which represent the maximum number of discs and drums is frequently used to represent the maximum number of auxiliary devices. The compile-time variable which represents the maximum number of discs is frequently compared to zero.

Compile-Time Transfers

The PDP requires compile-time transfer instructions capable of skipping over sections of the source program, either conditionally or unconditionally. This capability is used to ignore those portions of the PDP which are only

required for discs if the maximum number of discs is zero. This results in a shorter program if no discs are to be used.

Compile-Time Procedures (Macros)

The PDP requires compile-time procedures (sometimes called macros) of sufficient power to generate one or more statements. It must be possible to pass arguments to the procedure which will be substituted for the corresponding parameters in the procedure definition. It must also be possible to perform tests for the value of a parameter (including the existence of its corresponding argument) and to use the results of such tests to influence the flow of control through the procedure. Local variables and the ability to generate unique identifiers also must be available within the compile-time procedures.

Attributes of Data-Items

Some attributes of data-items must be accessible as compile-time values. For example, the lengths of data structures and character-strings are frequently required.

DATA STRUCTURE REQUIREMENTS

Homogeneous Structures

The Paging Drum Processor requires structures of homogeneous elements (vectors and arrays) where the elements of the structure can either be simple elements or structures. The elements of the structure are referenced by subscription. Example structures of homogeneous elements used in the PDP are vectors of fixed-length character-strings, vectors of pointers, and vectors of channel programs (which themselves are vectors of non-homogeneous structures).

Non-Homogeneous Structures

Structures of non-homogeneous elements where the elements of the structure can either be simple elements or structures are needed by the PDP. The elements of a non-homogeneous structure are referenced by name. Example structures of non-homogeneous elements used in the PDP are the PCB and the channel command.

Alternatives

Structures of non-homogeneous elements where all the elements of the structure are assigned to the same storage (often called alternatives or overlays) are required by the PDP. Again, the alternatives can be simple elements or structures and are referenced by name. The storage allocation schemes used for the alternatives should be straightforward so that the programmer can easily overlay two non-homogeneous elements in a useful way and understand the allocation of the bits in common. For example, the PDP requires that some fields eight bits in width sometimes be considered as one eight bit wide field and at other times be considered as eight fields, each one bit wide. The auxiliary address field of the PCB is treated as three fields of lengths 3, 5, and 8 bits, repectively for drums, and as two fields of lengths 3 and 13 bits, respectively, for discs.

Data-items and structure elements are defined to be the same structure so frequently that some provision should be made to allow a structure to be treated as an entity, much like a data-item.

Storage Allocation

It must be possible to cause data-items to be allocated either in main storage or in hardware registers. In the case of hardware registers, it must be possible to either specify the register to be used or allow the compiler to choose one of a class of registers. In the PDP certain data-items must reside in specific general-registers, for example, arguments to the SVC's. One approach to this problem is to allow the user to specify that certain variables are to be allocated to specific registers. An assignment of a value to the variable causes the contents of the register to be set. A reference to the value of the variable causes the register to be referenced. Specifying a variable which is allocated to a register as a parameter might be used to indicate that the register contains the value of the parameter at the time of the call. This might be used to describe the unusual parameter passing arrangement used at the initialization entry-point of the PDP.

Storage Class

The Paging Drum Processor requires data-items having the storage class attributes static, based, and parameter. By static, we mean that the data-item is allocated to a fixed main storage address throughout the program. The main storage address is specified at the time the program is loaded. By based, we mean that the data-item is dynamically

allocated to main storage while the program is running and is referenced indirectly through a pointer to the allocated storage. Parameters are data-items which are associated with arguments at the time a procedure is called. Based data-items are used for PCB's and all data structures pertaining to the auxiliary devices in the PDP. Parameters are used in several miscellaneous internal procedures. Static allocation is used for all other variables.

Scope and Ownership

The Paging Drum Processor requires data-items having internal and external scope. By internal scope we mean that the data-item is not known to other programs unless passed as an argument. On the other hand,a data-item of external scope is known to all other programs in which it is declared to be external. The PDP furthermore requires that the owner of static external data-items be specified as either the PDP or some other program. For example, most data-items used in the PDP are internal; they are known only in the PDP and their storage, provided that they are of static storage class, is allocated within the PDP program. Some data-items, such as the entry-point of the direct-access device error recovery routine and the PICQ head, are external and owned by programs other than the PDP. Still other data-items, such as the entry-point of the PDP, are external and owned by the PDP.

Length, Alignment, and Representation

The length, alignment, and representation of data-items need to be specified precisely in the PDP. The length should be specifiable in bits, alignment in the offset from some modulus. The representation is usually chosen from one of the possible representations accepted by the hardware, but this is not always the case. For example, a fixed-point fullword integer can be accurately described as a data-item 32 bits in length, with alignment 0 modulo 32, and a two's-complement representation. The PDP has several unusual data-items, such as the "slot number" field in the PCB, which is 5 bits in length, with alignment 3 modulo 16, and an unsigned binary magnitude representation.

Pointers

The Paging Drum Processor requires data-items which are pointers; addresses of some other data-item. The PDP has pointers which are 12, 24, and 32 bits in length, with alignments of 4 modulo 16, 8 modulo 32, and 0 modulo 32, respectively. All pointer values are represented as unsigned binary magnitudes, where the value is interpreted as the main storage address of the item which the pointer addresses (in bytes). Some pointers are scaled by 2^{12}

(4096), that is, the value of the pointer must be multiplied by 2^{12} to obtain the main storage address in bytes of the addressed item. This is used to store addresses which are page-aligned (alignment of 0 modulo 2^{12}) within the PCB's.

Bit-Strings

The PDP requires data-items which are bit-strings. The PDP uses bit-strings which are 1, 4, 8, 100, and 6500 bits in length with alignments of 0 modulo 1 and 0 modulo 8. All PDP bit-strings are fixed in length.

Integers

The PDP uses data-items which are integers. The PDP has integers which are 3, 5, 8, 13, 16, and 32 bits in length with several different alignments. The representations which are used for integers are unsigned binary magnitude, two's-complement, and packed decimal.

Machine Instructions

The PDP requires that a machine instruction be generated as the value of a data-item. This is required as a part of the control blocks passed to the direct-access device error recovery routine. The register contents at the time the instruction is executed are as they were at the time the error recovery routine was called. This fact eases the addressability problems associated with the instruction.

Character-Strings

The PDP requires character-string data-items. In all cases the character-strings are a multiple of eight bits in length aligned on a 0 modulo 8 boundary. Each eight bit group of bits (called a byte) represents one character using the EBCDIC representation. All character-strings used in the PDP are less than 256 bytes in length and each has a fixed length.

Initialization of Data-Items

The Paging Drum Processor must be able to initialize all the data-items which it owns, no matter what their storage class. Static items should be initialized at the time the PDP program is loaded. Based items should be initialized at the time the item is allocated.

A variety of external representations should be allowed as initialization values. In particular, numeric

representations should be allowed in any meaningful base, such as 2, 10, and 16 for the PDP. Compile-time expressions must be accepted both as initialization values and as multiplicity factors.

Pointers must be able to be initialized to the address of any data-item known to the program, including the address of an instruction and the address of some other data-item in the same based structure. Structures must be able to be initialized on an element-by-element basis.

Neighborhoods of Data-Items

It might be useful, for the sake of optimization, to specify the neighborhood of a data-item. For example, the PDP stores page-aligned pointers as the low-order 12 bits of a halfword. Thus, the pointer has a length of 12 bits and an alignment of 4 modulo 16, and is represented as a binary magnitude with a scale factor of 2^{12}. Such an item can be converted to a 24 bit address on the System/360 with the code sequence:

```
LH    R,ADR
SLL   R,20
SRL   R,8
```

or the sequence:

```
LH    R,ADR
SLL   R,12
N     R,=X'00FFF000' .
```

In fact, the top four bits of the halfword containing the pointer are always zero. Taking advantage of this fact allows the pointer to be converted to a 24 bit address using the sequence:

```
LH    R,ADR
SLL   R,12 .
```

One solution might be to define the primitive (non-structure) modes themselves as data structures for which certain-named components indicate the location and representation of the value.

COMPUTATIONAL REQUIREMENTS

Assignment

The Paging Drum Processor requires assignment of values to data-items of the same mode as the value. Specifically, the PDP requires assignment of integers to integers, bit-strings to bit-strings, character-strings to character-strings, and pointers to pointers.

The Paging Drum Processor also requires the assignment of structures to identical structures and a scalar value to all the elements of a homogeneous structure. The former is used to assign a transfer command to the end of the channel program when appending one channel program to another. The latter is used to set all the elements of a vector to zero.

Arithmetic Operations

The PDP requires addition of pointers and integers, in all four possible combinations. Shifting, division, and modulo are required with integer operands.

Relational Operations

The Paging Drum Processor requires the four relational operations "equal", "not equal", "greater than", and "less than". Required mode combinations are integer to integer, character-string to character-string, and pointer to pointer.

Boolean Operations

The PDP requires the Boolean operation "disjunction". The Boolean operations must be defined for operands of bit-string mode. It is assumed that the result of a relational operation can be considered a bit-string.

String Operations

The Paging Drum Processor requires that a sub-string of any bit-string or character-string be allowed as an operand of any operation which accepts string operands, including as the left-hand side of an assignment. An operation which scans for the first zero (or non-zero) bit in a bit-string and returns its position in the string is required.

Miscellaneous Operations

The PDP requires an operation which has as its value the address of its operand.

PROGRAM STRUCTURE REQUIREMENTS

The Paging Drum Processor is a program which has three entry-points, its initialization entry-point, an entry-point which is called for each PCI interrupt from the auxiliary devices, and an entry-point which is called for each device-end interrupt from the auxiliary devices. The two interrupt entry-points are called asynchronously and it is quite possible that the program is interrupted while executing the main body of code or the other interrupt routine. All three routines can optionally return. The initialization entry-point returns by calling an SVC to stop the task which is running the PDP. The other two entry-points return by calling an SVC which restores the task status to what it was at the time of the interrupt.

The Paging Drum Processor has a few internal procedures which perform miscellaneous computations. Most of these procedures return results and are referenced within expressions. All are passed arguments.

As has been noted earlier, the Paging Drum Processor dynamically allocates and frees based storage. There are three different types of dynamic storage which are used by the PDP. In all cases storage is acquired and released by calling SVC subroutines.

CONTROL REQUIREMENTS

Unconditional Transfer

The Paging Drum Processor uses unconditional transfer statements extensively. The destination of the transfer is always a labelled statement within the PDP program.

Conditional Statement

The Paging Drum Processor requires a conditional statement similar to the IF statement of PL/I or the Logical IF statement of FORTRAN.

Iteration Statement

The Paging Drum Processor contains several loops which could be effectively described with appropriate iteration statements. Most loops consist of the type which are adequately expressed in PL/I ("DO I=1 TO N") or FORTRAN ("DO I=1,N"). The PDP also contains some loops which iterate down a linked-list of data structures (in this case PCB's)

or around a circular-list of data structures (in this case the data structures associated with the auxiliary devices), where the instructions contained in the loop are executed once for each element in the list.

Calls

As has been indicated before, the PDP must call several different routines. These routines are called with radically different calling conventions. The supervisor (UMMPS) is called through the SVC instruction, the direct-access device error recovery routine is called with a non-standard calling sequence, and several internal routines are called with normal calling conventions.

Linked-List Maintenance

The Paging Drum Processor maintains several one-way and circular linked-lists. Items must be added and deleted from these linked-lists. Either these operations should be provided as primitives, or the necessary operations on pointers should be available so that these list operations can be coded in terms of those operations.

Pause

The Paging Drum Processor voluntarily ceases to use the CPU when it temporarily has no useful work to perform. These pauses are effected by calling one of two SVC subroutines, DORMANT or PDPWAIT.

EFFICIENCY CONSIDERATIONS

Since there are no deadlines in the MTS system and an ample amount of main storage, there are no concrete upper bounds on the efficiency required of the Paging Drum Processor. This does not mean, however, that there is no need for efficiency. It simply means there is no good measure of it. The performance of critical programs such as the Paging Drum Processor does have a significant effect on the performance of the total system.

Time Considerations

An upper bound can be computed for the amount of processor time which the Paging Drum Processor should use. Pages are formatted on the auxiliary drums in such a manner that one through nine pages can be read or written each two

rotations of a drum. Since the rotation time of a drum is 17.5 milliseconds, the total transfer time is 35 milliseconds per channel program. Thus, the Paging Drum Processor must use less than 35/n milliseconds of processor time per channel program (where n is the number of drums) in order to keep up with the drums. Discs need not enter into the computation since their transfer rate is significantly lower than that of the drums. In reality, the Paging Drum Processor must use significantly less processor time than this upper bound indicates, so that some processor time remains for other programs in a one-CPU system. Even in a multi-processor system, the PDP must use much less processor time than this upper bound suggests if it is to be called efficient.

