A MULTI-TASKING EXECUTIVE

by

SEDAT YILMAZER

B.S. in E.E., Bogazici University, 1981

Submitted to the Institute for Graduate Studies in Science and Engineering in partial fulfilment of the requirements for the degree of

Master of Science

in

Computer Engineering

Bogazici University

1984

A MULTI-TASKING EXECUTIVE

APPROVED BY

Doc. Dr. Tunc Balman .....
(Thesis advisor)

Dr. Selahattin Kuru .....

Dr. Akif Eyler Y. Dr. Ömer CERİD .....

DATE OF APPROVAL : 3/7/84

ACKNOWLEDGEMENTS

I consider it a previlage to acknowledge those people who helped and encouraged me during my education and specially my thesis. First, I would like to thank Doc. Dr. Tunc Balman , my thesis advisor, for his kind help and understanding throughout my undergraduate and graduate study as well as my thesis. I would like to thank Doc. Dr. Yorgo Istefanopulos , my under graduate advisor, who encourged me on taking a master thesis. I also would like to thank Mr. Tanju Argun and Mr. Sait Gozum who let me use the facilities of the research and development laboratories of NETAS. I would like to thank Mr. Semih Pekol who first showed me the power of multi-tasking on microcomputers. Finally, I would like to thank Mr. Selcuk Yilmazer, my dear brother for his patience in proofreading this manuscript.

ABSTRACT

This thesis describes the design and implementation of a real-time multi-tasking priority-driven executive, FOX. The name stands for "a East Object oriented eXecutive. FOX permits the micro-computer user to enjoy the multi-tasking capabilities of the high performance machines like minicomputers. FOX tries to unify the concept of multi-tasking and team study of projects. FOX uses the concept of exchange for inter task communication and synchronization. For interrupts, FOX uses special exchanges called flags. There is no limit on the number of tasks and number of exchanges that can be created under FOX. There are 16 flags managed by FOX. Eigth of these flags are assigned to eight external interrupt sources via an interrupt priority controller. The remaining eight can be used by the programmers for fast and easy event processing. Each task running under FOX has an eight bit priority level, which is used by the task dispacher. All task exchange and interrupt binding is done dynamicaly by FOX to ease the team study of projects of large code size.

OZET

Bu tez oncelikli bir gercel-zaman isletim cekirdeginin , FOX ,tasarim ve gerceklenmesini tanimlar. FOX micro islemci kullanicilarina daha buyuk islemcilerin, ornegin mini islemcilerin, coklu kullanim imkanlarini saglar. FOX coklu kullanim ve grup calismasi kavramlarina duzenli bir yapi kazandirmaya calisir. FOX islemler arasi iletisim ve eszamanlama icin iletisim kutulari kavramini kullanir. FOX kesintiler icin ozel iletisim kutulari kullanir. Bu ozel kutulara bayrak adi verilir. Cekirdegin icinde bulunabilecek iletisim kutulari ve islem sayisi uzerine bir sinirlama yoktur. FOX 16 adet bayragin kullanilmasini saglar. Bunlardan 8 adedi 8 diskaynakli kesintiye bir kesinti oncelik cozumleyicisi ile ilistirilmislerdir. Artakalan 8 adet bayrak, kulanici tarafindan hizli iletisim amaclari icin kullanilabilinir, hizli kesinti iletisimi ornegi gibi. FOX altinda calisan her islemin 8 dilimlik ve islemci bolusumunde kullanilan, bir oncelik seviyesi vardir. Butun islem, iletisim kutusu ve bayrak baglantilari , buyuk yazinimlarin ve grup calismasinin kolaylastirilmasi icin, FOX tarafindan cekirdek icinde kendiliginden yapilir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

OZET v

I. INTRODUCTION 1

1 AIM 1

2 HISTORY OF MICROCOMPUTER HW DEVELOPMENT 1

3 HISTORY OF MICROCOMPUTER SW DEVELOPMENT 3

4 APPLICATION PROGRAMMING 4

II. CONCEPT OF ABSTRACT MACHINE 6

1 DEFINITION OF THE ABSTRACT MACHINE 6

III. FOX MACHINE 8

1 DEFINITION 8

2 USE ENVIRONMENT 10

3 SYSTEM REQUIREMENTS 10

4 FOX AS AN ABSTRACT MACHINE 12

5 METHODOLOGY 14

6 MODULARITY 15

7 ENTRY TO FOX 16

8 ENGINEERING FOR HARDWARE DEPENDINCIES 17

9 GLOBAL VERSUS LOCAL OPTIMIZATION 19

10 METHOD OF COMMUNICATION IN FOX 20

IV. FOX MANAGEMENT ROUTINES 22

1 FUNCTIONS OF FOX 22

2 TASK DISPATCHING 23

3 EXCHANGE MANAGEMENT 26

4 EVENT PROCESSING 28

1 FLAG EXCHANGE MANAGEMENT 29

2 FAST EXCHANGE MANAGEMENT 30

V. FOX OBJECT DEFINITIONS 32

1 USE OF OBJECTS 32

2 TASK RELATED OBJECTS 32

1 STATIC TASK DESCRIPTOR 33

2 TASK DESCRIPTOR 34

3 TASK LINK DESCRIPTOR 38

3 EXCHANGE RELATED OBJECTS 40

1 STATIC EXCHANGE DESCRIPTOR 40

2 EXCHANGE DESCRIPTOR 42

3 EXCHANGE LINK DESCRIPTOR 46

4 INTERRUPT RELATED OBJECTS 48

1 STATIC INTERRUPT DESCRIPTOR 49

2 INTERRUPT DESCRIPTOR 51

VI. FOX INSTRUCTIONS 53

1 INSTRACTIONS ON AN ABSTRACT MACHINE 53

2 CREATE GROUP 54

1 CREATE TASK 55

2 CREATE EXCHANGE 56

3 CREATE INTERRUPT 57

3 LINK GROUP 58

1 LINK TASK 59

2 LINK EXCHANGE 60

4 UTILITY GROUP 61

1 SUSPEND 61

2 RESUME 62

3 DISPATCH 63

5 EXCHANGE OPERATIONS 64

1 SEND OPERATION 64

a Send if free or return 66

b Perform Send and return 67

c Perform Send or time out 68

2 WAIT OPERATION 69

a Wait if free or return 71

b Wait until receive 72

c Wait message or time out 73

6 FOX FLAG OPERATIONS 74

1 EVENT 74

2 SIGNAL 75

VII. MORE ON SEND AND WAIT OPERATIONS 76

1 SEND - WAIT INTERRACTION 76

2 COMMUNICATION 77

3 SYNCHRONIZATION 77

4 MUTUAL EXCLUTION 79

VIII. SYSTEM START UP 81

1 SYSTEM INITIALIZATION 81

IX. SAMPLE APPLICATION 83

1 SAMPLE APPLICATION 83

2 DEFINITION 85

3 INITIALIZATION 89

4 TASK BODIES 90

X. CONCLUSION 92

APPENDIX A FOX PROPOSAL 94

APPENDIX B SOURCE LISTINGS 105

APPENDIX C SAMPLE APPLICATION 144

REFERENCES 148

I. INTRODUCTION

1.1 AIM

The aim of this master thesis is to apply the best of the software engineering to a realworld problem. Current interests in microcomputers and operating systems lead us to a thesis where it was possible to work with microcomputers and operating systems. The term operating system is quite wide for this design. A restricted term EXECUTIVE will be more suitable for the subject of the master thesis.

1.2 HISTORY OF MICROCOMPUTER HARDWARE DEVELOPMENT

After the introduction of first micro-computer 4004 by INTEL, we have taken such a long way that it is hardly possible to find another branch of technology that has progressed this fast in the history of mankind. If we revise the micro-computer history, we can see that after the introduction of these, so called, first generation 4 bit, multi-chip devices, we jump to the single chip 8 bit devices in less than 3 years. These, so called, second generation chips found a wide application area in process control applications and intelligent instrumentation.

Suddenly many of applications that were cost inefficient, became so cheap that we started to see "microprocessor controlled" ovens and washing machines. At the same time, control applications where we use those huge minicompters or hardware logic, became usual application areas of the microcomputers. In parallel to all these, a new area suddenly opened to the man on the street, the home computers !. These 8 bit machines were really ideal for intelligent instruments and for small-sized process-control applications, as well as the new area of home computers. But in a very short period of time, about 3 years, medium-scale control applications and very-intelligent instruments demanded more powerful processors, even more powerful than the early mini-computers. This combined with the desire of getting the most of the market share by introducing the first 16 bit microcomputer forced the manufacturers, and finally we got 16 bit computing power in the microprocessors. Of course, the drag of the huge software investment on 8 bit machines and the standart communication method of 8 bits between the computers will keep the 8 bit microcomputers up for a very long time, if not forever.

1.3 HISTORY OF MICROCOMPUTER SOFTWARE DEVELOPMENT

Parallel to hardware developments in micro-computer technology, microcomputer software has also developed rapidly. Today we have most of the compilers, at least their subsets, and many of the operating systems like UNIX on the micro-computer systems. Also special operating systems for microcomputers have been written, and, today we may speak of a standardization in microcomputer operating systems. CP/M and MP/M of DIGITAL RESEARCH,. ISIS-II and ISIS-III of Intel have found a very large user base in 80 series micro-computers, namely the I8080, Z80 , I8086, I8088. These operating systems are designed for single or multi user development systems or for commercial applications like engineering terminals, business class machines and for home computers. On the other hand more than half of the micro-computers going into the control applications, like process control and robot control. Needs of these applications are quite different. In such applications the system must respond to the completely unexpected events of the real world. It must perform different tasks of different importance concurrently. To answer these requirements, the operating system of the control application should handle real-time events, should dispatch the processing power to the most urgent task, and finally should use minimal possible processing power, both in the processing unit

and in the memory. One should notice that if a user waits at his terminal for a response of ten seconds, he just gets angry for a while, whereas if a process controller can not respond to an event in one seconds it might crash the whole system. Also it is clear that control applications, usually, if not always, require very high reliability figures from both the hardware and the software.

1.4 APPLICATION PROGRAMMING

In a real world application , to get most out of the hardware, usually the designer of the control application forces the limits of the micro-computers. That inturn means more software and more memory. On the other hand the designer should bring his product to the market as soon as possible. That forces the design manager to use more software engineers on the same project. Leader of the design team breaks the project into several parts and assigns one or more of these parts to one or more of the design members. Then on, the leader is responsible for the interaction of the parts and resolving the potential or appearing inconsistancies or contradictions. That migth seem to be quite easy at the first glance. But this job directly influences the product complexity and reliability. Many software

projects fall behind schedule due to seemingly very small interface problems and bad partitioning of the project. No one can argue that he can partition the project in the perfect combination, but some can say that it migth not be the perfect but can easily be changed when need arises. This is called modularity or adaptability. To help the design leader to generate a good project partitioning, we must supply some tools for him, He should not be lost in the details of the interaction of the partitions, these methods should be standartized, so both the project leader and the project manager can be sure that changing any of the partition, or more important than that, changing any of the team members will not cause excessive impact on the other parts or on the members of the team.

In this thesis, we tried to design and implement a real-time, priority driven executive. It is designed to take most of the underlying hardware, meaning, using minimal memory and CPU time. It is designed to standardize the task interface, and interrupt interface. In short, it is designed to ease the job of the project manager, project leader and the team members. Of course trying to ease everybody's job will bring some restrictions on the system and some overload to the CPU. But once the user masters the system, he will see that all the restrictions, except the time and memory limits, enforces a structured approach to the software development and the project management.

II. THE CONCEPT OF ABSTRACT MACHINE

2.1 DEFINITION OF THE ABSTRACT MACHINE

We generally identify a processing unit as a particular collection of hardware that implements the instruction set of a "machine". This very physical definition of a machine dates from mechanical processors of the past. Even with the modern computers, before the LSI, it was easy to physically point at the processor as distinct from memories, peripherals and programs. Continuing integration of functions into silicon, at least, made this physical distinction more difficult with single chip computers which contains memory, peripheral and program. Also microprogramming (that is replacing the hardwired instruction decoding with a more elementary programmed processor) as the implementation strategy has blurred the distinction by one more step. That is when the instruction set of a machine itself is implemented in terms of more primitive instructions, it is more difficult to identify the "machine". It is clear that this narrow physical definition of the machine is not adequate for the current technology levels and is likely to become even less adequate with the technologies of tomorrow.

Actually, we are not using the physical "machine" for a long time, instead we developed an alternative definition. Current programming methods use another concept, known as "abstract machine". Programmers of today, regard the machine that they are programming as much more than a collection of instructions that are defined in the reference manual. Indeed the physical machine is of little concern to the programmers.

Viewed as above, we can identify any collection of hardware and software that provides a well defined set of functions as defining an "abstract machine". In such a machine, instuction set consist of the functions provided by the hardware and software combination. For a particular application it may be possible to view multiple such abstract machines by taking various pieces of the whole.

III. FOX MACHINE

3.1 DEFINITION

The terms "operating system", "executive", "kernel", "nucleus" have been used to describe software systems of widely different functionality. These machines generally provide management of some machine resources such as I/O, memory or CPU time. So we might define an operating system as a collection of software modules which defines an abstract machine that includes resource management functions as well as the hardware supplied computational functions. Clearly the range of software systems by this definition is large. Rather than trying to resolve this disparity, term "executive" will be used. That is, this master thesis will describe a software system which provides a **"minimal"** base for the construction of real-time applications.

The important thing to realize from above discussion is that any operating system functionally enlarges the processor seen by the user. The functions that are provided become as much like a part of the machine's functionallity as 'ADD' instruction. Indeed it is functionally unimportant to the user desiring to create a task whether it requires a single instruction or a large software routine to accomplish it. In terms of the abstract machine discussion in the previous chapter we will exemine a software package which defines an abstract machine that includes functions required to coordinate programs, and perform real-time control application.

The key overall requirement of the executive will be that it will supply a "minimal covering set" of functions to permit coordination of asynchronous tasks. Like the instruction set of the base machine, the executive itself performs no work but rather provides an environment within which useful tasks can performed.

Here are some of the limitations of the system which differs it from large-scale operating systems. First, it is not primarily intended for a multi-user environment, because it is addressed to control applications . Also it does not assume a backing store from which program and data overlays can be loaded (although can easily support such extentions).

3.2 USE ENVIRONMENT

In terms of abstract machines we might characterize the hardware as essentially the same machine at the processor level, but different machines at the computer system level. It was desired that the abstract machines defined by adding FOX to the underlying computer be as much the same as possible. During the design of FOX , it is assumed that its user would span the entire broad range of applications. This implies that it might find uses ranging from minimal single board systems that function as single device controlers to complex multiboard applications implementing a real-time process or industrial control functions.

3.3 SYSTEM REQUIREMENTS

The hardware environment and anticipated uses of FOX defines a tight set of requirements. Foremost among these are its memory constraints; indeed for the anticipated uses, memory size considerations dominate the execution speed ones over a considerable wide range. Sice applications that would reside entirely on a single board with about 8kb of EPROM, the maximum

size of the FOX should not go over the half of that space, that is 4kb of EPROM. Moreover, unlike the minicomputer systems many applications of microcomputers would not have available any mass storage or other program loading device. So FOX is designed to be ROM (or EPROM) resident and capable of automatically initializing the system at power on.

Finally , to define the general functionality of FOX, we shall examine common characteristics of the anticipating applications. Real-time applications commonly need to perform a number of tasks of different importance logically in parallel, with the preference always given to executing the most critical one first. While these tasks can be relatively independent they may need to periodically synchronize themselves with one or another distinct task or with the outside world. For the second, interrupts are usually of the type of the hardware supplied mechanism. Some tasks may also need to communicate data with one another. Finally the tasks must have the ability to control themselves relative to real-time, either by delaying their execution for certain periods or by guaranteeing that they are not indefinitely delayed by, for example, a faulty device.

The best software engineering techniques are used to minimize the development effort. We feel that there was more to gain, both in development effort and code space, by avoiding optimized specialization of function in favour of more general designs. For that, most of time is spent on defining the objects that FOX will heavily depend on. The resulting external design, therefore, has a single mechanism to provide task communication, synchronization and time references. To do so, it incorporates the operating system design approches favoured in much of the modern computer litarature. Likewise, the internal structures are highly modular and designed to be as uniform as possible so as to avoid replicating similar, but nonidentical internal management routines.

3.4 FOX AS AN ABSTRACT MACHINE

The abstract machine defined by FOX augments the base microprocessor by introducing some additional computational concepts. A task is defined as an independent executable program segment. That is, a task embodies the concept of program in execution on processor. FOX permits multiple tasks to be defined which can run in parallel, or in multi-tasked fashion. That is, FOX makes individual tasks running on the same processor appear to be running on separate processors by managing the dispatching of the processor to a particular task. The registers on the

processor reflect the activity or state of the running task. Other tasks may be ready to execute, but for some reason have not been selected to run yet, so have their processor states saved elsewhere in the system. From the point of view of the program, that is a task, execution proceeds as though it were the only one being run by the processor. Only the apparent speed of the execution is affected by the multi-tasking. From the point of view of the system, every task is always in one of the three states: running, ready or waiting. The task actually in execution is running. Any other task which could be running but for the fact that the system has selected some other task to actually use the processor, is ready. Tasks which are delayed or suspended for some reason are waiting.