Measurements have been taken of the performance of the current PDP program. A multi-linear regression analysis of these measurements is being attempted to fit the processor time required by the Paging Drum Processor to the equation

$$P = A+BT+CU+DV+EW+FX+GY+HZ$$

where A,B,...,H are the regression coefficients, P is the processor time required by the PDP, T is the number of read operations performed, U is the number of write operations performed, V is the number of channel programs built, W is the number of times the program cycle has been executed, X is the number of pages which have been processed from the RPQ, Y is the number of pages processed from the PIQ which did not require read operations (this was their first reference), and Z is the number of pages processed from the POQ which did not require write operations (they had not been changed since they were last written). In a similar fashion, the processor time required by the supervisor to service the Paging Drum Processor is being fitted to the equation

$$S = A'+B'T+C'U+D'V+E'W+F'X+G'Y+H'Z.$$

These regression analyses have not been completed at this time.

From these regression equations we will be able to determine tha amount of processor time which the PDP requires to construct one channel program, in the worst case. Likewise, we will be able to determine the combined processor requirements of the PDP and the supervisor to construct one channel program, in the worst case. We then will be able to compare these figures with the transfer time of the drums.

Storage Considerations

The storage considerations of the PDP efficiency are not as critical as the timing considerations. The current PDP program requires approximately 7130 bytes of main storage for the program, plus approximately 1930 bytes of main storage for the data structure associated with each auxiliary device.

The Current PDP as a Standard

Perhaps the best standard against which to measure the efficiency of a reformulation of the Paging Drum Processor in a higher-level language is the current PDP. The current program was written and designed by two highly-competent system programmers, each with several years of experience coding for the System/360. The code itself is very efficient and uses the full repertoire of machine instructions, as needed. Any compiler-produced code (or man-produced, for that matter) can be expected to perform no better than the current PDP program. Thus, the efficiency of the code produced by a higher-level language compiler can be measured by comparing the performance of the code produced for the Paging Drum Processor as expressed in that language with that of the current PDP program. Although the timing and storage considerations of the compiler-generated code should ideally be as close to those of the current PDP program as possible, in reality some degradation can be expected and tolerated.

2.1.2 The Computer and Communication Sciences 573 Supervisor

The Computer and Communication Sciences 573 Supervisor is a simplification of the MTS supervisor (UMMPS) which has been developed as a pedagogical tool for teaching the internal structure of supervisors. The students taking this course are required to write a supervisor program which meets the specifications of the Computer and Communication Sciences 573 Supervisor. The program which we have analyzed is the solution to this problem coded by one of the instructors for the course. The function of this program is to provide a time-sharing environment for up to fifteen tasks on an imaginary machine which closely resembles a System/360. Each task owns one input/output device, owns one region of main storage, and executes a shareable program. The input/output device and program to be used by the task are specified by the operator at the time he requests that the task be created. The main storage region is also allocated to the task at the time it is created. To perform this function the 573 supervisor processes all of the imaginary machine's hardware interrupts (which are identical to those of the System/360), maintains tables indicating the status of each input/output device and each task, schedules the use of the system resources (input/output system, processor, and main storage), provides supervisor services to tasks through SVC subroutines, acts upon commands from the operator, and maintains a log on the operator's console showing the creation and termination of each task. The term "supervisor" in this subsection always refers to the Computer and Communication Sciences 573 Supervisor.

The supervisor begins execution at its initialization entry-point. The supervisor then initializes all of its data structures, including the new PSW's which are used by the hardware to effect interrupts. At the end of initialization, the supervisor enters the hardware wait state until an interrupt occurs.

At each interrupt, the supervisor stores the processor status into its data structure associated with the task which was executing on the processor (if any) and then continues to process the interrupt. After the interrupt is processed, the supervisor enters a scheduling algorithm. This algorithm chooses one of the tasks which are ready to use the processor, restores the processor status from the data structure associated with that task, and thus relinquishes control of the processor to that task until the time of the next interrupt. If no task is ready to use the processor, the supervisor enters the hardware wait state until an interrupt occurs. From the time of the interrupt until the supervisor either enters wait state or relinquishes control of the processor to a task, the processor is run with interrupts disabled. Any external or input/output interrupt which occurs in this period of time

is deferred until the interrupts are again enabled at the end of this period. The processing performed for each type of interrupt is explained in more detail in the next several paragraphs.

An external interrupt occurs whenever the hardware interval timer is decremented from a non-negative value to a negative value. The supervisor places the maximum amount of time a task will be allowed to use the processor (called its time-slice) into the interval timer prior to relinquishing control of the processor to the task. At every interrupt the supervisor stores the contents of the interval timer, which is the amount of time remaining in the task's time-slice, as a part of the processor status of the task. Thus, a negative time-slice value signals that the task used up its full time-slice. This signals the scheduling algorithm to give this task a new (non-negative) time-slice value and to defer running this task again until all other tasks which are ready to use the processor have been run. If the external interrupt occurs while the supervisor is processing another interrupt, the external interrupt is deferred until the supervisor enables the interrupts. In this case, the external interrupt may in fact not correspond to the task which has just been dispatched. Relying on the contents of the interval timer rather than on the fact that an external interrupt has occurred to mark the end of a time-slice will prevent this spurious external interrupt from prematurely ending the time-slice of the task just dispatched.

An SVC (supervisor call) interrupt occurs whenever a task executes an SVC instruction. The operand of the SVC instruction is stored as the interrupt code in the old SVC PSW and indicates which SVC subroutine was called. There are nine SVC subroutines (0) END which causes the task to be terminated and all resources which it owns to be released, (1) DORMANT which relinquishes the remainder of the task's time-slice, (2) SIO which causes an input/output operation to be started on the device owned by the task with the channel program specified by the task, (3) WAIT which causes the task to be ineligible for use of the processor until the last input/output operation started on its device has stopped and returns the device status as of the end of the operation, (4) PROTOFF which causes the task to be run on the processor with the storage protection mechanism disabled, (5) PROTON which causes the task to be run on the processor with the storage protection feature enabled, (6) WEVENT which causes the task to be ineligible for use of the processor until another task causes an event, (7) CEVENT which causes an event which some other task is waiting for or will later test through the WEVENT SVC, and (8) REVENT which resets an event and terminates any task which is waiting for the event. The supervisor enters the appropriate SVC subroutine and performs whatever processing need be accomplished. In most cases the processing amounts to making slight changes to the supervisor data structures.

In the case of the SIO SVC, the supervisor must issue an SIO instruction to start the input/output operation on the device owned by the task.

A program interrupt occurs whenever an error is detected during the execution of an instruction by the processor. A program interrupt which occurs while the supervisor is executing indicates that an error has been detected in the supervisor. In this case the supervisor enters wait state with all interrupts disabled. The operator can then perform whatever steps are required to obtain a diagnostic dump of the supervisor. On the other hand, a program interrupt which occurs while some task is in control of the processor indicates that an error has been detected in the program which that task is executing. The task is stopped and diagnostic information is printed on the operator's console.

A machine-check interrupt occurs whenever a hardware malfunction is detected. The supervisor enters wait state with all interrupts disabled. The operator can then perform whatever steps are required to obtain diagnostic information concerning the hardware malfunction.

An input/output interrupt occurs whenever the input/output system desires to communicate status information to the supervisor. This happens whenever an input/output device stops running (either normally or abnormally) or an asynchronous attention condition is signalled by an input/output device. The address of the input/output device for which the status is being presented is stored as the interrupt code in the old input/output PSW and the status information is deposited in the Channel Status Word (CSW). At every input/output interrupt the supervisor makes changes to its data structures pertaining to the device addressed by the interrupt. These changes include copying pertinent information from the CSW. If the input/output device is owned by a task then these changes will cause the task to become ready to use the processor if it is waiting for its device to stop running and the CSW indicates that the device has stopped. If the input/output device addressed is the operator's console the supervisor (1) starts an input/output operation to read a command from the operator's console if the CSW indicates an attention condition was signalled by the operator, (2) acts upon the command received if the reading of an operator command has terminated, and (3) starts an input/output operation to type the next line of the log if the operator's console is now idle and the operator has not signalled that he wishes to enter a command.

Commands from the operator are interpreted as requests to create tasks. Each command line consists of a command name followed by the name of an input/output device. The supervisor processes a command by creating a task. The

entry-point of the program to be executed by the task is found in a table using the command name as a key. The input/output device referenced in the command becomes the device owned by the new task. A region of main storage is allocated to the task for its use from a pool of available regions. The newly created task is then marked as ready to use the processor.

ENVIRONMENTAL CONSIDERATIONS

The supervisor runs in the supervisor state environment of a simulated machine which is very similar to a System/360. The supervisor can execute all privileged and non-privileged instructions of this machine, which are identical to the non-privileged instructions of the System/360 plus the four privileged instructions Load PSW (LPSW), Start Input/Output (SIO), Set Storage Key (SSK), and Insert Storage Key (ISK).

Calling Sequences

The supervisor is called at its initialization entry-point by the bootstrap loader when the loading of the supervisor and the task programs has been completed. This call is nothing more than a direct transfer; the contents of all registers are arbitrary, the processor is in supervisor state, and all interrupts are disabled. The supervisor never returns, but rather keeps overall control of the hardware system from this point of time on.

The supervisor is "called" at one of its five other entry-points whenever a hardware interrupt occurs, each of the five types of hardware interrupts corresponds to one of the five entry-points. The calling sequence conforms to the action taken by the hardware at the time of the interrupt; the PSW is stored at the appropriate main storage location, the new PSW is taken from the appropriate main storage location, and a single argument (the interrupt code) is stored in the appropriate main storage location. The contents of all registers are arbitrary. The new PSW contents indicate that the processor is to run in supervisor state with the storage protection feature disabled and with further interrupts disabled. The supervisor then saves the entire status of the processor at the time of the interrupt into a data structure pertaining to the task which was in control of the processor at that time (if any) to complete the calling sequence. The supervisor must take advantage of the fact that the first 4096 bytes of main storage (called the PSA) can be referenced without a base register in order to save the processor status, since all register contents are arbitrary at the point of the interrupt. Eventually,

the supervisor will "return" to the task which was interrupted by restoring the complete processor status. Again, the fact that the PSA can be addressed without a base register is required to restore all of the processor status. The hardware is always put into the problem state with all interrupts enabled, with the storage protection feature enabled (unless a PROTOFF SVC has been issued by the task), and with a small, positive quantity (the time-slice) in the interval timer when the supervisor returns to a task.

A newly created task is inserted into this general scenario by initializing a data structure pertaining to it in such a manner that when the supervisor "returns" to it the processor status is restored so that execution continues at the entry-point of the program which the task is to run, General-Register 1 contains the address of the main storage region allocated to the task, and the other register contents are arbitrary. Since it is possible that the supervisor "returns" to a task which has never "called" it we actually have a co-routine structure rather than a strict subroutine structure. The fact that the supervisor sometimes "returns" to a task other than the one which was most recently interrupted also requires a co-routine type structure to express.

Dynamic Storage Allocation

The supervisor itself does not require dynamically allocated storage, since the maximum number tasks and devices is known and rather small, which means the necessary data structures can be assigned statically. Likewise, the tasks also have no dynamic storage allocation requirements; each is given a uniformly sized region of main storage when it is created. These regions also can be statically assigned to the sixteen potential tasks.

Input/Output

The supervisor performs input/output operations for the operator's console and for the devices owned by the tasks. All input/output activity is started via the Start Input/Output (SIO) instruction of the System/360. The termination of an input/output operation is signalled via an input/output interrupt. Likewise, the occurrence of an asynchronous attention condition is signalled via an input/output interrupt. Since input/output operations are performed using the System/360 input/output system directly, the supervisor must be able to describe and manipulate data structures corresponding in format to the Channel Address Word (CAW), Channel Command Word (CCW), and Channel Status Word (CSW) as defined by the hardware.

CPU Scheduling

The supervisor uses the hardware wait state as a means of suspending the execution of instructions by the processor when there is no useful work to be done. If interrupts are left enabled, the supervisor will be given an interrupt by the hardware when some change of state (such as an input/output interrupt) occurs. If interrupts are disabled, the hardware will remain in wait state until the operator intervenes. This is normally used after a fatal condition such as a supervisor error or hardware malfunction to suspend all activity and alert the operator to the malfuction.

In a sense, the supervisor performs CPU scheduling when it chooses one of the tasks to be given control of the processor. These tasks can be thought of as co-routines. When the supervisor has finished its interrupt processing it chooses one of these co-routines from among the set of those which are ready to use the processor and "returns" to it.

COMPILE-TIME FACILITY REQUIREMENTS

Compile-Time Procedures (Macros)

The supervisor requires compile-time procedures of sufficient power to generate one or more statements. It must be possible to pass arguments to the procedure which will be substituted for the corresponding parameters in the procedure definition. It must also be possible to perform tests for the value of a parameter (including the existence of its corresponding argument) and to use the results of such tests to influence the flow of control through the procedure. Local variables and the ability to generate unique identifiers also must be available within the compile-time procedures.

Attributes of Data-Items

Some attributes of data-items must be accessible as compile-time values. For example, the lengths of data structures and character-strings are frequently required.

DATA STRUCTURE REQUIREMENTS

The supervisor requires structures of homogeneous elements (vectors and arrays) where the elements of the

structure can be either simple elements or structures. The elements of the structure are referenced by subscription. Example structures of homogeneous elements used in the supervisor are vectors of pointers, vectors of task control blocks (which themselves are non-homogeneous structures), and vectors of channel commands (which themselves are non-homogeneous structures).