Each task is assigned a priority which determines its relative importance within the system. Whenever a decision must be made as to which task of those that are ready should run next, the one with the highest priority is given preference. Furthermore, in the spirit of the mechanism, the same priority scheme replaces a separate mechanism for disabling interrupts. Interrupts from external devices are logically given software priorities. If the application system designer wants a particular task as of important than responding to certain interrupts, he can specify this by simply setting the FOX priority of that task to be higher than the FOX priority associated with that interrupt.

3.5 METHODOLOGY

This section considers some details of the implementation of the system as an illustration of the design of such software products. First the methodology applied to effort will be discussed, and this will be followed by some samples of the mechanism.

To provide the abstract machine just described, and meet the other requirements of the system, FOX is implemented as a combination of ROM resident code and some RAM resident tables. Just as a hardware designer uses VLSI or LSI devices in preference to more elementary TTL components, we chose to use a high level language rather than elementary assemble coding. The system is therefore designed using the PL/M, Intel's high level system implementation language. The space constraints and a good level of internal maintainablity goal is achieved by maximizing the modularity of the design. The board independent functions of multi-tasking , communication and control are completely isolated from board dependent timing and interrupt handling functions. As a result, movement of the FOX to another board requires only re-implementation of these board dependent

functions. In addition , data structures of the internal and user visible objects are generalized so that single algorithm could deal with any of them. Individual optimizations could have been made in local design of many parts of the data structures to improve their space or time costs slightly. Such optimizations, however, would have cost considerably more in code space and code complexity.

3.6. MODULARITY

The module feature of PL/M is used to simulate the abstract data type concept and enforce information hiding. That is , every data structure used by the FOX is under the exclusive control of a single module. The modules supply to each other a restricted set of public procedures and variables. It is only through these paths that agents outside a module may access the data structures maintained by the module. The only assumption that outside agents may make about a module and its data structures are those specified by the definition of the public paths. As a result, as long as these interface specifications are maintained, any given data structure may be re-organized by re-designing its controlling module without affecting other parts of the system. This approach improves the understandability of the implementation and facilitates the

debugging and maintenance of the system.

3.7 ENTRY TO FOX

The method of activation of the executive from the user programs found to be very important for the control of the user activities. Two methods are considered. The first method was to define one entry for each system procedure supported. That would be done by a jump table at a fix memory location and tell the user where to call when he needs one of the system procedures. This method has the following drawback. At each system procedure call parameters should be checked, if the status of "in FOX" need be updated. Also giving so much information to the user violates the information hiding rule. The second method, which is called gating, states that only one entry should be given to the user for an executive. User should pass the required operation as parameter, and depending on this parameter the gate checks the legality of the call and marks all related information on the task descriptor and then activates the required operation. The drawback of this method is that one more parameter should be sent to the executive, which might increase the user code space a little bit. But, all system procedures are so designed that they require only one parameter, and with this method one more parameter of " what to do " need be given. The

overload on the code space was estimated to be negligable and gate method of executive activation was selected. The standart FOX system activation is defined to be as follows.

```
CALL FOX(ACTION$ID,PARAMETER);
```

The address of the gate is defined to be ROM start address plus three. So it is 0003 for a standart FOX implementation.

3.8 ENGINEERING FOR HARDWARE DEPENDENCIES

The two functions which vary most significantly across the single board computers (SBC) are the timing and interrupt facilities. To accomodate these variations, the implementation separates the logical and physical parts of these functions.

The interrupt facilities are split between the module which implements communication operations and a hardware interrupt handler module. The communication module provides a special 'interrupt send ' operation which performs the logical translation of interrupt event into a flag message. This facility is independent of the interrupt structure of the machine and remains the same in any version of FOX. The hardware dependent interrupt module deals directly with the hardware interrupt structure and invokes the Signal operation at a logical level. Only this module need be redesigned when

generating a FOX version for a different board. With this approach FOX takes full advantage of the hardware vectored priority structure on high performance products and can simulate this desirable structure at sligthly higher software cost on low performance products.

The same sort of variations are faced in providing a source of system time unit. Again, one module provides all of the logical timing functions associated with providing timing delays and time limits to the user. This module is independent of the type, frequency, or location of the physical source of the timer. A seperate module is responsible for clocking the logical level by invoking it once every system time unit. Once again, this permits a consistent definition of time unit in FOX, regardless of the sophistication of the available time source, and limits the amount of re-implementation needed to support a new product.

3.9 GLOBAL VERSUS LOCAL OPTIMIZATION

We have already discussed some aspects of global versus local optimization, at the overall design level in terms of avoidance of redundant features. A good example of this trade off in the implementation is provided by the linked list data structures within FOX. Like many such systems, there are a number of linked lists which must be maintained to reflect the status of the system. Local optimization on the placement of links within data structures or in the form of the heads used for the links would be guaranteed to save a few bytes of the data space across the various lists. Further, the list insertion, scanning, and deletion algorithms could be specially tailored to the individual list structures to save microseconds of execution time for some operations on some lists. Indeed, any such tailored algorithm might well use less code than a single more general one.

On the other hand, many of the list operations are , in no sense , time critical. Generalizing all the structures to use a single form replaces multiple algorithms with just one, thus saving code space. The particular form can be chosen to favor those operations that are frequent, thus limiting the impact of the generalization on the execution time of the system. Perhaps, most important, however, is that, by reducing the number of algorithms and structures used, we decrease the potential number of errors and improve the maintainability of the resultant code.

3.10 METHOD OF COMMUNICATION IN FOX

As mentioned before, tasks may desire to communicate information to one another. To do this FOX machine defines a message to be some arbitrary data to be sent between tasks. To mediate the communication of messages, FOX defines an exchange to be a conceptual link between tasks. An exchange functions somewhat like a mailbox in that messages are deposited by one task and collected by another. Its function is commplicated by the fact that a task may attempt to collect a message from an exchange that is empty or may attempt to deposit a message to an exchange which is full. In such cases the execution of that task is delayed until the exchange becomes free or depending on the type of the exchange operation, till the defined time limit has expired. Tasks that are so delayed are in wait state. This indirect communication mechanism is prefered over the one which directly addresses tasks (as in REMTEX written in NETAS by Mr AYDIN KUBILAY), because it permits greater flexibility in the arrangement of receiver and sender tasks. That is the receiving task need only know the interface specification for the function to be performed. Task or tasks which implement that function need not be known and thus may be conviniently changed if desired.

The conventional mechanism used by programs to communicate with the external devices is interrupt. Unfortunately, interrupts are by nature unexpected events and programming with them tends to be error prone. The essential characteristics of an interrupt is that a parallel, asynchronous activity (the device) wishes to communicate with another activity (a program). Since this communication should be faster than the normal exchange operations a special exchange is managed by FOX, namely the FLAG. Flags are special objects to link software programs to the events. FOX machine eliminates interrupts by changing them into flag messages, which indicate that an interrupt has occured. A flag is Signaled by a particular interrupt. Tasks which service the interrupts do so under FOX by an Event system call on a flag. Thus prioritized nested interrupts are handled.

The last concept embodied in the FOX abstract machine is that of time. The FOX machine defines time, interms of system time units. It then permits tasks to delay themselves for a given period of time, so that they can synchronize themselves with the outside world. It also guards against unduly long delays caused by attempting to collect a message or deposit a message to an empty or full exchange, by limiting the length of time that they are willing to spend waiting for some message to arrive or space become available.

FOX is a multi-tasking executive that performs task dispatching and system timing. It also performs exchange management and flag management. The following sections describe these functions in detail.

Although FOX is a multi-tasking executive, at any given point in time, only one task has access to the CPU resource. Unless it is specifically written to communicate or synchronize execution with other tasks, it runs unaware that other tasks may be competing for the system's resources. Eventually, the system suspends the task from execution and gives another task the opportunity to run.

IV. FOX MANAGEMENT ROUTINES

4.1 FUNCTIONS OF FOX

FOX is a multi-tasking executive that performs task dispatching and system timing. It also performs exchange management and flag management. The following sections describe these functions in detail.

Although FOX is a multi-tasking executive, at any given point in time, only one task has access to the CPU resource. Unless it is specifically written to communicate or synchronize execution with other tasks, it runs unaware that other tasks may be competing for the system's resources. Eventually, the system suspends the task from execution and gives another task the opportunity to run.