Non-Homogeneous Structures

Structures of non-homogeneous elements where the elements of the structure can be either simple elements or structures are needed by the supervisor. The elements of a non-homogeneous structure are referenced by name. Example structures of non-homogeneous elements used in the supervisor are the task control block and the channel command.

Data-items and structure elements are defined to be the same structure so frequently that some provision should be made to allow a structure to be treated as an entity, much like a data-item.

Storage Allocation

It must be possible to cause data-items to be allocated either in main storage or in hardware registers. In either case, it must be possible to either specify the specific allocation or allow the compiler to choose a free resource of the proper type. In the supervisor much has been gained by globally assigning certain critical values (such as pointers to the task control block and device control block currently being operated upon) into the general registers. The compiler either should do this type of optimization routinely or allow the programmer to specify that he desires a global assignment of a value to a register.

Storage Class

The supervisor requires data-items having static, based, and parameter storage class. Based data-items are used for task control blocks and device control blocks. Parameters are used for the interrupt code at the entry-points which are called by hardware interrupts. Static allocation is used for all other variables.

Scope and Ownership

All data-items referenced by the supervisor are of internal scope and thus are owned by the supervisor.

Length, Alignment, and Representation

The length, in bits, alignment, in the offset from some modulus, and representation of data-items must be specified precisely in the supervisor. The representation is usually chosen from one of the representations accepted by the hardware, but this is not always the case.

Pointers

The supervisor requires data-items which are pointers; addresses of some other data-items. The supervisor has pointers which are 24 and 32 bits in length, with alignments of 8 modulo 32 and 0 modulo 32, respectively. All pointer values are represented as unsigned binary magnitudes, where the value is interpreted as the main storage address of the item which the pointer addresses (in bytes).

Bit-Strings

The supervisor requires data-items which are bit-strings. The supervisor uses bit-strings which are 1, 4, and 8 bits in length with alignments of 0 modulo 1 and 0 modulo 8. All supervisor bit-strings are fixed in length.

Integers

The supervisor uses data-items which are integers having lengths of 4, 8, 16, 32, 40, and 64 bits with several different alignments. The representations which are used for integers are unsigned binary magnitude, two's complement, and packed decimal.

Reals

The supervisor references data-items which are real numbers in assignments although it actually does no computation with such items. The lengths of these items are 64 bits with an alignment of 0 modulo 64. These numbers are represented in floating-point as defined by the System/360 hardware.

Character-Strings

The supervisor requires character-string data-items. In all cases the character-strings are a multiple of eight bits in length aligned on a 0 modulo 8 boundary. Each eight bit group (called a byte) represents one character using the EBCDIC representation. All character-strings used in the supervisor are less than 256 bytes in length. All have a

fixed-length, with the exception of the input message area for operator commands, which can be considered to have a variable length with a maximum length less than 256 bytes.

Initialization of Data-Items

The supervisor must be able to initialize all the data-items which it owns. A variety of external representations should be allowed as initialization values. Numeric representations should be allowed in the bases 2, 10, and 16. Compile-time expressions must be accepted both as initialization values and as multiplicity factors. Pointers must be able to be initialized to the address of any data-items known to the program. Structures must be able to be initialized on an element-by-element basis.

COMPUTATIONAL REQUIREMENTS

Assignment

The supervisor requires assignment of values to data-items of the same mode as the value. Specifically, the supervisor requires assignment of integers to integers, bit-strings to bit-strings, character-strings to character-strings, pointers to pointers, and reals to reals.

The supervisor also requires the assignment of structures to identical structures and the assignment of the "null" structure (all zero bits) to a structure. This is used to clear many structures during initialization.

The supervisor requires the assignment of character-strings to integers and integers to character-strings, with the implied conversions between internal and external representations taking place. Both decimal and hexadecimal external representations are required. The replacement of non-significant leading zeroes by blanks and the specification of decimal-points as literal characters in the conversion to external form are required.

Arithmetic Operations

The supervisor requires addition, subtraction, multiplication, division, shifting, and modulo with integer operands. In some cases arithmetic with operands whose length is greater than 32 bits is required.

Relational Operations

The supervisor requires the six relational operations "equal", "not equal", "less than", "greater than or equal", "greater than", and "less than or equal". Required mode combinations are integer to integer, character-string to character-string, and pointer to pointer.

Boolean Operations

The supervisor requires the Boolean operations "conjuction" and "disjunction". The Boolean operations must be defined for operands of bit-string mode. It is assumed that the result of a relational operation can be considered a bit-string.

String Operations

The supervisor requires that a sub-string of any character-string be allowed as an operand of any operation which accepts string operands, including as the left-hand side of an assignment. Operations which scan a string for the first character in a set (or the first character not in a set) and returns its position in the string are required.

Miscellaneous Operations

The supervisor requires an operation which has as its value the address of its operand.

PROGRAM STRUCTURE REQUIREMENTS

The supervisor is a program which has six entry-points, its initialization entry-point and five other entry-points, each corresponding to one of the five types of hardware interrupts. The initialization entry-point never returns; the supervisor simply enters the hardware wait state at the end of the initialization. The five interrupt entry-points are called asynchronously. They can return, either to the task which was in control of the processor at the time of the interrupt (if any) or one of the other tasks which is ready to use the processor. The calling sequence and return code of these interrupt entry-points must save and restore all of the processor status, which requires the use of the PSA and the LPSW instruction, among others.

CONTROL REQUIREMENTS

Unconditional Transfer

The supervisor requires unconditional transfer statements extensively. The destination of the transfer is always a labelled statement within the supervisor. At times, a transfer is made to one of a set of labels, where the proper label is indicated by a computed value, such as an index value.

Conditional Statement

The supervisor requires a conditional statement similar to the IF statement of PL/I or the Logical IF statement of FORTRAN.

Iteration Statement

The supervisor contains several loops which could be effectively described with appropriate iteration statements. All loops consist of the type which are adequately expressed in PL/I ("DO I=1 TO N") or FORTRAN ("DO I=1,N").

Linked-List Maintenance

The supervisor maintains several one-way and circular linked-lists. Items must be added and deleted from these linked-lists. Either these operations should be provided as primitives, or the necessary operations on pointers should be available so that these list operations can be coded in terms of those operations.

Pause

The supervisor enters the hardware wait state with interrupts enabled when there is no useful work to be done by the processor.

Stop

The supervisor enters the hardware wait state with interrupts disabled when a fatal error in the supervisor or a hardware malfunction is discovered. This causes all processing to be discontinued until the operator intervenes to reload the system.

EFFICIENCY CONSIDERATIONS

Since there are no deadlines imposed by the hardware or tasks of the Computer and Communication Sciences 573 Supervisor and there is an ample amount of main storage, there are no concrete upper bounds on the efficiency required of the supervisor. However, the performance of the supervisor has a most significant effect on the performance of the total system.

Time Considerations

The time required by the supervisor to process an interrupt has a direct effect on the efficiency of the total system. For instance, if the mean number of interrupts required to service a task is 100 per second of processor time required by the task and the time required by the supervisor to process an interrupt is .5 milliseconds, then the supervisor overhead is 50 milliseconds per second of task processor time, or 5% of the total processor time spent by tasks. Under a fully loaded situation, the supervisor would require 4.76% of the total processor time, with the tasks using the remaining 95.24% of the processor time.

Code is currently being added to the simulator for the machine on which the supervisor runs to gather statistics on the performance of the supervisor. This code has not been completed in time for this report.

Storage Considerations

The supervisor has available 16384 bytes of main storage for the program and its static storage. The current supervisor requires 7504 bytes of main storage, about 45% of that available to it. Thus, any reformulation of the supervisor in a higher-level language can require no more than twice the amount of storage used by the current supervisor, which is written in assembler language.

Current Supervisor as a Standard

As is the case with the Paging Drum Processor, the current supervisor was written in assembler language by a competent system programmer. Any compiler-produced code can be expected to perform no better than the current supervisor. Thus, the efficiency of the code produced by a higher-level language compiler can be measured by comparing the performance of the code produced for the supervisor as expressed in that language with that of the current supervisor.

2.1.3 Conclusions

The two programs which we have analyzed in detail are probably the most difficult programs in a system such as MTS to express in higher-level languages. In the next several paragraphs we will summarize the requirements of these programs and indicate the degree to which these requirements are satisfied by contemporary higher-level languages.

ENVIRONMENTAL CONSIDERATIONS

Both the programs which we have analyzed operate in unusual environments. The Paging Drum Proccessor runs in the absolute task environment as defined by UMMPS and the Computer and Communication Sciences 573 Supervisor runs in the supervisor state environment of a simulated machine which is very similar to a System/360. These environments have a great impact upon the calling sequence conventions used by these two programs and upon the manner in which they request dynamic storage allocation, input/output operations, and affect the processor scheduling policies applied to them. Furthermore, even within the same environment there may be alternate ways to accomplish the same end. For example, the Paging Drum Processor uses two distinct services for performing input/output operations and three distinct types of dynamic storage.

These environmental considerations require that a higher level language designed for system programming must allow the programmer to specify the general environment in which the program will run plus more detailed environmental information when requesting services such as input/output requests or storage allocation.

Contemporary higher level languages are particularly weak in this area of environmental considerations. In fact, the situation is the reverse of the one we have outlined above; the higher-level language processor assumes a very specific and specialized environment suitable for the needs of its generated code. The system is augmented through a large set of library subroutines to provide precisely this environment. For example, higher-level languages generally generate calling sequences which conform to a particular set of conventions appropriate to their individual needs. Thus, the System/360 FORTRAN IV compilers always generate a specific type of calling sequence suitable for their needs and the System/360 PL/I compilers always generate another incompatible type of calling sequence suitable for theirs. For this reason, among others, it is not possible to call any program from a FORTRAN program which does not conform to the FORTRAN calling sequences, including programs written in PL/I. The code generated by the compiler and the library

subroutines would have to be extensively modified to allow programs written in a specific higher-level language to be used in an environment other than its normal one.

This rigidity in contemporary higher level languages and their processors with regard to their demands upon the environment has prevented the wide use of higher-level languages for system programming. In the few exceptions, such as MULTICS, where the bulk of the system programming has been done in a higher-level language, the system has been designed to meet the requirements of the paticular higher-level language used to write the system. For example, in MULTICS, which is written in PL/I, the environment in which almost all system programs operate is that expected by PL/I. All subroutines use PL/I calling conventions, even when more specialized, faster conventions would be adequate. A stack is always available to a program for use in subroutine calls. Hardware interrupts, which can be thouhgt of as asynchronous subroutine calls using highly stylized calling conventions, are quickly modified through interface routines into PL/I compatible calling sequences. Other services which are not generally available in PL/I, such as the issuing of privileged operations or allocating different types of dynamic storage, are performed through interface subroutines written in assembler language.

Although no contemporary programming language allows the programmer to specify environmental information to the compiler, there is no technical reason why this could not be done. For example, an option on the ALLOCATE statement of PL/I (which allocates dynamic storage) could specify the type of dynamic storage to be allocated. The code generated for this type of statement would vary depending upon the setting of this environment option.

COMPILE-TIME FACILITY REQUIREMENTS

The compile-time facility requirements which we have encountered in our analysis of two system programs are largely met by the compile-time facility of PL/I. The PL/I compile-time facility is weak in the area of availability of data-item attributes (such as the length of a data-structure or string) during compile-time. Again, this extension is easily made.

DATA STRUCTURE REQUIREMENTS

The requirements for homogeneous and non-homogeneous structures are satisfied by higher-level languages such as

PL/I and JOVIAL. The need for alternatives is not recognized in many higher-level languages as this ability to treat the same data-item in more than one way in considered undesirable by many language designers as it allows implementation dependent programs to be written and can complicate optimization techniques.

No higher-level language today gives the programmer the control over storage allocation which our two system programs require. Many languages such as Bliss and PL360 allow the user to allocate variables to machine registers. No language that we are aware of gives the user the ability to specify the allocation of variables to specific main storage addresses. Again, there is no technical difficulty associated with the implementation of this feature in a higher-level language.

The requirements for the specification of storage class, scope, and ownership of data-items also are largely met by PL/I and MAD/I.

One area in which contemporary higher-level languages are lacking is in the specification of the length, alignment, and representation of data-items. For example, PL/I on the System/360 always implements integer quantities as halfword, fullword, or packed decimal quantities. It is not possible to specify that a particular integer is to be represented in two's complement with a length of 7 bits, or in one's complement, or as an usigned binary magnitude, and so forth. Again, there is no reason why the programmer cannot be allowed to specify this information from a rather large set of acceptable alternatives. In fact, JOVIAL does allow a small amount of this information to be specified. Of course, those representations which are not natural for the particular computer upon which the program will be run will not generate as efficient code as more natural representations.

The primitive modes (pointers, bit-strings, integers, character-strings, and reals) are generally available in higher-level languages such as PL/I. We require the ability to specify the representation to be used for particular data-items separate from its mode, as discussed in the previous paragraph.

All higher level languages allow data-items to be preset. The programs which we have analyzed require only modest extensions to these facilities, such as the ability to specify repitition factors and preset quantities as compile-time expressions.

COMPUTATIONAL REQUIREMENTS

The computational requirements of these programs are well within the capabilities of higher-level languages such as PL/I. It is interesting to note that a large variety of the operations which are available in PL/I, but not older languages such as FORTRAN are heavily used, especially in the area of bit-string operations.

PROGRAM STRUCTURE REQUIREMENTS

The program structure capabilities of contemporary higher-level languages are not sufficient for the programs we have analyzed. It is necessary that these capabilities be extended to take into account environmental considerations. That is, the programmer must be able to specify the calling sequence conventions to be used for each entry-point and return in his program, and for each subroutine which his program calls. As stated earlier under environmental considerations, there are no technical reasons why this cannot be done.

CONTROL REQUIREMENTS

Again, the control capabilities of contemporary higher-level languages are in general adequate for the writing of system programs. The pause and stop facilities, which exist in most higher-level languages, must be extended to use environmental information. Both of the programs we have analyzed have made some use of linked list structures. It would be profitable, although not essential, to add facilities to a higher-level language for the maintenance of linked lists.

SUMMARY

From our analysis of two representative system programs, it is evident that these programs could be written in an appropriate higher-level language. Most of the data processing requirements of these programs are provided for in higher level languages such as PL/I. The major program requirement which is not adequately expressed in current higher-level languages is the description of the environment in which the program operates. However, the problems in these areas do not seem to be insurmountable.

Thus, our tentative conclusion is that although no higher-level language currently satisfies all the

requirements of system programs it certainly would be feasible to design a higher-level language which would satisfy these requirements.

3.0 GENERAL APPROACHES TO CPU SCHEDULING IN RTOS

A small real-time system, such as might be found in an industrial process control application, can often be designed as a single entity, with any internal subdivisions being made largely for the programmer's convenience. But the design problem for large-scale systems, such as BMD systems, seems to require that a multilevel scheduling and control hierarchy be established early in the design process. The nature of this hierarchical structure will affect the entire design and performance of the system. This discussion will present several general approaches to the design of two levels of this hierarchy; namely, (1) the subdivision of the system into basic functions, and (2) the subdivision of basic functions into tasks.

3.1 THE DIVISION OF SYSTEMS INTO BASIC FUNCTIONS

We assume, as a first principle, that some sort of functional modularity is essential, in order to approach the design process at all. The BMD problem is typically divided into basic functions, such as search, verification, discrimination tracking, interceptor allocation, interceptor guidance, and intercept assessment. Each of these functions may have several inputs to process simultaneously. The tracking function, for example, may have to track several objects at the same time. Thus, in an effort to visualize the several possible approaches to this level of hierarchy, we may consider the two-dimensional space of functions and inputs illustrated in Figure 3.1.

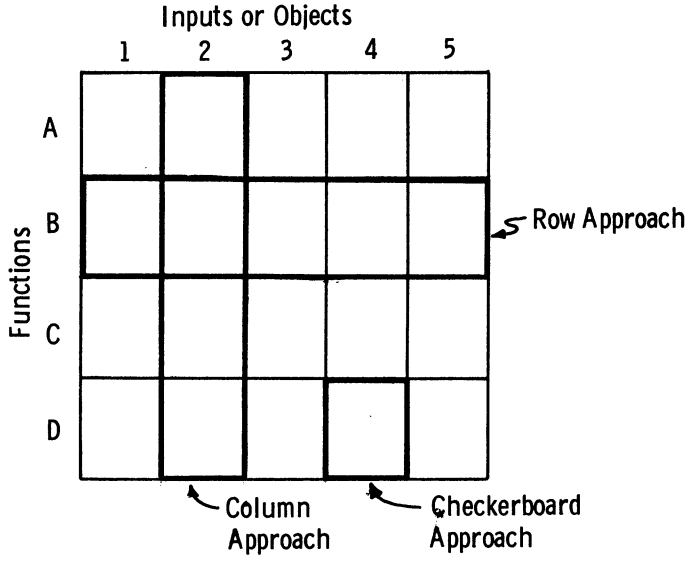


Figure 3.1

The three approaches suggested by this picture might be called:

(1) The row approach, in which the basic entity is a function, which processes as many of its inputs as possible, each time it is invoked.

(2) The column approach, in which the basic entity is a task, responsible for processing a single input, or object, through all of the basic functions.

(3) The checkerboard approach, in which the basic entity is the invocation of a single function to process a single input set.

We will consider the various motivations for each of these approaches, in order.

3.1.1 The Row Approach

This approach, which is essentially that used in AN/FPS-85, to be described, has the advantage that those portions of the function which are not input-specific are performed only once per invocation of the function. In addition, some advanced hardware designs allow vector and array processing, which could be used advantageously by this approach. There are two potential disadvantages to this approach. The first of these concerns overload situations. It will be necessary for each function to evaluate its performance, and relate it to the overall system load, in order to make decisions as to degrading its performance in one of two ways: (1) lower its accuracy in processing each input, or (2) choose to ignore certain inputs. If these decisions are not made correctly, it is possible that some functions may use so much time that others cannot be completed on time. Certainly these decisions must be made somewhere in the system; it is a question of where. More will be said on this matter later. The other potential disadvantage is that it may not utilize multiprocessor hardware configurations very well, since it may be advantageous to use several processors to accomplish the same function simultaneously. But as we will see, there is another level of hierarchy which deals with this problem more directly.

3.1.2 The Column Approach

This is the approach used typically by computer utilities, where the "objects," or "inputs" processed are the users of the utility. There are no instances of this approach in BMD or related areas, and it is rather confusing to try to apply it there, since the inputs processed by the various functions are not the same. Tracking, for example, deals with

reentry vehicles, but interceptor guidance deals with interceptors. One might envision grouping functions which do process the same inputs, under this approach. The only advantage which this might claim, however, is some degree of logical simplification, and even this is unclear.

3.1.3 The Checkerboard Approach

This approach is the one taken by NEXOS and SENTOS. It would appear that this organization will allow the greatest flexibility, in that the system can allocate its resources to smaller units, thus allowing both more efficient usage of resources and simpler decisions for handling overload situations. But this comes at the cost of additional overhead, and in any case it does not appear that NEXOS or SENTOS really make much use of this flexibility, in view of the rather nondynamic nature of their scheduling algorithm. Also, there is the additional necessity for several entities, called basic function controllers, which oversee the operation of the collection of functions which each operate on single inputs. This may not be a disadvantage however, since we now have a natural place to make degradation decisions, based on system load, etc. Unfortunately, this organization may not allow much flexibility in these decisions, because of additional constraints such as timing, and data set locking, which will be discussed later.

3.2 THE SUBDIVISION OF FUNCTIONS INTO TASKS

The second level of hierarchy is the subdivision of the basic entities described above into tasks. There are several motivations for making this subdivision. It is perhaps worth noting, first, that functional modularity is not one of them, although functional considerations will affect the design of the specific subdivision. What we are discussing here is the creation of even smaller schedulable entities than those discussed above.

The most important motivation for this subdivision is the efficient utilization of multiprocessor hardware. It is argued that the greater the "granularity" of asynchronous parallel processes, the greater will be the utilization of the processors. Thus it would appear that granularity and multiprocessor utilization are more properly included as motivations for the second level of hierarchy rather than, as it is usually purported, for the first level (Section 3.1).

A second motivation deals with handling interrupts and priorities. There are two possible approaches here; one approach allows tasks to be interrupted for the execution of higher priority tasks, and the other does not. The first of these requires additional overhead, to save task

state, etc. The second requires some method for returning to the scheduler fairly often, to check for higher priority tasks. The usual method for accomplishing this is to restrict the execution time of all tasks to some small interval (~1 msec in NEXOS). This necessitates the subdivision under discussion. Note that both of these methods require some overhead, with the "polling" or short-task method likely to require more.

A third motivation for the subdivision of functions into tasks is simply that some functions may have inessential, low priority, portions, such as trace recording, etc., which can be separated and automatically eliminated in system overload situations.

This subdivision usually takes the form of a task set, organized into a network, with precedence and enablement conditions, for synchronized execution of the various parallel tasks. An example is shown in Figure 3.2. Here, tasks 2, 3, and 4 must wait for the completion of task 1, task 5 waits for 2 and 3, etc.

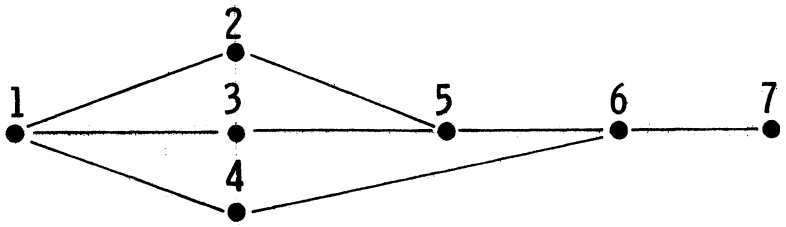


Figure 3.2

3.3 OTHER PROBLEMS

Finally, we consider several approaches to timing constraints and data set locking problems. The BMD and related systems designed to date have all based their operation on some fixed timing cycle, on the order of 50 msec, and the NEXOS and SENTOS systems have subdivided this major cycle into eight minor cycles. The motivation for this design decision is not clear, since none of the peripherals (phased-array radar, interceptor guidance) physically require these cyclic timing constraints. One possible motivation might simply be tradition; that is, radars have been cyclic in the past, and it may have become customary, for this or other reasons, to design real-time systems around such a basic cycle.

There are, however, two more compelling possibilities. The first is the problem of nonreal-time resource allocation. If all tasks are required to operate cyclically, then resources may be allocated to them in fixed time slots which are part of a basic cycle, and this may be done as part of the system design, rather than during execution. It is not clear how much efficiency is gained by this prescheduling; certainly computer

utilities, such as MTS, do not utilize such timing cycles, and successfully allocate resources on a completely dynamic basis.

A second motivation for use of this timing cycle might be that it eliminates the need for locking of shared data sets, simply by assuring that different tasks reference shared data sets during different time slots in the cycle. Data set locking will influence CPU scheduling in any operating system, but there are many approaches in addition to the above, which appears to be that used in NEXOS, and to some extent in SENTOS. Other methods require some sort of dynamic "blocking" of tasks trying to simultaneously reference shared data sets. In AN/FPS-85, this blocking is done by controlling entry to so-called shared processors, while in many nonreal-time systems this is accomplished by certain "event control" primitives which allow tasks to wait for arbitrary events, such as a task finishing its use of a shared data set.

Finally a note about deadlines. For various reasons real-time tasks may have deadline times, before which their execution must be completed. None of the systems described here have any method of dynamically assuring the meeting of deadlines, although they do have methods for checking deadlines and taking some corrective action if necessary, and the AN/FPS-85 system has methods for changing priorities in an effort to meet deadlines if possible. For the most part, however, these timing requirements are met by some undescribed nonreal-time scheduling, and by a lot of hope.

4.0 A DESCRIPTION OF CPU SCHEDULING IN THE AN/FPS-85 MONITOR

The approach taken here will be to describe first the basic schedulable entity, followed by a presentation of the properties of the schedulable entity which affects the scheduling algorithm, and finally, to describe the algorithm itself. The primary reference for this description is the AN/FPS-85 specifications [Ref. 3].

4.1 SCHEDULABLE ENTITY

The schedulable entity is called a sequencer. The system supports on the order of 70 sequencers. There is one entry for each sequencer in the monitor's "DO-list," which is the primary table used by the scheduling algorithm. A sequencer is, logically, a sequence of instructions to be executed synchronously. It is organized into a hierarchy of program modules, which we describe here because it can affect scheduling decisions.

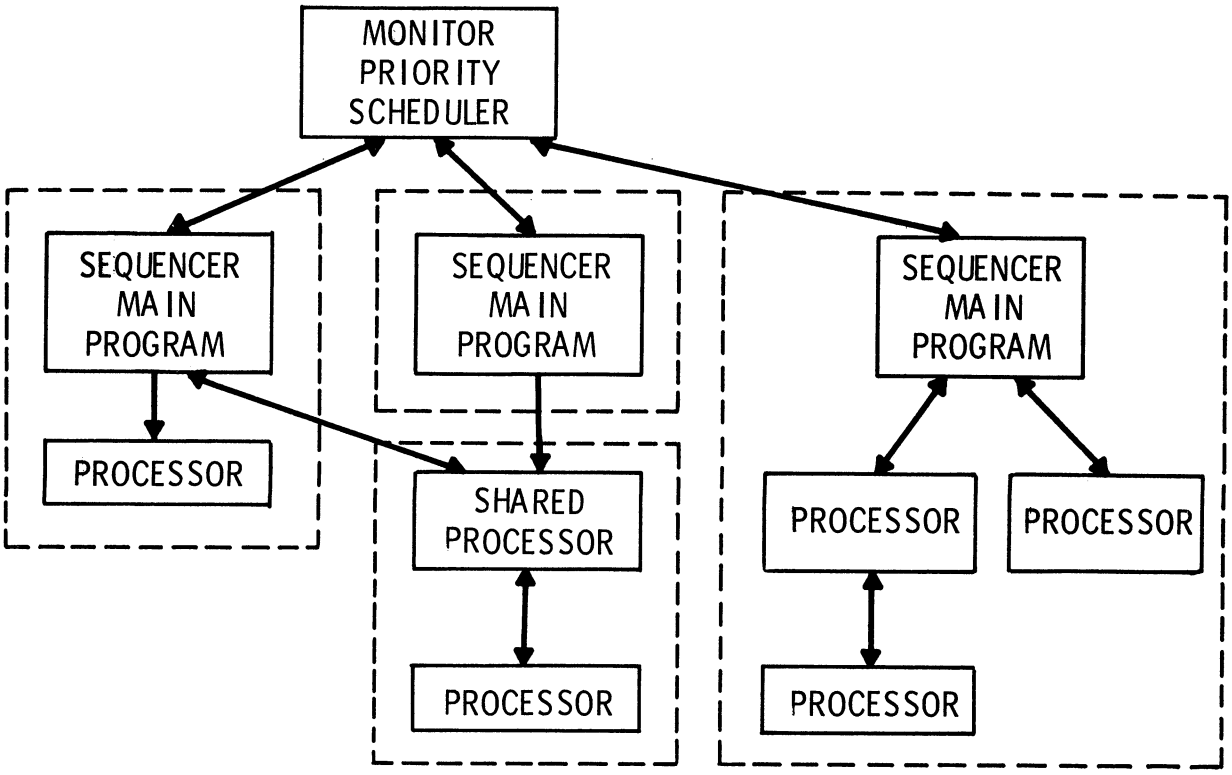
The highest level program module in a sequencer is called the sequencer main program (in the AN/FPS-85 literature, this is also called a sequencer but we modify it here to distinguish the two concepts).