4.2 TASK DISPATCHING

The primary task of the executive is transfering the CPU resource from one task to another. This is called DISPATCHING and is performed by a part of the executive called the dispatcher. Under FOX each task is associated with an object called a TASK DESCRIPTOR. The dispatcher uses this object to save and restore the current state of a running task. Every task in the system resides in one of three states : ready , running or suspended. A ready task is the one that is waiting for the CPU resource. A suspended task is the one that is waiting for some other system resource or for a defined event. A running task is the one that the CPU is currently executing.

Dispatch operation for a runnig task can be described as follows:

1-) The dispatcher suspends the task from execution and stores the current state in Task Descriptor.

2-) By definition the ready queue head is the higest priory task in the system, So the dispatcher selects that task as running.

3-) The dispatcher restores the state of the selected task from its Task Descriptor and gives the CPU resource.

4-) The task executes until it makes a system call, or an interrupt, or a tick of the system clock occures. Then, dispatching is repeated.

Only tasks that are placed on the Ready List are eligible for selection during dispatch. By definition, a task is on the Ready List if it is waiting for the CPU resource only. Tasks waiting for other system resources can not execute until their resource requirements are satisfied. Under FOX, a task is blocked from execution if it is waiting for :

1-) A exchange message, so that it can complete a read exchange operation.

2-) Space to become available in an exchange so it can complete an exchange write operation.

3-) An interrupt flag to be set.

4-) A specific number of system ticks before it can be removed from the system delay list.

5-) Another task to resume its execution.

These situations are discussed in more detail in the following sections.

FOX is a priority driven system. This means that the dispatcher selects the highest priority ready task and gives the CPU resource. Tasks with the same priority are round-robin scheduled. That is, they are given equal CPU time slices when executing CPU bound code. With priority dispatching, control is never passed to a lower priority task if there is a higher priority task on the ready list. Since high priority compute bound tasks tend to monopolize the CPU resource, it is advisable to lower their priority to avoid degrading overall system performance. In addition, compute bound tasks can make FOX dispatch calls periodically to promote sharing of the CPU

resource. When a task makes a dispatch call, the call appears as a null operation to the task, but allows other tasks to gain access to the CPU resource.

FOX requires that at least one task be running at all times. To ensure this, FOX maintains the IDLE task on the ready list so it can be dispatched if there are no other tasks available. The IDLE task runs at the lowest priority and is never blocked from execution. It does not perform any useful task, but simply gives the system a task to run when no other ready tasks exist.

4.3 EXCHANGE MANAGEMENT

Exchanges perform several critical functions for tasks running under FOX. They are used for communicating messages between tasks, for synchronizing task execution, and for mutual exclusion.

Exchanges are special objects, implemented in FOX as memory files that contains room for a specific number of fixed length messages. Like files, Exchanges can be created, read from and written into with FOX system calls. When an exchange is created with the FOX create exchange command, it is assigned a six

character name that identifies the exchange in FOX. Messages are read from an exchange on first in first out basis.

A task can read messages from an exchange or write messages to an exchange in three ways : conditionally,unconditionally or with time-out. If no messages exist in the exchange when a conditional read is performed, or the exchange is full when a conditional write is performed, the system returns an error code to the calling task. If a task performs an unconditional read from an empty exchange , the system suspends the task until another task writes message to that exchange. A task suspended in this manner is placed on the exchange's dequeue list. A similar situation occurs when a task makes an unconditional write to a full exchange. A task suspended in this manner is placed on the exchange's enqueue list. If a task performs an unconditional read from an empty exchange with time-out, the system suspends the task until either a time-out occurs, or another task writes a message to that exchange. A task suspended in this manner is placed on both the exchange's dequeue list and the system delay list. A similar situation occurs when a task makes an unconditional write to a full exchange with time-out. A task suspended in this manner is placed on both the exchange's dequeue list and the system delay list. FOX uses these enqueue/dequeue lists to synchronize task execution.

4.4 EVENT PROCESSING

FOX suuports events of any kind, including the interrupts, in two ways. One, via flags which are special exchanges (as in the Digital Research's MPM-II) where no more than one task is allowed to wait. The other method is like interrupt routines, the user service is immediately executed at the priority level of the calling task, with interrupts masked or disabled.

The first method converts events into messages so that a strict priority scheduling can be achieved and a logical event mechanism can be implemented at a cost of a little bit time over head of flag management. Where as the second method is useful1 when the, overhead of flag management can not be tolareted due to tight timing requirements of event processing, such as fast interrupt servicing.

In the following sections, both of the methods are discussed in detail.

4.4.1 Flag Exchange Management

This is a unified approach to event or interrupt servicing. For this kind of flag handling the type of the flag should be set as FLAG$EXCHANGe. FOX maintains 16 predefined flags, of which 15 are user accessable. The flag 15 is reserved for future extentions. Each of the 15 flags (0 through 15) can be used freely by the user. For flag management, FOX has two simple system procedures, the EVENT and the SIGNAL. A task which wants to wait on a flag issues an EVENT system call, and a task which wants to signal an event issues a SIGNAL system call. Details of Event and Signal system calls are desicribed at the FOX operations.

To avoid flag-under/over-runs the user is advised to follow the below listed rules, for his own sake.

1-) Do not use flags if you can handle the case in some other way, such as normal exchange messages, ofcourse, within the defined limits of the problem such as the time constraints.

2-) Make sure that only one task can issue EVENT system call on the selected flag. That guaranties that no flag-under-run can occur.

3-) Make sure that the event service time is strictly less than the signal rate. This guaranties that there will never be case of flag-over-run.

4-) Avoid the use of flag 15 which is reserved.

4.4.2. Fast Event Processing

For this type of operation the flag type is defined as FAST$SERVICE. In this mode the service routine is immediately activated at the current priorioty level. In this mode, only the SIGNAL system call is used. EVENT system call has no meaning. SIGNAL system call appears like a normal subroutine call to the calling task. This mode is useful when the event rate can not be handled via normal EXCHANGE$FLAG method.

FOX executive uses flags for signaling and synchronizing tasks with defined events. Tasks access the flags with the FOX system calls EVENT and SIGNAL Internally a flag can reside in one of two states ; set or reset. The reset state is further divided into two categories:

1-) No task is waiting for the flag to be set.

2-) A task is waiting for the flag to be set, and blocked from execution until it is set.

Two tasks are not allowed to wait on the same flag. This is an error situation refered to as "flag under-run". Similarly, a task attempting to set a flag that is already set is another error situation, called "flag over-run".

Flags provide a logical interrupt system independent of the physical interrupt system of the processor. They are primarily intended for use by FOX to support the interrupt handler. For example, when the interrupt handler receives a physical interrupt indicating that an I/O operation is completed, it sets a flag and branches to the dispatcher. A task suspended from execution because it is waiting for the flag to be set, is placed on the ready list, making it eligible for selection during dispatch. Once dispatched, the task can assume the I/O operation is completed.

V. FOX OBJECTS

5.1 USE OF OBJECTS

This section contains information on FOX objects. FOX uses these objects to :

1-) Synchronize the tasks

2-) Communicate messages between tasks

3-) Synchronize the events

5.2 TASK OBJECTS

Every task in FOX has at least two task objects and possibly one or more task link descriptors to form a logical link to other tasks.

Following sections explain the functions and the details of the task objects, namely Task Descriptor, Static Task Descriptor and Task Link Descriptor.

5.2.1 Static Task Descriptor

Each task running under FOX is associated with a Static Task Descriptor. This object defines all static characteristics of the task. FOX uses this object to initiate the actual [dynamic] Task Descriptor of the task. The structure of the Static Task Descriptor is given below in PL/M programming language

```
DECLARE     STATIC$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
      L             PTR             /*LINK TO EXCHANGE */
    NAME(6)         CHARACTER ,     /* TASK NAME */
    P               BYTE,           /*PRIORITY*/
    PC              PTR,            /*INITIAL PC*/
    SP              PTR,            /*CURRENT STACK */
    EH              PTR             /*EXCEPTION HANDLER*/
                                    ) ';
```

Elements of the static task descriptor defined above are described below.

L — 2 bytes, task descriptor link, points to the dynamic task descriptor of the task. This descriptor will be used by the FOX and be initiated from the Static Task Desctiptor at the time of task creation.

NAME	6 Character, task name, contains the name of the task
P	1 byte, priority, contains the priority of the task
PC	2 bytes, initial program counter, points to the first executable code of the task.
SP	2 bytes, stack pointer, contains the initial stack pointer value.
EH	2 bytes, exception handler, points to the task exception handler start address,

5.2.2. Task Descriptor

Each task running under FOX is associated with a task descriptor that defines all characteristics of the task. The FOX uses the task descriptor to save and restore the state of the task. The task descriptor object is shown below in PL/M language.

```
DECLARE        TASK$DESCRIPTOR STRUCTURE (
      L                PTR,          /* TASK LINK */
      NAME(6)          CHARACTER ,   /* TASK NAME */
      SL               PTR,          /*SYSTEM LINK */
      D                PTR,          /*DELAY LINK */
      E                PTR,          /*EXCHANGE LINK*/
      P                BYTE,         /*PRIORITY*/
      ST               BYTE,         /*STATUS */
      PC               PTR,          /*INITIAL PC*/
      SP               PTR,          /*CURRENT STACK */
      EH               PTR,          /*EXCEPTION HANDLER*/
      C                INTEGER,      /*DELAY COUNT */
      R(3)             PTR           /*RESERVED */
                                     );
```

The elements of the Task Descriptor object shown above are defined below.

L — 2 bytes, FOX task link pointer

NAME — 6 character, task name, set by the user

SL — 2 bytes FOX system list pointer

D — 2 bytes, Delay Queue link, tasks that are on the delay list are linked through this pointer.

E — 2 bytes, Exchange Pointer, used for the tasks that are enqueueing or dequeueing on an exchange to indicate on which of the exchanges the task is waiting.

P — 1 byte, Priority, indicates the priority of the task. A high value indicates a high priority. Priorities range from 0 to 255.

ST — 1 byte, Status of the Task, shows the current status of the task. A task can be in one or more of the following states.

1-) Ready : Task is eligible for dispatch.

2-) Notready : Task is not eligible for dispatch.

3-) Suspended : Task is suspended from execution and either on system suspend list or on exchange enqueue/dequeue list.

4-) Notsuspended : Task is neither on system suspend list nor on an exchange enqueue , dequeue list.

5-) Delayed : Task is on system delay list

6-) Notdelayed : Task is not on system delay list.

7-) In FOX : Task runs in FOX region

8-) Not In FOX : Task runs out of FOX region

PC 2 bytes, initial program counter value, Points to the first executable code of the task.

SP 2 bytes, initial Stack Pointer, points to the initial stack of the task. It will be used byte dispatcher to save the internal state of the CPU resource at the time of dispatch.

EH 2 bytes, Exception Handler, FOX uses this pointer to execute user exception routine in case of an abnormal condition. If no exception handler is defined, this pointer should be initiated as NIL.

C 2 bytes, Delay Count, this field is used by the delay manager, also indicates the system time-ticks to pass before the task enters the ready list, to become eligible for dispatch.

5.2.3. Task Link Descriptor

To ease the dynamic binding of the tasks, a special object is defined in FOX. Using FOX system call LINK$TASK, one task can create a link to another task. By using this object as parameter to FOX system procedures, a task can SUSPEND or RESUME another task. To link a task to another, user creates a task link object and fills the name field. Activating FOX system procedure LINK$TASK will supply the other neccessary pointers. Once these pointers are initiated, user can Resume or Suspend another task.

The structure of the object is given below in PL/M language.

```
DECLARE TASK$LINK$DESCRIPTOR STRUCTURE (
        L               PTR,        /* PTR TO TASK DESCRIPTOR *
        NAME(6)         CHARACTER,  /*EXCHANGE NAME */
        S               BYTE,       /* TASK STATUS */
        EH              PTR         /* EXCEPTOIN HANDLER  */
                                    );
```

Elements of the task link descriptor defined above are described below.

L	2 bytes, pointer to the task descriptor of the task to be linked. Set by FOX.
NAME	6 Character, Name of the task to be linked.
S	1 byte, Status of the task linked, at the time of the link operation. A zero value indicates that the desired task is not currently on the system.
EH	1 bytes, pointer to the exception routine in case of an abnormal condition.

5.3 EXCHANGE RELATED OBJECTS

An exchange is a first in first out [FIFO] mechanism that is implemented in FOX to provide severel essential functions in the multi-tasking environment. Exchanges can be used for the communication of messages between the tasks, to synchronize the tasks and to provide mutual exclusion.

FOX is designed to simplify exchange management for both user and the system tasks. Exchanges are treated like files, so they can be created, opened, written into and read from.

Exchange objects used by FOX include the Exchange Descriptor, Static Exchange Descriptor and Exchange Link Descriptor.

5.3.1 Static Exchange Descriptor

Each exchange in FOX has a static exchange descriptor which defines all of the static characteristics of the exchange. The dynamic Exchange Descriptor is created from the Static Exchange Descriptor at the creation phase.

Structure of the static exchange descriptor is given below in PL/M language.

```
DECLARE STATIC$EXCHANGE$DESCRIPTOR LITERALLY ' STRUCTURE (
  L         PTR,              /*  LINK  TO  EXCHANGE */
  NAME(6)   CHARACTER,        /*EXCHANGE NAME */
  SZ        BYTE,             /*MAX MESSAGE # */
  LNG       BYTE,             /*MESAGE  LENGTH */
  EH        PTR               /*EXCEPTION HANDLER */
                                                  ) ';
```

Elements of the static exchange descriptor defined above are described below.

L — 2 bytes, Link to the exchange descriptor

NAME — 6 characters, Name of the exchange. Initialized at the system definition.

SZ — 1 byte, maximum number of messages that can be deposited at the exchange

LNG — 1 byte, message length.

EH — 2 bytes, pointer to exception handler.

5.3.2 Exchange Descriptor

Each exchange in FOX is associated with an Exchange Descriptor object that defines all the characteristics of the exchange. FOX uses this object to keep track of the message buffer pointers, to save the status of the exchange and to save the exchange enqueue/dequeue links. The exchange descriptor object is given below in PL/M language.

```
DECLARE EXCHANGE$DESCRIPTOR STRUCTURE (
  L               PTR,          /*  EXCHANGE LINK */
  NAME(6)         CHARACTER,    /*EXCHANGE NAME */
  SL              PTR,          /*SYSTEM LINK */
  MF              PTR,          /*MESSAGE HEAD PTR */
  MB              PTR,          /*MESSAGE TAIL PTR */
  SZ              BYTE,         /*MAX MESSAGE # */
  BSZ             PTR,          /*BUFFER SIZE*/
  C               BYTE,         /*CURRENT MESSAGE COUNT*/
  K               BYTE,         /*EXCHANGE TYPE    */
  S               BYTE,         /* "        STATUS */
  L               BYTE,         /*MESSAGE   LENGTH */
  EH              PTR           /*EXCEPTION HANDLER */

                                );
```

Elements of the exchange descriptor defined above are described below.

L	2 bytes, exchange link
NAME	6 characters, Name of the exchange, copied from Static Task Descriptor.
SL	2 bytes, System Link, FOX keeps a list of all processes and exchanges, used by FOX.
MF	2 bytes , Message head pointer, points to the exchange buffer, the address of the next message to be read, used by FOX.
MB	2 bytes, Message tail pointer, points to the exchange buffer, the address of the next message to be written, used by FOX.
SZ	1 bytes, Exchange Size, indicates the maximum number of messages that can be kept in the exchange message buffer, set by the user
BSZ	2 bytes, Exchange Buffer size, by definition, buffer size is Exchange size times the exchange message length, set by FOX.
C	1 byte, current number of messages in the exchange buffer waiting to be read, used by the FOX.

K 1 byte, exchange kind, exchanges can be of three types:

1-) Perform operation or return type. This type of exchanges do not suspend the calling task if there is no message in the exchange when the task attempts to read message from the exchange or if there is no room for a new message when the task attempts to write into the exchange. Instead, the exchange returns a status to indicate that whether the exchange operation is completed successfully or failed to be performed. Status is returned in the status byte of the exchange link descriptor.

2-) Wait until the exchange operation is successfully completed.

3-) Wait either the exchange operation is successfully completed or until the time-out, indicated in the exchange link descriptor, occures. Return "success" status if exchange took place or "failed" status if a time out occured before the exchange operation.

S 1 byte, Status, shows the current status of the exchange. It can take one of three values, set by FOX.

1-) exchange is empty

2-) exchange is full

3-) exchange is free [neither full nor empty]

L 1 byte, message length, indicates the length of each message of the exchange, set by user.

EH 2 bytes, exception handler, points to the exception handler which will be activated if an abnormal condition occures during exchange operation, set by user.