The sequencer main program may call several submodules, called processors, and processors may in turn call other processors. A processor may be reentrant, in which case it can be shared among several sequencers. In addition, certain nonreentrant processors may be shared, and these are called shared processors. The monitor must intercept calls on shared processors and assure that they are executed by only one sequencer at a time. This fact can affect scheduling decision. One might ask why these processors cannot be made reentrant; the reason appears to be that separate uses of shared processors update a common data base, and the system does not provide a locking mechanism, or rather, no means is provided for waiting on any events except I/O completions.

An example of this hierarchy is shown in Figure 4.1. The program modules are also grouped, as shown by the dotted lines, into association groups, but it is not clear that this is any more than a descriptive device.

4.2 SEQUENCER TIMING PROPERTIES

We next consider the timing requirements of sequencers. The specific properties of interest here are (1) triggering time, t_T; (2) dispatch times, t_{D_1}, t_{D_2},...; (3) completion time, t_C; (4) deadline, t_E; (5) time checking interval, t_I; and (6) priority. These times are illustrated in Figure 4.2.



– – – – Indicate ASSOCIATION groups.
Each association group except the shared processor is a sequencer.

Figure 4.1

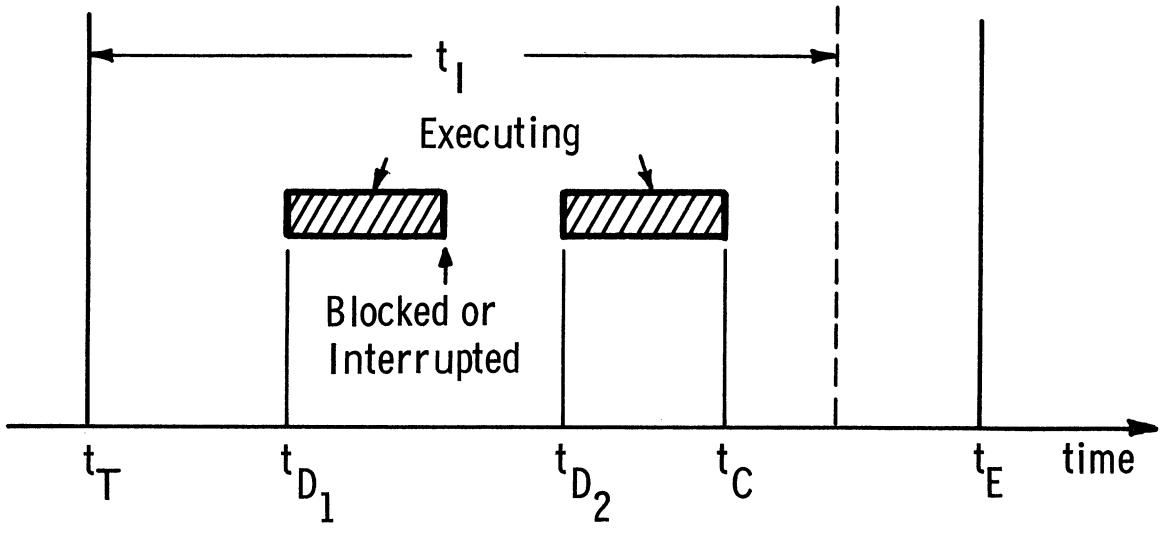


Figure 4.2

The first of these items to be described here is triggering time. There are three basic methods for triggering sequencers; these are: (1) periodic; (2) interrupts; and (3) direct triggering by the sequencers themselves.

4.2.1 Periodic Triggering

The basic timing period in the system is called a pulse burst period (PBP), and is 50 msec in length. Most of the sequencers responsible for actually controlling the radar run once in every PBP. These sequencers are actually triggered by the external interrupt supplied by the radar interface and control equipment (RICE) every 50 msec. Hence, although they are periodic, the triggering mechanism is an interrupt. One of these sequencers (MRTIME) is responsible in part for triggering the other periodic tasks, which have periods ranging from 1/2 sec to 24 hr. For this purpose, an ordered table of timing events is maintained, and at the beginning of every PBP the first entry is compared to the current time. Thus, all timing events are accurate to only 50 msec.

4.2.2 Interrupt Triggering

Aside from the PBP interrupt, the only other interrupts relevant here are those related to communications, i.e., requests to display information, maintainance of the communications link to the Space Defense Center (SDC), etc. That is, sequencers responsible for man-machine interaction are triggered by these interrupts. I/O interrupts and SVC's occur also of course, but these do not trigger sequencers, although they may result in dispatching of different sequencers (namely those which were in I/O wait, or those which were executing in a shared processor).

4.2.3 Program Controlled Triggering

This function is normally called enablement. It is the mechanism whereby sequence dependencies are handled. Any sequencer may trigger itself, or any other, in one of three ways: (1) a call to monitor entry TRIGR; (2) a call to monitor entry DTRIGR (delayed trigger) in which one specifies a time delay until the actual triggering (actually, the triggering of periodic tasks is also initiated via this monitor entry); and (3) by enqueueing a queue block on the standard input queue belonging to the sequencer.

Figure 4.3 lists some of the functions which fall into categories 4.2.1 and 4.2.2. Each of these functions may be performed by several sequencers, the first of which would directly trigger others, and so forth. This adds an additional level of program hierarchy above the sequencer. This level corresponds to the Operational Program Configuration (OPC) in NEXOS. This level

Periodic				Interrupt	
24 hr	30 min	5 min	1/2 sec	PBP	Anytime
·Compute Entry/Exit times for next 24 hr	·Organize sats. into 1/2-hr radar cov. intervals for next 90 min ·Calculate Ephemerides for sats. in coverage next 1/2 hr ·Generate C.C. for next 1/2 hr, up to 90 min ahead ·Organize C.C. into 1-min intervals ·Organize tasking for next 1/2 hr into 5-min intervals	·Calculate tasking parameters for next 5 min	·Hold tasks until 1/2 sec before first transmission	·Schedule transmissions ·Process returns ·Search ·Confirm ·Correlate ·Unknown track ·Radar control ·Radar reconfiguration ·Real time maintenance	·Operator inputs and outputs ·OE update ·Display information ·Smooth rets. into OB's ·Send OB's to SDC

Figure 4.3. Functional timing.

also corresponds to the top level of hierarchy described in general approaches, and the sequencers themselves correspond to the tasks into which the basic functions are divided. The subdivision into processors is yet a third level of hierarchy.

The actual triggering is accomplished by setting a bit in the monitor's DO-list entry for the triggered sequencer. These DO-list entries also serve another purpose; they are the queue directory blocks for the sequencers' input queues. This explains the triggering mechanism in (3) above. This is apparently considered the standard triggering mechanism, hence the name standard input queue. The DO-list entries for all other triggering methods, in which no input data is passed, are called pseudo input queues. Note that the use of input queues implies the use of the "row approach" described in the section on general approaches. That is, each sequencer processes as many of its inputs as possible, each time it is triggered. A sequencer with a standard input queue is considered to be in the triggered state as long as there are any queue blocks in its input queue. The removal of blocks from any queue is completely under control of the sequencer itself, via the various queue manipulation primitives supplied by the monitor.

4.3 PRIORITIES AND DEADLINES

Consider next the question of priority determination. The DO-list entries are ordered by priority, with the highest priority sequencers first. Since the dispatching algorithm scans the DO-list in order, the highest priority sequencers are usually executed first.

The DO-list is prepared in nonreal-time, apparently by ad hoc, trial and error methods. There is provision for a preplanned reordering of priorities, however. This is accomplished by inserting multiple DO-list entries for some sequencers, with the higher priority entries called alternate pseudo input queues. Every sequencer which has an alternate pseudo input queue has a time checking interval associated with it (see Figure 4.2). Upon expiration of this time checking interval, starting when the standard or pseudo input queue is triggered, the alternate pseudo input queue is triggered.

It is this mechanism which attempts to assure that sequencers meet their deadlines. Deadline times are not recorded explicitly within the system, and apparently it is left to individual sequencers to do their own time checking if necessary. The time checking described above is handled by the same mechanism as the triggering of periodic tasks. Hence it is accurate only to 50 msec, and cannot be applied to those sequencers which run in every PBP. One must therefore suppose that all of these sequencers have highest priority, and hope that they all get done on time. Fortunately there is no great loss if they do not; the system can skip a PBP if necessary.

4.4 STORAGE ALLOCATION

Before describing the dispatching algorithm, it is necessary to briefly describe storage allocation, since this is one of the factors in that algorithm. Main storage is divided into six regions, as shown in Figure 4.4. Every sequencer is either resident in the root segment, or is assigned to a specific one of the four shared areas. This assignment, as well as all required relocation and linking, are done in nonreal-time, and a core image is written into a disk file, one of which exists for each shared area. The problem, of course, is that the triggered sequencer of highest priority may not be in core, and, what is worse, its shared area may be occupied by a triggered or executing sequencer. In addition, it is sometimes necessary to expand queue storage into shared area 4, and this area may be occupied by a triggered or executing sequencer.

4.5 THE DISPATCHING ALGORITHM

Now we can describe the actual scheduling algorithm, or, more exactly, the dispatching algorithm. What we are describing here is the method by which the dispatch times, t_{D1}, t_{D2},..., indicated in Figure 4.2 are determined. The reason for indicating multiple dispatch times is that the execution of a sequencer may be interrupted, for either of two reasons: (1) it may be blocked, waiting either for an I/O completion, or to use a shared processor; (2) it may be interrupted for the execution of a higher priority sequencer.

Basically, the procedure is to scan the DO-list for a triggered input queue, and give it control if possible; if not possible, various actions may be taken, most of which include a continued scan of the DO-list. The DO-list scan does not begin at the top, but usually at the highest priority triggered sequencer. Apparently, the triggering mechanisms maintain the scan pointer, for this purpose.

In more detail; having found a triggered sequencer, several conditions may exist:

A. The sequencer is in core and has not begun execution (is not interrupted). In this case, the sequencer is given control unless it is in shared area 4 and this area has been requested for expanded queue storage, in which case the DO-list scan continues. (Supposedly the area is then given to queue control and the sequencer marked as not in core, but this is not stated, and something seems a little strange here.)

B. The sequencer is in core and has been interrupted. In this case the sequencer is given control. Note that one sequencer

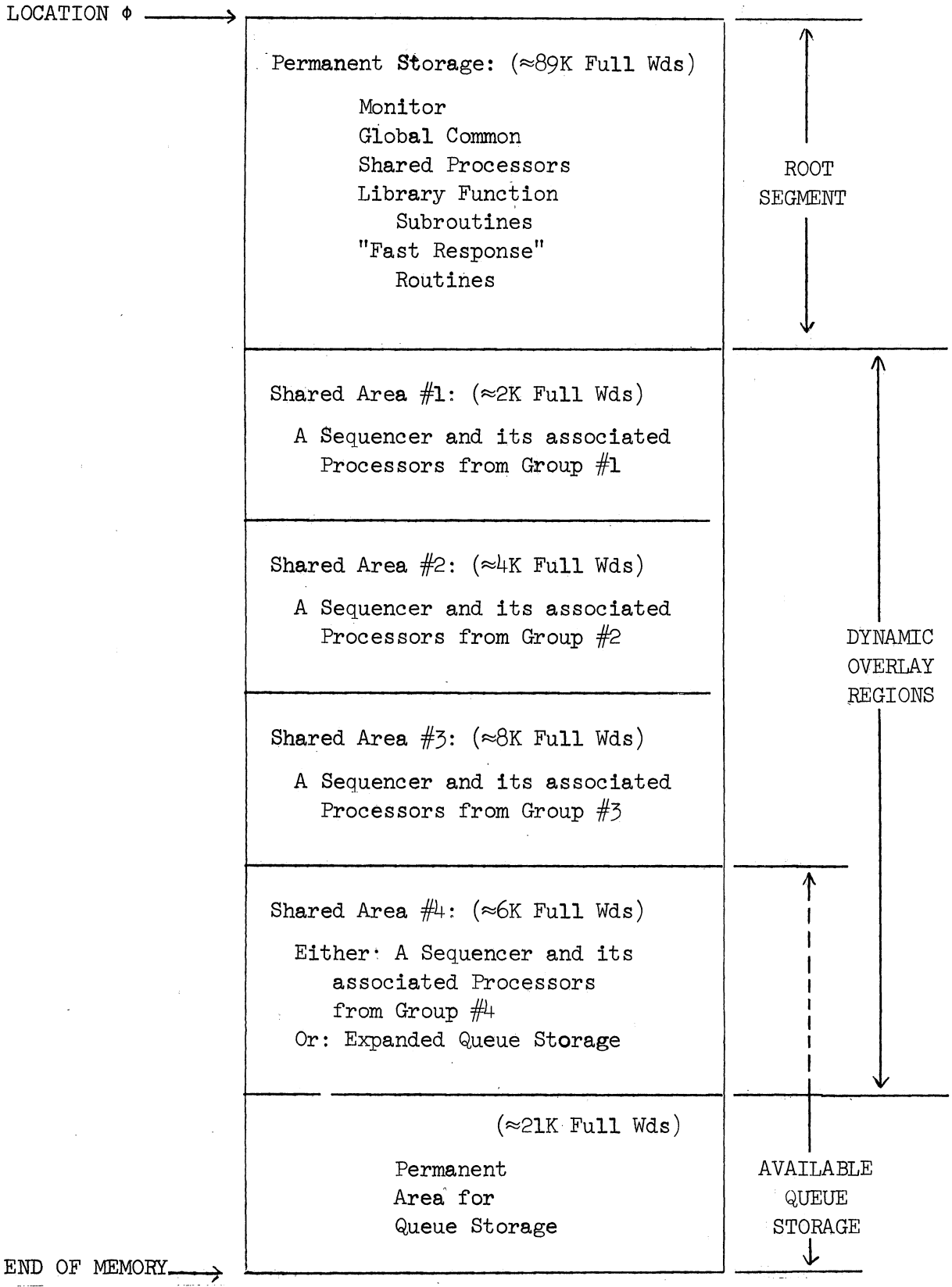

LOCATION ϕ
Permanent Storage: (≈89K Full Wds)
Monitor
Global Common
Shared Processors
Library Function
Subroutines
"Fast Response"
Routines
ROOT
SEGMENT
Shared Area #1: (≈2K Full Wds)
A Sequencer and its associated
Processors from Group #1
Shared Area #2: (≈4K Full Wds)
A Sequencer and its associated
Processors from Group #2
Shared Area #3: (≈8K Full Wds)
A Sequencer and its associated
Processors from Group #3
DYNAMIC
OVERLAY
REGIONS
Shared Area #4: (≈6K Full Wds)
Either: A Sequencer and its
associated Processors
from Group #4
Or: Expanded Queue Storage
(≈21K Full Wds)
Permanent
Area for
Queue Storage
AVAILABLE
QUEUE
STORAGE
END OF MEMORY

Figure 4.4

may be referenced by several standard or pseudo input queues. It is thus possible that the triggered queue just found is a different, higher priority queue than the one for which the sequencer was initiated. Thus its execution will automatically be resumed at the priority of the highest input queue requesting it.

C. The sequencer is in core, but is waiting, either for an I/O completion or for a shared processor. The DO-list scan continues. The scheduling of shared processors is handled by a separate algorithm, to be described later.

D. The sequencer is not in core. There are four subcases here, depending upon the status of the shared program region assigned to this sequencer:

(1) The region is free. A READF is initiated to load the sequencer, and the scan continues.

(2) The region is in use for, or has been requested for expanded queue storage. The scan continues.

(3) A READF is in progress for the region. If the I/O is complete, the sequencer read in is marked as in core. If this sequencer is the one currently under scan, and the area has not been requested for queue storage, it is dispatched. Otherwise the scan continues.

(4) The sequencer occupying the region is in use. If possible, the occupying sequencer is given control. If not, the scan continues.

This completes the description of the dispatching algorithm. Figure 4.5 summarizes the various conditions and actions.

A brief description of the shared processor scheduler is also relevant here. All calls on shared processors are intercepted by the monitor. If the desired processor is in use, the calling sequencer is marked as waiting (condition C above) and the using sequencer is given control, thus running at the (necessarily higher) priority of the calling sequencer. There are some unanswered questions here however. What happens if this lower priority sequencer is interrupted again, before concluding its use of the shared processor? In this case two possibilities may occur which do not seem to be accounted for.

(1) A sequencer of lower priority than the waiting sequencer may be given control, since condition C above states that the DO-list scan will skip the waiting sequencer; and

(2) if any sequencer, other than the using sequencer is dispatched, then the shared processor may be called a third time.

This concludes the description of CPU scheduling in AN/FPS-85. There are two areas in which research seems most relevant. These are both concerned with priority determination.

(1) The discovery of an efficient, effective procedure for producing a DO-list, together with the time checking intervals, etc., which is optimal or nearly optimal. This may require, or be aided by—

(2) a method for dynamically reordering priorities, based upon examination of system load factors, etc.

	Sequencer Conditions				Actions	
	In Core	Interrupted	Waiting	Q Storage Requested	Give Control	Continue Scan
A	√	---	---	---	√	
	√	---	---	√		√
B	√	√	---	√/-	√	
C	√	---	√	√/-		√
D	Region in Use		Fetch in Progress			
E		---	---	---	issue READ	√
F		---	---	√		√
G		---	√	---		√
H		---	√	√		√
J		√	---	---	Give control to occupying sequencer	
K		√	---	√		√

Figure 4.5. Monitor priority scheduler.

5.0 NEXOS-SENTOS OPERATING SYSTEMS

5.1 GENERAL PROGRAM ORIENTATION

NEXOS (Nike-X Operating System) and SENTOS (Sentinel Operating System) [Ref. 4] are operating systems designed to run special purpose military computers, phased array radars, and other special purpose peripherals in a real-time environment, namely, that of ballistic missile defense. The operating systems are designed to be executed on a large scale (large number of processors) multiprocessor system. The system of processors and peripherals is designed to carry out the detection, verification, tracking, and interception of hostile reentry vehicles (RV's). The underlying design philosophy in NEXOS-SENTOS is the "checkerboard" organization, outlined in Section 3.1.3. In particular, the checkerboard in Figure 5.1 may be used as a first approximation to program organization.

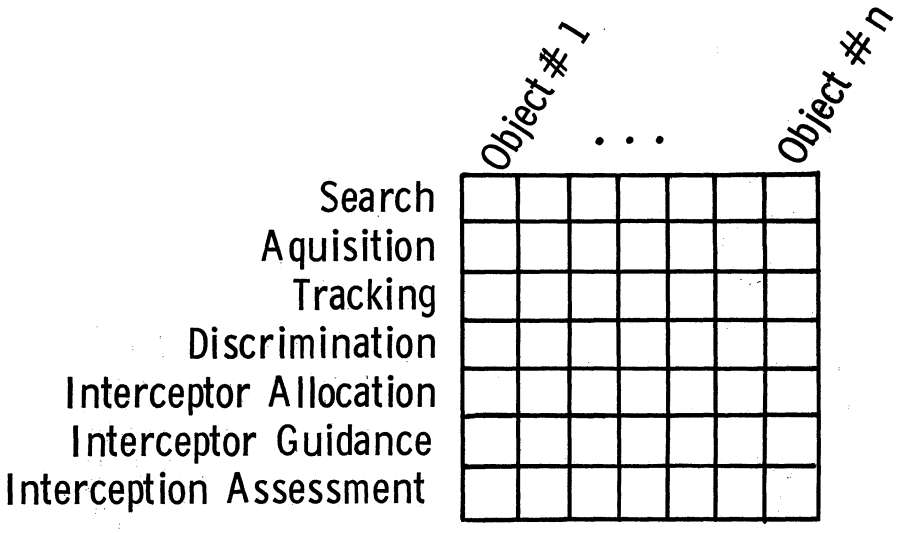


Figure 5.1. NEXOS-SENTOS checkerboard.

For each square in the checkerboard, an organizational element called a timed array is provided. A precise definition of a timed array will be given later; however, for the time being, it will be treated in terms of its behavior and logical structure rather than in terms of the particulars of implementation. At any point in time, a time array, TA_{ij}, which carries out the ith function on the jth RV, may be turned on or off. TA_{ij} is in turned-off status for one of the following reasons: (1) RV_j has not yet entered the system; (2) RV_j is in the system, but is not currently at the ith stage or processing; or (3) TA_{ij} has been turned off because of system overload conditions, although if resources were available, it could still be meaningful to

execute TA_{ij}. When TA_{ij} is in a turned-on status, one or more of the following TA properties affect its operation:

(1) A partitioning into units called tasks, each of which has a (static) priority and may require the completion of other tasks prior to its initiation (precedence conditions).

(2) Times relative to the beginning of a major cycle, a fixed cycle of operation, at which certain of its tasks are to be enabled by the operating system to execute.

(3) Major-cycle-relative times, deadlines, at which the operating system must check the completion of certain of its tasks.

(4) Data set locking conventions which administer the sharing of data by two or more tasks.

(5) Dependencies of task initiations on completion of I/O operations.

The goals of this organization are (1) to rigidly control the bounds within which the system operates, and (2) to partition the workload into many small pieces (tasks) which are capable of being executed in parallel on a large-scale multiprocessor system. The advantages and disadvantages of this approach are direct consequences of its design goals. Rigidity of task scheduling may introduce an inordinate number of implementation constraints which are not fundamentally a part of the actual problem. For example, the decision to operate (in the case of NEXOS and SENTOS) the system on a fixed length cycle implies that performence degradation must come in discrete "chunks" as the system is driven into overload. Thus a task either misses its deadline or makes it; and the consequence of either of these alternatives is to either run or not run other tasks. This discrete degradation is markedly different from the more nearly continuous degradation possible in a "row-oriented" approach, wherein basic functions can, over some reasonable continuum, reduce their execution frequencies. If one considers scheduling to be the determination of the "best" time to execute a given piece of code, and dispatching to be the operating system activitiy necessary to carry out what has been sceduled, then NEXOS-SENTOS has at run time a crude scheduler and a sophisticated dispatcher. By contrast, the row approach (Section 3.1.1) would need a sophisticated run time scheduler because of the variable execution frequencies, but would also utilize a much simpler dispatcher. It is not immediately apparent that one approach has a decided intrinsic advantage over the other. The partitioning of the workload into discrete tasks with fixed timing constraints also has the disadvantage of introducing occasional anomalous behavior, i.e., situations where lower priority tasks are run to completion while higher priority tasks miss their deadlines. While legislating task lengths to a minimum will reduce anomalous behavior as well as introduce more parallelism, it has

the disadvantage of raising relative overhead and perhaps forcing unnatural breaks in execution flow.

5.2 SPECIFIC PROGRAM ORGANIZATION

This section is intended to specify the implementation structure of NEXOS-SENTOS-like system. We begin with some difinitions of terms and then present paragraphs detailing for NEXOS the general concepts presented above.

5.2.1 Definition of Terms

DATA SET - a description of the structure of a collection of data-items operated upon by tactical programs (in SYSTEM/360 parlance, a DSECT).

DATA SET INSTANCE - one physical copy (record) of data items described by a data set.

ROUTINE - a segment of code that may be logically connected to one or more (fixed) instances of (fixed) data sets.

ROUTINE USE # - an identification of a particular routine and (implicitly) the data set instances associated with it. (Different use numbers may identify the same physical copy of the routine in which case it is usually reentrant.)

TASK - a collection of serially executed routine use numbers, which is the dispatchable unit in system scheduling. Once a processor has begun a task, it will attempt to run that task to completion, i.e., tasks are not multiprogrammed. A task has a priority, CEB string and AEB (defined below).

START TASK - a task which is enabled to run at a specified phase point by the operating system.

CEB STRING - conditional enablement bit string. For each event that must occur before a task can be run, there is a 1-bit in its CEB string. An event may be the completion of another task, the unlocking of a shared data set, or the completion of an I/O operation. As an event takes place, the 1-bits in various CEB-strings corresponding to that event are set to zero. When all the bits in a task's CEB-string are set to zero, its AEB is set to 1.

AEB - absolute enablement bit. The AEB signifies that a given task is ready to be run. When a processor becomes free it will be assigned to the highest priority task with its AEB = 1.

MAJOR CYCLE - the fixed length basic cycle of system operations.

PHASE INTERVALS - fixed length equal divisions of a major cycle.

PHASE POINT - the delimiting point between successive phase intervals.

TIMED ARRAY - a collection of one or more tasks which may have interacting precedence conditions and which usually includes a start task, and an end task, run at a fixed repetition rate, e.g., once per major cycle.

PROCESS - a collection of timed arrays, which in conjunction with the operating system carries out the complete tactical mission.