5.3.3. Exchange Link Descriptor.

To ease the exchange operations and free user from absolute exchange descriptor address manupulations, FOX supports a special object called exchange link descriptor. As in the task link descriptor, if a task wants to send a message to an exchange, it prepares an exchange link block, specifies the name of the exchange that the message is to be sent, and calls the FOX system procedure LINK$EXCHANGE. FOX provides the necessary pointers so that user can invoke exchange write or exchange read operations of FOX.

The structure of the Exchange Link Descriptor is given below in PL/M language.

```
DECLARE EXCHANGE$LINK$DESCRIPTOR STRUCTURE (
  L                PTR        /* PTR TO  EXCHANGE */
 NAME(6)           CHARACTER, /*EXCHANGE NAME */
  M                PTR,       /* PTR TO MESSAGE */
  K                BYTE,      /* EXCHANGE TYPE      */
  S                BYTE,      /* EXCHANGE STATUS */
  EH               PTR        /* EXCEPTION HANDLER  */
                              );
```

Elements of the exchange link descriptor defined above are described below.

L — 2 bytes, exchange pointer, points to the desired exchange, set by FOX.

NAME — 6 characters, Name of the exchange to be linked, set by user

M — 2 bytes, points to the message buffer to be sent or received, set by user.

K — 1 byte, desired exchange method, if not nil this parameter overrides the exchange kind ,if nil then the method specified in the exchange descriptor is assumed.

S — 1 byte , return status of the exchange operation, set by FOX.

EH — 2 bytes, exception handler, points to the exception handler of the link. if not set to nil then this exception handler overrides the exchange exception handler, if not given then the exception handler of the exhange is assumed.

5.4 INTERRUPT OBJECTS

An interrupt is a signal from one process to another. It can be from an internal process, or in general from an external process to a internal process. By nature, they are random signals at random intervals. Also the mechanism to respond to those interrupts differ from processor to processor. To hide the actual machanism implemented on a micro-computer, they are converted to special signals by FOX. This method can be summarized as follows. An external event creates an interrupt signal indicating the completion of a certain operation. When the micro-computer recognizes the interrupt it completes its last instruction and then starts interrupt acknowledge sequence. At the end of this sequence current state of the process is saved on the stack and a special part of FOX, called interrupt handler is activated and depending on its level, it generates a Signal call to FOX. At that instance the physical interrupt signal is completely converted to a software signal to the related interrupt service routine. To indicate FOX whereabouts of the interrupt service routines two special data structures, called an Interrupt Descriptor and Static Interrupt Descriptor is used.

Special care is paid to decrease the interrupt latency to increase interrupt service rate.

5.4.1 Static Interrupt Descriptor

Each interrupt, physical or software, is associated with a Static Interrupt Descriptor object that defines all static nature of the interrupt service. FOX uses this object to determine the service routine, its interrupt priority level and its service type. The Static Interrupt Descriptor object is given below in PL/M language.

```
DECLARE   STATIC$INTERRUPT$DESCRIPTOR STRUCTURE(
    L           PTR,        /*POINTER TO DYNAMIC DESCRIPTOR */
    NAME(6)     CHARACTER, /*INTERRUPT NAME                  */
    T           BYTE,       /* INTERRUPT TYPE               */
    SRV         PTR,        /* INTERRUPT SERVICE ROUTINE    */
    EH          PTR ,       /* EXCEPTION HANDLER            */
    LVL         BYTE        /* INTERRUPT LEVEL              */
                                    );
```

Elements of the static interrupt descriptor defined above is given below.

L — 2 bytes, Pointer to the dynamic interrupt descriptor. Initiated by user.

NAME — 6 characters, interrupt name, Initiated by user.

T — 1 byte, interrupt service type, can be assigned as exchange type or fast type as explained in the Signal and Event procedures.

SRV — 2 bytes, if the exchange type is fast type then contains the address of the service routine, else has no meaning. Initiated by user.

EH — 2 bytes, pointer to the exception handler. It will be used on flag-over/under-run conditions. If not used then should be initialized as nil, or else should be initialized with the address of the exception handler.

LVL — 1 byte, interrupt priority level, initiated by user.

5.4.2. Interrupt Descriptor

Each interrupt in FOX system is associated with an interrupt descriptor to define its dynamic status. FOX uses this object to locate the service routine of the interrupt and its flag status. The structure of the interrupt descriptor is given below in PL/M language.

```
DECLARE    INTERRUPT$DESCRIPTOR      STRUCTURE(
     L            POINTER,        /* RESERVED FOR COMPATIBILITY */
     NAME(6)      CHARACTER,      /* INTERRUPT NAME             */
     SL           POINTER,        /* SYSTEM LINK                */
     TASK         POINTER,        /* SERVICE TASK               */
     SRV          POINTER,        /* SERVICE ROUTINE            */
     EH           POINTER,        /* EXCEPTION HANDLER          */
     F            BYTE,           /* FLAG                       */
     T            BYTE,           /* TYPE                       */
     LVL          BYTE            /* LEVEL                      */
                    );
```

Elements of the interrupt descriptor defined above is explained below.

L — 2 bytes, reserved for compatibility with the other descriptors,

NAME — 6 characters, interrupt name.

SL — 2 bytes, System Link, used by fox to keep all defined objects under FOX.

TASK — 2 bytes, if interrupt type is defined as exchange then contains the address of the service task's Task Descriptor address of the interrupt, otherwise has no meaning.

SRV — 2 bytes, points to the interrupt service routine, if the type of the interrupt service is defined as fast service.

EH — 2 bytes, pointer to the exception handler. Activated when a flag error occures.

F — 1 byte, indicates the state of the interrupt flag as defined at the Signal and Event system procedures.

T — 1 byte, interrupt type, initiated by user.

LVL — 1 byte, interrupt priority level. initiated by user.

VI. FOX INSTRUCTIONS

6.1 INSTRUCTIONS IN AN ABSTRACT MACHINE

The concept of abstract machine is realized in FOX by introducing some new data objects and instructions. Just as the base processor can deal with such data objects as 8 bit bytes or unsigned integers, FOX abstract machine can deal directly with the more complex objects ; tasks , messages , exchanges and flags. Each of these data objects consists of a series of bytes with a well defined structure, and may be operated on by certain instructions. This is completely analogous, for example, to a machine that permits direct operations on floating point data objects which consist of a number of bytes with a particular internal structure to represent the fraction, exponent and the signs. In each case there are only certain instructions that can be used correctly with the object and the internal structure of the object is not of particular interest to the programmer.

The new instructions that are provided by FOX are SEND, WAIT, CREATE_TASK,CREATE_EXCHANGE, CREATE_INTERRUPT, LINK_TASK , LINK_EXCHANGE , SUSPEND, RESUME, SIGNAL , EVENT. Create instructions accept blocks of free memory block and some creation information to format and initialize the block with the appropiate structure. Link instructions link a task or an

exchange or an interrupt to the calling task. The remaining instructions are of most interest to the operation of FOX. The SEND and WAIT instructions are discussed in detail at the and of this section and in the section entitled Send wait interaction.

In addition to the above mentioned standart FOX system instructions, there are optional system instructions. These instructions are hardware dependent, so they are provided if the underlying hardware can support the instruction. LEVEL$ON , LEVEL$OFF and END$INT system instructions are such instructions in the initial prototype.

6.2 CREATE GROUP

This group of FOX system calls introduces a new object to FOX system. Group contains three system procedure namely CREATE$TASK, CREATE$EXCHANGE and CREATE$INTERRUPT. Each of these system calls is described in detail in the following sections.

6.2.1. "CREATE TASK" System Call

This FOX system call introduces a new task into the FOX system. The newly created task is inserted into the system suspend queue. It will become a ready task when a running task resumes its execution by a FOX RESUME system call.

Gate entry code of the CREATE$TASK is defined to be 01.

To create a task in FOX system, user should prepare a static task descriptor to define all static nature of the task. The structure of the static task descriptor is given in FOX object definitions. If a task tries to create a task with a name with which there is already another entry in FOX system, then the exception handler of the calling task is activated , if it has any.

EXAMPLE

```
DECLARE TICKER$STATIC$TASK$DESCRIPTOR TASK$DESCRIPTOR
 DATA(.TICKER$DESCRIPTOR,          /* TASK DESCRIPTOR ADDRESS */
      'TICKER',                    /* TASK NAME */
      3,                           /* TASK PRIORITY */
      .TICKER,                     /* TASK START ADDRESS */
      .TICKER$STACK,               /* TASK STACK ADDRESS */
      .TICKER$EXCEPTION            /* TASK EXCEPTION HANDLER */
                                   );
 CALL      FOX(CREATE$TASK,.TICKER$STATIC$TASK$DESCRIPTOR);
```

6.2.2. "CREATE EXCHANGE" System Call

This FOX system call introduces a new exchange to FOX system. Then on, other tasks can link themselves to that exchange via LINK$EXCHANGE FOX system call. Then, exchange can accept SEND and WAIT operations.

Gate entry code of CREATE$EXCHANGE is defined to be 02.

To create a new exchange, user should prepare a Static Exchange Descriptor to define all static characteristics of the exchange. The structure of the static exchange descriptor is defined in FOX object definitions. If any task tries to create an exchange whose name is already defined in FOX system, the exception handler of the calling task is activated, if it has any.

EXAMPLE

```
DECLARE CONSOLE$STATIC$EXCHANGE$DESCRIPTOR STATIC$EXCHANGE$DESCRIPT
  DATA(.CONSOLE$EXCHANGE$DESCRIPTOR, /* LINK TO EXCHANGE DESCRIPTOR
  'CONSOL',                          /* EXCHANGE NAME */
  2,                                 /* MAX # OF MESSAGES *
  80,                                /* MAX MESSAGE LENGTH */
  .CONSOLE$EXCEPTION                 /* EXCEPTION HANDLER */
                                        );

 CALL  FOX(CREATE$EXCHANGE,.CONSOLE$STATIC$EXCHANGE$DESCRIPTOR);
```

6.2.3. "CREATE INTERRUPT" System Call

This procedure introduces an interrupt to FOX system. Then on the corresponding interrupt will be served depending on the parameters passed via the static interrupt descriptor. The cause of the interrupt need not be a real physical interrupt, it can also be SIGNALed by another task or even by the task itself. The structure of the static interrupt descriptor is given in the FOX object definitions.

Gate code of the CREATE$INTERRUPT is defined to be 03.

If a user tries to create an interrupt with a name which is already in the system, then the exception handler of the calling task is activated, if it has any.

EXAMPLE

```
DECLARE CONSOLE$STATIC$INTERRUPT$DESCRIPTOR
                                    STATIC$INTERRUPT$DESCRIPTOR
DATA(    .CONSOLE$INTERRUPT$DESCRIPTOR,/* POINTER TO DESCRIPTOR*/
  'CONINT',                         /* INTERRUPT NAME */
   FAST$SERVICE,                    /* INTERRUPT TYPE */
  .CONSOLE$INTERRUPT$SERVICE        /* SERVICE ADDRESS*/
  .CONSOLE$EXCEPTION                /* EXCEPTION HANDLER */
   3                                /* INTERRUPT LEVEL */
                                                        );
CALL FOX(CREATE$INTERRUPT,.CONSOLE$STATIC$INTERRUPT$DESCRIPTOR);
```

6.3 LINK GROUP

This group of FOX system procedures links the calling task to a task or to an exchange or to an interrupt. To ease the programmer interaction in a design team, only the name of the task or the exchange or the interrupt service is needed. This opens the way of dynamic linking. The team leader defines all the tasks, exchanges and the interrupts at logical name level, and the interaction between them. Each programmer, then on, can proceed on his own way. Only the name of the other units need be known to him. When everybody brings his own part of the system, FOX will bind all interfaces dynamically. That also guarantees that if any of the team member changes his own part, he need not inform the other team members as long as the predefined interface standarts hold.

The LINK group includes the following procedures

1-) LINK$TASK /* to create a link between tasks */

2-) LINK$EXCHANGE /* to create a link betweeen a task and an exchange */

Each of the above procedures is described in details in the following paragraphs.

6.3.1. "LINK TASK" System Call

This FOX system procedure creates a link between two tasks. This is needed when a task needs to suspend or resume the execution of another task. To create the link, user prepares a task link descriptor where he only needs to give the name of the task to be linked, and then activate FOX system procedure LINK$TASK. FOX searches FOX system link for an entry with the same name and then, returns the descriptor address in the link part of the task link descriptor. Further calls, to suspend and to resume other tasks, will use this information to actually locate the task descriptor, rather than searching the whole system link every time it gets a Suspend or Resume system call. The structure of the task link descriptor is given in FOX data structures.

Gate code of LINK$TASK is defined to be 04.

EXAMPLE

```
DECLARE A$TASK$LINK$DESCRIPTOR TASK$LINK$DESCRIPTOR
        INITIAL(.NIL,'ATASK ', 0,.EXCEPTION$HANDLER);

CALL    FOX(LINK$TASK,.A$TASK$LINK$DESCRIPTOR);
```

6.3.2. "LINK EXCHANGE" System Call

This FOX system call creates a link between a task and an exchange. This is needed when a task wants to exchange messages with an exchange. To create the link between the task and the exchange user prepares an exchange link descriptor and supplies the name of the exchange that he wants to use in the rest of the task. After the preparation of the exchange link descriptor user activates FOX system call LINK$EXCHANGE. FOX will supply the neccessary link between the task and the exchange. This information will be used later by FOX, when user issues a SEND or a WAIT system call. If a user attempts to use either of the system calls SEND or WAIT before the link operation then, the result is unpredictable and such a task is defined to be erroneous. The structure of the exchange link descriptor is given in FOX data structures.

Gate code of LINK$EXCHANGE is defined to be 05.

EXAMPLE

```
DECLARE X$EXCHANGE$LINK$DESCRIPTOR EXCHANGE$LINK$DESCRIPTOR
    INITIAL(.NIL,X$EXCHANGE,0,.X$EXCEPTION);
CALL LINK$EXCHANGE(.X$EXCHANGE$LINK$DESCRIPTOR);
```

6.4 UTILITY GROUP

This group of FOX system calls are used to suspend or to resume any task in the system. Group contains three system procedures, namely ,SUSPEND, RESUME and DISPATCH. Each of these FOX system procedures are described in details in the following sections.

6.4.1. "SUSPEND" System Call

This FOX system call removes the designated task from the system ready queue and puts it into the system suspend queue. If the designated task is already suspended and put into the delay queue, then it will be removed from the system delay queue as well. Any task which is suspended can only be resumed by another task. If there is no other task to resume the suspended task, then there is no chance of reactivating the task.

SUSPEND system call requires the task link descriptor as input parameter. The gate code of SUSPEND is 06.

EXAMPLE

```
DECLARE   A$TASK$LINK$DESCRIPTOR TASK$LINK$DESCRIPTOR ;

 CALL MOVE(6,.('CONSOL'),.A$TASK$LINK$DESCRIPTOR.NAME);
 CALL LINK$TASK(LINK$TASK$CODE,.A$TASK$LINK$DESCRIPTOR);
 CALL SUSPEND(SUSPEND,.TASK$LINK$DESCRIPTOR);
```

6.4.2. "RESUME" System Call

This FOX system call removes the designated task from the system suspend queue and puts it to the ready queue. Only those tasks that are on the system suspend queue can be resumed. If the designated task is not on the system suspend queue then the effect of the call is the same as of Dispatch. If a high priority task is made ready by the RESUME system call then the calling task looses the CPU resource, till it becomes the highest priority ready task again.

RESUME system call requires the task link descriptor address as parameter. The gate code of resume is 07.

EXAMPLE

```
DECLARE   A$TASK$LINK$DESCRIPTOR TASK$LINK$DESCRIPTOR ;
  CALL MOVE(6,.('CONSOL'),.A$TASK$LINK$DESCRIPTOR.NAME);

  CALL LINK$TASK(LINK$TASK$CODE,.A$TASK$LINK$DESCRIPTOR);

  CALL RESUME(RESUME$CODE,.TASK$LINK$DESCRIPTOR);
```

6.4.3. "DISPATCH" Sytem Call

This FOX system call has no effect on the calling task. It just removes the calling task from the ready queue and then re-inserts it to the ready queue again. The effect is that if there are equal priority ready tasks in the ready queue, they can gain the CPU resource.

Dispatch system call requires no parameter, so NIL can be used. The gate code of DISPATCH is defined to be 08.

EXAMPLE

```
CALL    FOX(DISPATCH,NIL);
```

6.5 FOX EXCHANGE OPERATIONS

FOX machine provides several operations that the user can access with programmed calls. Two basic operations are of the most importance. These two operations are described in detail.

1-) SEND, Send a message to an exchange.

2-) WAIT, Wait for a message or time interval.

These two operations provide the capability to pass messages between tasks in a system running under FOX.

6.5.1. Sending a Message to an Exchange

The SEND operation enables a task to post a message at an exchange. When user SENDs a message to an exchange, FOX actually posts the whole body of the message to the buffer area of the exchange. This avoids the overheads of free space management. If user wants to get rid of the overheads required to move the entire message, he may post the address of the actual message. this method is more efficient than passing the address of the actual message. Passing the address of the actual message forces the sender not to modify the message buffer until it is received, that in turn, increases the programming efforts.