PROCESS CONSTRUCTION - assembly of symbolically coded descriptions in nonreal time, of all components of a process into tables used at execution time to control the process.

PROCESS DESIGN - the intelligent synthesis of process construction inputs, based on problem analysis and prior experience.

5.2.2 How the Structure is Utilized

Within a process, there exist a privileged class of TA's called basic function controllers, BFC's, each of which oversees the operation of TA's which carry out the basic function. The basic function controller, via operating system requests, turns on and off the TA's which it oversees in response to requests made by other BFC's and in response to system loading conditions. For example, assume the search BFC gets a positive reply. It can then pass position data on to the verification BFC, which will assign a TA to handle the alleged RV. If a positive verification is made, appropriate information can be passed on to the tracking BFC, etc. As the system is driven into overload, some of the deadline tasks will begin to miss their deadlines. When the frequency of missed deadlines exceeds some threshold value, an overload stage number is incremented. The basic function controllers have built-in rules (functions of overload stage number) for selective deletion or curtailment of the functions which they oversee. Thus the feedback loop, comprised by a BFC requesting TA turn-on and the ultimate incrementing of overload stage number which may in part be due to the request, is usually long and rather indirect.

5.2.3 Task Dispatching

Figure 5.2 displays the principle data sets employed in NEXOS task dispatching. Task dispatching is comprised of three parts: (1) finding the highest priority task for which all precedence conditions have been satisfied;

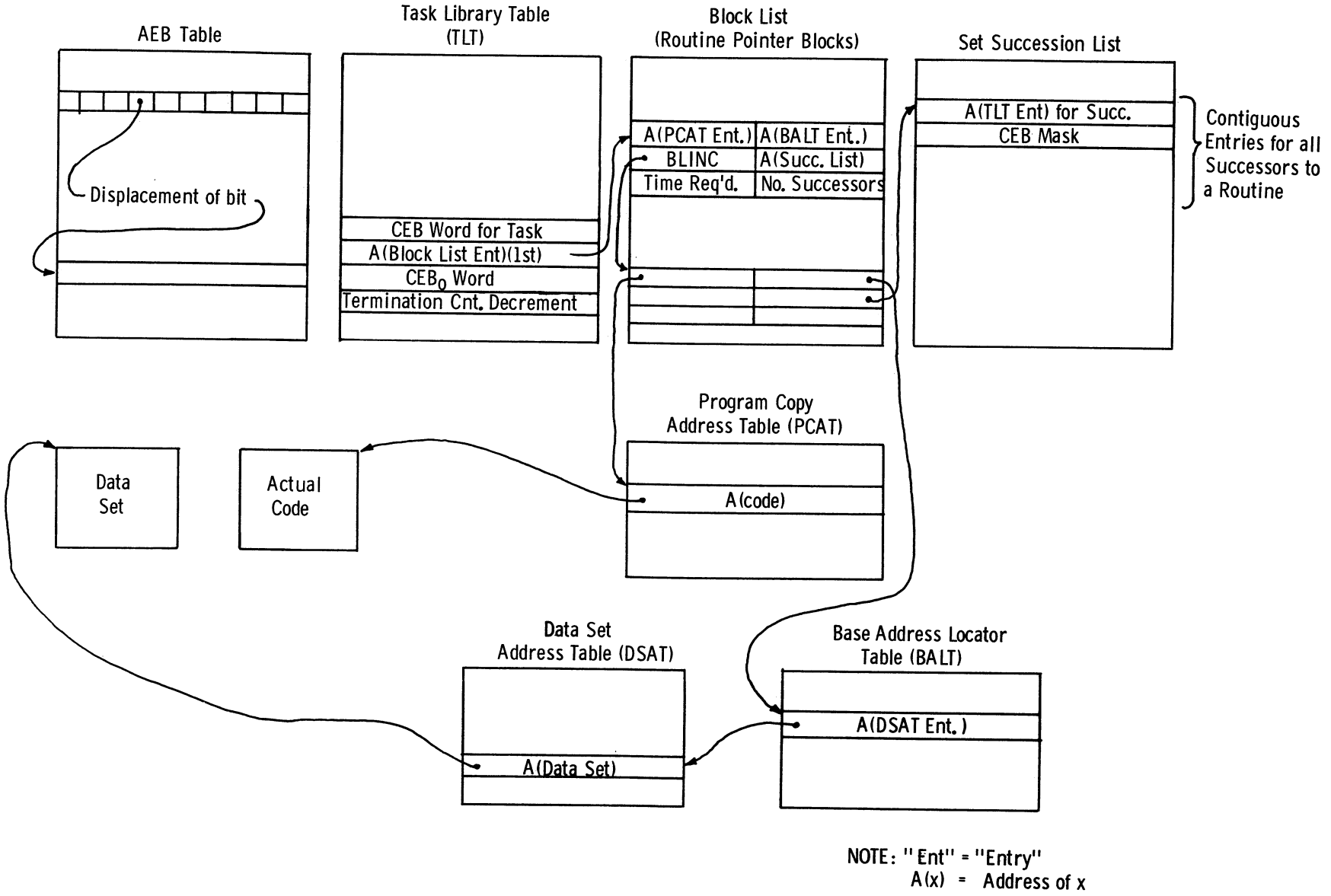

AEB Table
Displacement of bit
Task Library Table
(TLT)
CEB Word for Task
A(Block List Ent)(1st)
CEB0 Word
Termination Cnt. Decrement
Block List
(Routine Pointer Blocks)
A(PCAT Ent.)
A(BALT Ent.)
BLINC
A(Succ. List)
Time Req'd.
No. Successors
Set Succession List
A(TLT Ent) for Succ.
CEB Mask
Contiguous Entries for all Successors to a Routine
Program Copy
Address Table (PCAT)
A(code)
Data Set
Actual Code
Data Set
Address Table (DSAT)
A(Data Set)
Base Address Locator
Table (BALT)
A(DSAT Ent.)
NOTE: "Ent" = "Entry"
A(x) = Address of x

Figure 5.2

(2) carrying out the calling sequence necessary to invoke the task; and (3) serving succession, that is, altering the CEB words of other tasks (whose execution must be preceded by—hence the term precedence condition—the current task) to reflect the condition is now satisfied. The following description assumes that a free processor is starting at step (1). The processor scans the AEB string until it finds an absolutely enabled task (AEB = 1) and sets the AEB to zero. The displacement of the AEB in the AEB string is used to located the task library table (TLT) entry for the task. The TLT entry points to the block list, which is a linked list of blocks that define the various routines (tactical programs), data set linkages, succession to be served, and maximum run time, for each routine in a task.

Through several levels of indirection, the address of the code to be executed and its data set pointers are located; and control is passed to the routine. Within the routine all data (other than internal scratch variables, which are allocated from a pushdown stack "owned" by a processor) is located via a GETADR macro which maps at assembly time a data set name into a displacement from the pointer into the data set address table. When the routine returns, an exit number (return code) is used as a displacement in the block list to determine what succession should be served on other tasks. The exits taken by a routine are generally a function of data, e.g., presence or absence of returns for a search function routine. The block list entry selected by the exit number has a pointer to a set succession list and a count of the number of continuous entries to process on this list. Each item in the set succession list has a bit mask and a pointer to the TLT entry for another task. The logical product (AND) of the mask and the TLT entry CEB work is formed, and if the result is all zeros, the task's AEB is set and the CEB work set back to the initial value, CEB_0, defined for the particular task. When all succession has been served, the processor continues execution with the routine just completed. Note that all routines in a task are completed, even though a higher priority task may have been enabled. When there are no more routines left in a task, the processor is free to rescan the AEB string to select a new task.

5.2.4 System Timing

The timing characteristics of a timed array are determined by (1) its start phase point, end phase point, and frequency (possible fractional number of executions per major cycle), (2) the precedence conditions of its constituent tasks, and (3) the task types of its constituent tasks. Precedence conditions have been described above, so we present now a discussion of the task types depicted in Figure 5.3, and their utilization. A SYS-START or SYS-ENABLE task in a TA has a special CEB bit which is set [only] by NEXOS at the start phase point of the TA. START and NEITHER tasks are enabled only by

		STARTED BY NEXOS?	
		YES	NO
DEADLINE CHECKED BY NEXOS?	YES	SYS-START	START
	NO	SYS-ENABLE	NEITHER

Figure 5.3. NEXOS task types.

fulfillment of precedence conditions by the completion of other tasks. Associated with STS-START and START tasks are initial termination counts used for deadline checking. When a SYS-START or START task is absolutely enabled, a termination counter is incremented. The completion of the task, and possibly the completion of other tasks causes the appropriate termination counter to be decremented. At the deadline phase point of a TA, all termination counters for tasks in the TA are examined, with non-zero counters indicating missed deadlines. When deadlines are missed, all tasks in the TA are disabled (AEB's set to zero). Most TA's begin with a SYS-START task followed by NEITHER tasks which also decrement its termination counter. Such TA's are said to be synchronous.

When a TA is turned-on, entries for the appropriate tasks are inserted in enablement lists and deadline lists for the start- and deadline-phase points, respectively, of the tasks. These lists are used by NEXOS to carry out the functions described above. TA's with a frequency of greater than once per major cycle have entries for multiple pairs of phase points, while TA's executed at fractional frequencies have counters that are decremented each major cycle, with task enablement being performed only when the counter has gone to zero.

One of the distinct differences between NEXOS and SENTOS is in the handling of data set conflicts. It is possible that any given object might, for example, be under consideration for interceptor allocation while continuing to be tracked, giving rise to the necessity to insure that certain programs cannot have simultaneous access to the same data set, i.e., they must access certain data in a serial fashion. In NEXOS, this must be accomplished by time frame exclusion, that is, the process designer must guarantee that certain tasks will never be absolutely enabled at the same time. The most apparent way of doing this is to assign mutually exclusive enablement-to-deadline intervals for the tasks involved. If all tasks of a certain function are constrained to run time-exclusive of the tasks of another function, then the process designer is forced to "stack up" similar tasks in the process. This appears to be the case in the MIP/4 process. To the extent that such stacking conditions are present, the checkerboard approach degenerates into a multiprocessor-implemented row approach, i.e., first all tracking computations are done, then all the searching, etc. The capability of data set locking and unlocking which exists in SENTOS allows a much greater granularity in time-exclusive task execution, since a task may be permitted to run immediately

after its data sets are free rather than being enabled at some future phase point (as in NEXOS) which guarantees time exclusion since it follows the deadlines of other contending tasks.

Within the checkerboard architecture it is advantageous, for two reasons, to enforce limitations on task run times. First, it is essential to the success of a multiprocessor strategy to break the entire job into many separately identifiable tasks which can be run in parallel. Second, granularity is desirable to guarantee a high frequency of task dispatching so that low priority tasks preclude the execution of higher priority tasks for as short a time as possible. There are two principal disadvantages to the enforced limitation of task run times. First, artificial partitioning of the workload may result. Second, and more important, the overhead in such a system is at least inversely proportional to mean task length. The overhead is reflected both directed in the increased frequency of dispatching of more tasks and in the internal task structure, since proportionately more of the task must be devoted to initiation and "cleanup."

5.3 OBSERVATIONS

1. The NEXOS-SENTOS checkerboard philosophy requires a very sophisticated scheduling algorithm. The term "algorithm" may be misleading, however, since scheduling appears to be done in a nonmechanized ad hoc manner in the nonreal-time activities of process design. Algorithms for optimizing (or even measuring, for that matter) load distribution, minimizing scheduling anomalies, etc., if employed have not been specified in documentation available to us. Analytic approaches to optimal scheduling have been outlined in articles by Jordan and R. L. Graham, among others, and the modification, application, and simulation of such algorithms should be explored in future efforts.

2. The NEXOS-SENTOS checkerboard philosophy also requires a sophisticated run-time dispatching algorithm to carry out all aspects of system timing, enablement, deadline checking, and time exclusive data set access. Because of the sophistication of dispatching, the deterministic approach of process construction (resulting in a fixed "menu" of system behavior) is buried in a vast probabilistic combinatorial problem.

3. It is not clear that all the restrictions of implementation under the checkerboard philosophy are inherent to the basic problem. One example of this is the discrete nature of system degradation under overload conditions: one either runs or does not run subsets of fixed sets of fixed tasks of fixed priorities with fixed timing and enablement constraints. It is not at all clear that a decrease in execution frequency of certain basic functions might be a reasonable alternative under a more dynamic approach.

4. Inversely, certain natural constraints may limit the range of applicability of the checkerboard philosophy. As an example of this, consider the "stacking" phenomenon discussed above. Since one pays dearly for a very general checkerboard system in terms of overhead, natural constraints which restrict applicability must be weighed very carefully.

6.0 SUMMARY AND GENERALIZATION OF SECTIONS 3-5

The summary of this brief study of variable load real time operating systems is divided into two parts. Section 6.1 consists primarily of some observations about the two systems studied and specific instances where the software organization appears troublesome to implement or to generalize and where alternate approaches might have been preferable. It should be emphasized that, although such observations are unavoidably criticisms, the authors are aware of such considerations as the state of computing systems at the time the design was undertaken, the pressure to adopt certain organizations because of existing software packages and the value of an early commitment to specific hardware and software organization—even with full knowledge that the selected structure is not ideal.

Section 6.2 reconsiders the general problem of variable load real time operating systems and suggests some alternate implementations using recently developed machine organizations.

6.1 SOME OBSERVATIONS ABOUT THE TWO SYSTEMS

6.1.1 The AN/FPS-85 System

This system must be judged to be successful. To the authors knowledge it is the only case where a beam-switching radar is routinely controlled by a computer in the task of tracking extra terrestrial objects. It is apparently conservative in that the load rarely exceeds the capacity and, if it does, it is admissible to simply skip radar cycles. Deadlines are detectable only with respect to 50 msec increments. As programmed the system is not readily extended to multiprocessors. The utilization of serially reentrant common routines precludes the simultaneous execution of such routines. Even in the single processor case the necessary completion of shared sequencers precludes high priority scheduling changes during the execution. The overlaying into fixed storage regions is, however, the most serious programming problem. The fixed boundaries complicate programming and limit the dynamic load-leveling capabilities desirable in a real time system. The system operates on a demand overlay strategy which is apparently adequate for the program responses required. The handling of intertask dependencies is accomplished for both data and program sequence by the passage of queue (and pseudo queue) blocks. Again there would be complications in the multiprocessor case. The dual processor model 65 is used for back up only. It seems that reliability and load handling would be improved if it could be run as a duplex (true) multiprocessor.

6.1.2 The Safeguard System

The Safeguard software exploits the specialized nature of the problem to an extreme degree. It is a true multiprocessor and hence is subject to the problems of simultaneous use of routines and data. The process construction operation is a prescheduling of functions and storage up to some maximum load. The simultaneous data set access problem is avoided, at least in part, by this procedure. The execution of functions are specified in certain time intervals relative to the major cycle but the dispatching to the several processors is handled dynamically. There is no dynamic storage assignment except for the overlay of some nontime-critical functions. The complete dependence on primary storage contributes to growth difficulties (i.e., more equipment) and, in a sense, is not necessary. The old time-sharing argument that a hierarchical storage machine is adequate provided that there are always tasks ready to accept a processor seems to apply here, too. It is clear that the "paging" implied by such a hierarchy should make use of the known structure of the system. A hierarchical system is more readily enlarged.

The storage management is one reason for the specialized process construction which, in itself, inhibits change in the system. It is unclear at this point, whether this exploitation of the special structure significantly improves performance over a simpler, more general, and dynamic scheduling.

The designed granularity of the system introduces programming impediments. The running time constraints lead to artificial partitioning of function.

6.2 THE GENERAL PROBLEM

The general problem comes under the heading of a scheduling problem and is related to, but more complex than, jobshop scheduling. Typically with the latter one has several machines, a number of jobs whose machine use characteristics are known and certain sequence dependencies among the jobs. The goal is to obtain a machine schedule with the highest utilization but without regard for processing deadlines. A variant which admits deadlines but not sequence dependencies is the knapsack problem where space (time) must be allocated to packages (jobs) having certain marginal values (priorities) such that maximum value is packed (executed). There is a linear programming approach [Ref. 5] to solve this problem but the extension to sequence dependencies seems most difficult. When this complication is added there is apparently no known algorithm to produce an optimal schedule but there are some for suboptimal (hopefully near optimal) assignments [Ref. 7].

It is probably worthwhile to extend the "checkerboard" model adopted in this report (see Section 3.1) to incorporate time as a third dimension,

because such an extension permits the statement of the problem as an assignment problem assuming quantized time and provides a basis for a discussion of scheduling strategies.

Consider a three-dimensional matrix $A = \{a_{m,n,t}\}$ where the dimensions represent, respectively, the functions, the objects, and the discrete time intervals. The indices are bounded $1 \leq n \leq N$ = some maximum numbers of objects. The operation is cyclical with $1 \leq t \leq T$ = the cycle time. This conceptual assignment array is depicted in Figure 6.1 below, with the unit elements designating the assignment of a particular function to a particular object over a specific time interval in the cycle.

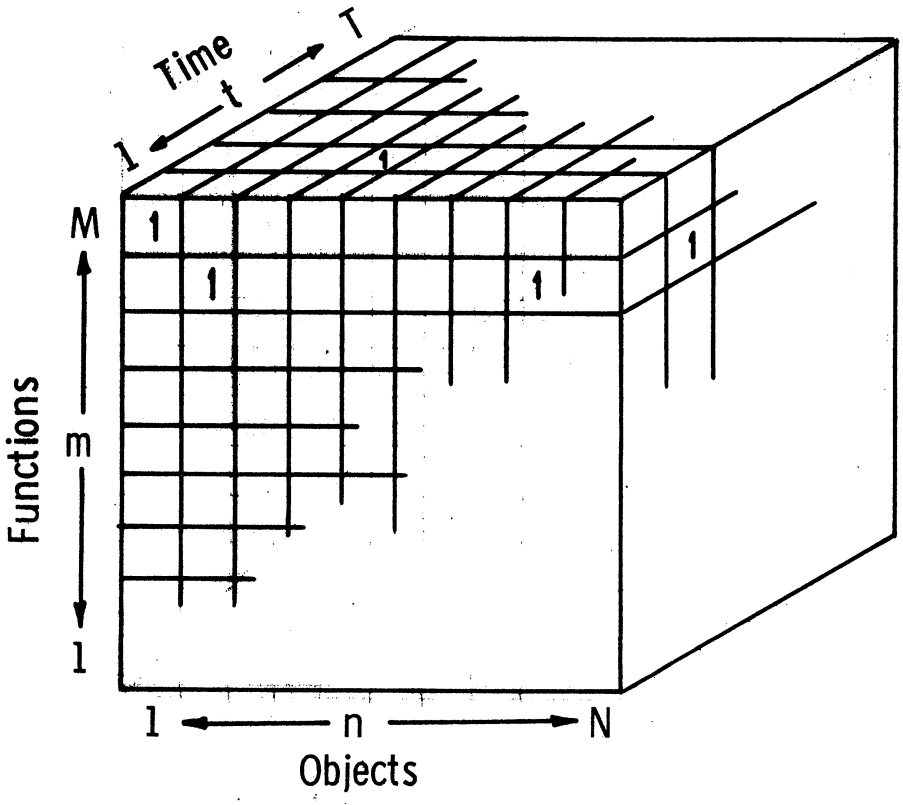


Figure 6.1. Assignment array.

If, in addition, it is assumed that there are P processors, the statement of an assignment problem becomes possible. In any given m-n plane only P functions can be invoked, hence

$$S_t = \sum_{m,n} a_{m,n,t} \leq P.$$

If the "object" axis of the assignment array is refined to represent the data sets which describe an object, then for shared data sets which are indivisible (incapable of being multiply accessed) over a given time interval, we have the further summation constraint that

$$S_{n,t} = \sum_{m} a_{m,n,t} \leq 1.$$

This is probably not an admissible assumption but the point will be reconsidered later. The sequence dependencies are somewhat more cumbersome to express but essentially certain may be unity only if certain elements preceding in time are unity.

$$\left\{ a_{m_2,n_2,t_2} = 1 \text{ iff } \left\{ a_{m_1,n_1,t_1} = 1 \right\} \right\}$$

The indexed subscripts can have any value in their ranges but $t_1 < t_2$. If m_2, n_2 are to assume all possible pairs then some provision for the insertion of a time function (a unit element in each column in each time step) could be included but that is a complication that can be avoided here.

The assignment problem is then to find a {0,1} valued matrix A subject to the given constraints and a given object computation

$$S_n = \sum_{m,t} a_{m,n,t} = K$$

such that

$$S = \sum_{m,n,t} a_{m,n,t} = NK$$

and P is minimal.

This assumes an unrealistic uniformity of objects but it permits a formulation. It is generally agreed that the objects constituting the load are largely independent and the sequence dependencies arise mainly from the processing within an object. In fact the scheduling algorithms such as those of Manacher [Ref. 7] would apply mainly in the m-t planes.

The complications arising from using multiple processors within an object suggest a strategy that appeared in computing history with multiprogramming. When a single task was handled on a channel-structured machine considerable program complexity was introduced to permit the simultaneous use of channels and processors. The decision to handle a number of programs at the same time permitted a return to the simpler sequential execution of single tasks and yet keep the device utilization high. Under the assumption that $N \gg P$ it seems that a similar assumption can be made for the real time case. That is, unless the object calculations are highly nonuniform or if, what amounts to the same thing, there are some extremely pressing deadlines for a small subset of tasks the object calculations can be carried out in a sequential (possibly often suspended) manners. These counter assumptions seem unlikely since, if they were true, the seemingly productive "row-wise" or pipe line processing approach would not be valid. In other words, a successful row-wise implementation implies a uniformity that would permit the simplifying sequential approach within an object task. Pursuing the analogy to multiprogramming (and time sharing) somewhat further this simplifying assumption would allow the fixed task population saturation and balance analysis like that of Kleinrock [Ref. 6] but more appropriately Moore [Ref. 8]. The obvious and much oversimplified relationship $N \leq PT/K$ could probably be refined by such analyses. However the assumptions need verification and the modelling examined in detail.

6.2.1 Machine Organization

The appearance of "de facto paging" in the FPS-85 and the overcommitment of storage in the Safeguard system suggest that properly designed virtual memory, used with anticipatory paging that exploited the known computational structure, would greatly simplify the system. Not only would this sort of hardware solve the storage allocation problem, but it would enable the function-to-object linkage to be greatly simplified. While the storage allocation function of virtual addressing is widely understood, the table-localized dynamic linking that is possible because of this indirection is not widely appreciated. For example, the functions (common to all objects) could be in the addressing space of every object computation but the object data could all be addressed with the same virtual addresses, with data <u>real</u> addresses provided by the supervisor dynamically. Thus a programmer could write the object computation as if it were a single, conventional program with bound addresses, and the linkage would be handled dynamically by a table substitution. Of course such table driven address transformation provide an inter-data protection that would be valuable in a BMD implementation.

Another hardware feature that would be desirable is priority controlled interrupt circuitry. The issue of granular programs versus interrupts and program state storage is a long standing one. The interrupts approach makes programming simpler but there still remains the supervisory issue of whether

controlling programs should run with interrupts enabled—thus permitting queue entries and dynamic scheduling changes—or disabled so that interrupts are stacked and a flood of interrupts is prevented. The usual answer is that both strategies are desirable on occasion and that a hardware implemented task list could produce the appropriate division between these approaches. A running processor would have a register containing the priority of the running program. The insertion in the task list of a higher priority program would cause the processor(s) to be interrupted. If the priority were lower the task would be stacked until a processor completion and no higher priority was active. Conventional interrupts would operate in the same manner: relatively high priorities in all processors would cause hardware stacking but lower priorities would permit interrupts. Rudimentary hardware algorithms of this sort currently exist in some small process control machines.

The time quantization of Figure 6.1 was necessary for the discussion of an assignment matrix. As has been stated it should be avoided—at least as an explicit program division. It would be necessary to establish a regular, schedule-checking cycle, however, and this could be handled within the priority mechanism just described. If the timer interrupt is (hardware) assumed to be at a priority just above the lowest priority processor, one processor would then make periodic checks of the schedule. This processor's initiation of a suitable high priority rescheduling task would then cause the proper number of processors to store their state and be reassigned. Of course such a task-list driven algorithm could be implemented in the software but only at the price of taking all interrupts and deciding which should be queued and which should initiate a reassignment.

Finally, on the subject of interrupts, it seems that the appearance of cache techniques on modern machines should permit the solution to the problem of state storage overhead. Just as a cache provides a continuous "store through" to memory a cache could provide a continually updated state of the machine. Therefore at interrupt time the time consuming sequential stores need not be taken.

REFERENCES

1. Alexander, M. T. "Time-Sharing Supervisor Programs," The University of Michigan Summer Conference Series Notes on Advanced Topics in Systems Programming. Ann Arbor: The University of Michigan, 1970.

2. Arden, Bruce W. and Hamilton, James. "A Study of Programming Language Effectiveness," ORA Project 03222 under contract with U.S. Army Safeguard Systems Command, Contract No. DAHC60-70-C-0036, Huntsville, Alabama. Ann Arbor: The University of Michigan,1970.

3. Bendix Corporation. "AN/FPS-85 Design Specifications." Private communication.

4. Eastwood, D. E., et al. "SENTOS Philosophy and Structure," Case 27703-1500. Bell Telephone Laboratories, October, 1968.

5. Gilmore, P. C. and Gomory, R. E. "A Linear Programming Approach to the Cutting Stock Problem—Part II." Operations Research 11, No. 6, (1963), 863-887.

6. Kleinrock, L. "Certain Analytical Results for Time-Shared Processors." Proceedings of IFIPS-68, August 1968, D119-D125.

7. Manacher, G. K. "Production and Stabilization of Real-Time Task Schedules." Journal of the Association for Computing Machinery, 14, No. 3, July 1967, 439-465.

8. Moore, Charles G. "Network Models for Large-Scale Time-Sharing Systems." Presented at the 38th National ORSA Meeting, Session on Analytical Models of Time-Shared Computer Systems, October 1970.

9. The University of Michigan Computing Center. "Michigan Terminal System." Ann Arbor: The University of Michigan.