In contrast to many executives (iRMX/80) user of FOX is allowed to modify the message buffer after the SEND ooeration is completed. The format of the SEND operation is as follows.

CALL FOX(SEND,EXCHANGE$LINK$DESCRIPTOR$ADDRESS)

Gate code of SEND is defined to be 09.

SEND has one parameter; the address of the exchange link descriptor. Instruction moves the message into its circular first-in-first-out queue if there is available room in the exchange, or suspends the execution of the task and puts the task into the exchange enqueue list. If a task is waiting in the exchange dequeue list then the instruction removes the task from exchange dequeue list and also from the system delay list, if it had requested a time out, and inserts into the system ready list so as to make it eligible to execute on the processor.

The calling task must be linked to the exchange via a LINK$EXCHANGE operation before any SEND operation, or else the result of the SEND operation is not defined.

When a task SENDs a message to an exchange several functions are performed depending on the type of the exchange requested.

6.5.1.a. SEND if exchange is free or return

1-) The designated exchange is checked ,to see if there is room for the incoming message.

2-) If there is no room for a new message return "fail" status

3-) If there is room for the incoming message then move the mesage body to the exchange buffer.

4-) If one or more tasks are waiting at the exchange, the first task is given the message, removed from the exchange's suspend queue and from the system deley queue, if inserted, and thenmade ready.

5-) If a high priority task is , thereby, made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.5.1.b. Wait until SEND operation is completed

1-) The designated exchange is checked to see if there is room for the incoming message.

2-) If there is no room for a new message, the calling task is suspended at this point from futher execution. This condition will be removed when space becomes available. The calling task is inserted into the suspend queue of the exchange.

3-) If there is room for the new message or room becomes available after step 2, the message body is moved to the buffer of the exchange.

4-) If one or more tasks are waiting at the exchange, the first task is given the message, removed from the exchange's suspend queue and from the system deley queue, if inserted there, and then made ready.

5-) If a high priority task is thereby made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.5.1.c. SEND or wait time out

1-) The designated exchange is checked to see if there is room for the incoming message.

2-) If there is no room for a new message, then the calling task is put into the exchange's suspend queue and into the system delay queue, both at the same time. This condition will be removed when space becomes available or when the time limit has expired.

3-) If the time limit has expired then return "fail" status.

4-) If there is room for the new message or room becomes available after, step 2 the message body is moved to the buffer of the exchange.

5-) If one or more tasks are waiting at the exchange, the first task is given the message, removed from the suspend queue and made ready.

6-) If a high priority task is, thereby, made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.5.2. WAITing for a message at an exchange

WAIT operation causes a task to wait for a message to arrive at an exchange. It is also possible to delay execution of the task when no message is anticipated for the exchange. The task simply waits at an exchange where no message is ever sent. When a task waits for a message at an exchange, several operations are performed depending on the type of the exchange operation requested. All of the SEND operation varieries are also applicable to WAIT operation. The only difference is that the task which has requested the WAIT operation is suspended when the exchange is empty, ie. there is no message at the time of WAIT operation.

The WAIT instruction has two parameters; the address of the exchange link descriptor and the WAIT system call gate code. The exchange link descriptor contains the following information.

1-) Type of the exchange operation desired

2-) Address of the buffer area to which the message data will be moved.

3-) Status of the operation returned by FOX.

4-) Address of the exception handler to be activated on an abnormal condition.

5-) Address of the exchange descriptor supplied by FOX at exchange link operation

6-) Maximum time (in system units) for which the task is to await the arrival of a message, if the exchange is empty.

The result of the wait operation is that the message is moved to the indicated memory block, if any could be moved, and a status indicating if the operation is successfuly completed or not. Either "success" or "failed" status is returned. From the programmer's point of view, this instruction simply executes and returns the specified result. Actual execution of the instruction will involve the delaying of task execution if no message is available, by queueing it in a first-come-first-serve manner queue. Any such delay, however, is not visible to the programmer. This approach unifies the communication and timing aspects of the design. It directly provides reliability in the case of lost events due to hardware or software failure. Task can be guaranteed not to be indeterminately delayed due to such failures and thus, attempt recovery from them. It also permits tasks to use the same mechanism to delay themselves for a given time interval by waiting at an exchange at which no message will ever arrive.

To request a wait operation, user sould activate FOX as follows:

```
CALL FOX(WAIT,.A$EXCHANGE$LINK$DESCRIPTOR);
```

Gate code of wait is defined to be 10.

6.5.2.a. WAIT if exchange is full or return

1-) The designated exchange is checked to see if there is any message available.

2-) If there is no messages at the exchange return "fail" status

3-) If there are one or more messages, then move the message body to the user buffer.

4-) If removing a message from the exchange creates free space for a suspended task, the first task is given the free space , removed from the exchange's suspend queue and made ready.

5-) If a high priority task is thereby made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.5.2.b. Wait until WAIT operation is completed

1-) The designated exchange is checked to see if there is any message available.

2-) If there is no message at the exchange the calling task is suspended at this point from further execution. This condition will be removed when a message becomes available for the calling task. The calling task is inserted into the suspend queue of the exchange.

3-) If there is one or more messages at the exchange or a message arrives after step 2, the message body is moved to the user buffer.

4-) If one or more tasks are waiting at the exchange for space become available,, the first task is given the free space, removed from the exchange's suspend queue and made ready.

5-) If a high priority task is ,thereby, made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.5.2.c. WAIT a message or wait time out

1-) The designated exchange is checked to see if there is any message available at the exchange.

2-) If there is no message available at the exchange then the calling task is put into the exchange's suspend queue and to the system delay queue, both at the same time. This condition will be removed when a message arrives the exchange or when the time limit has expired.

3-) If the time limit has expired then return fail status.

4-) If there is one or more messages available or a message arrives after step 2, the message body is moved to the user buffer.

5-) If one or more tasks are waiting at the exchange for space become available, the first task is given space, removed from the suspend queue and made ready.

6-) If a high priority task is , thereby, made ready, the sending task looses the CPU resource until it again becomes the highest priority ready task.

6.6. FOX FLAG OPERATIONS

FOX machine provides two operations on flags that the user can access with programmed calls. Through the use of these two operations any user can easyly write interrupt routines, which are the the hardest to develope and debug. Following sections describes the use of these two FOX system routines.

6.6.1. "EVENT" system call

The EVENT system call requires two parameters, the flag number on which the task wants to wait and the Event system call gate code. A task which has issued an EVENT system call will be suspended till a task issues a SIGNAL system call on the designated flag. If the flag is already SIGNALed then the effect of EVENT system call is the same as DISPATCH system call. A task which is suspended in this manner will be placed in the system suspend queue. User must make sure that no more than one task could issue an EVENT call on the same flag. Such a case is defined as flag-over-run and the exception handler of the calling task is activated, if it has any.

6.6.2. "SIGNAL" system call

The SIGNAL system call has two parameters, the flag number to be SIGNALed and the Signal system call gate code. When a task or interrupt service has issued a signal system call, the event waiting task, if there is any, is removed from the ready queue and made ready. If thereby a high priority task is made ready, then the calling task looses the CPU resource until it again becomes the highest priority ready task. If there is no task waiting on the flag then depending on the flag state either the flag is set to indicate that it is signaled. If it is already set then this is identified as a flag-under-run and the exception handler of the task is activated, if it has any.

VII. MORE ON SEND AND WAIT OPERATIONS

7.1 SEND - WAIT INTERRACTION

To a large extent the power of FOX as an abstract machine is derived from the interaction between SEND and WAIT. The interaction includes three multi-tasking operations.

1-) Communication.

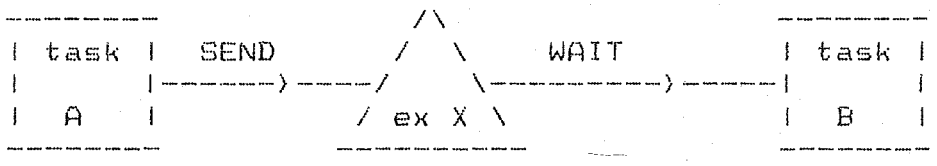
2-) Synchronization

3-) Mutual Exclusion

In describing these operations, a graphic notation , for diagramming tasks, exchanges and their interaction is utilized. Here on, rectangles designate tasks while triangles represent exchanges. Arrows that are directed from tasks to exchanges are SEND operations. WAIT operations are shown by arrows directed from exchanges to the tasks.

7.2 COMMUNICATION

The most common interaction between tasks is that of communication, the transfer of data between one task to another via an exchange, as shown in figure below.

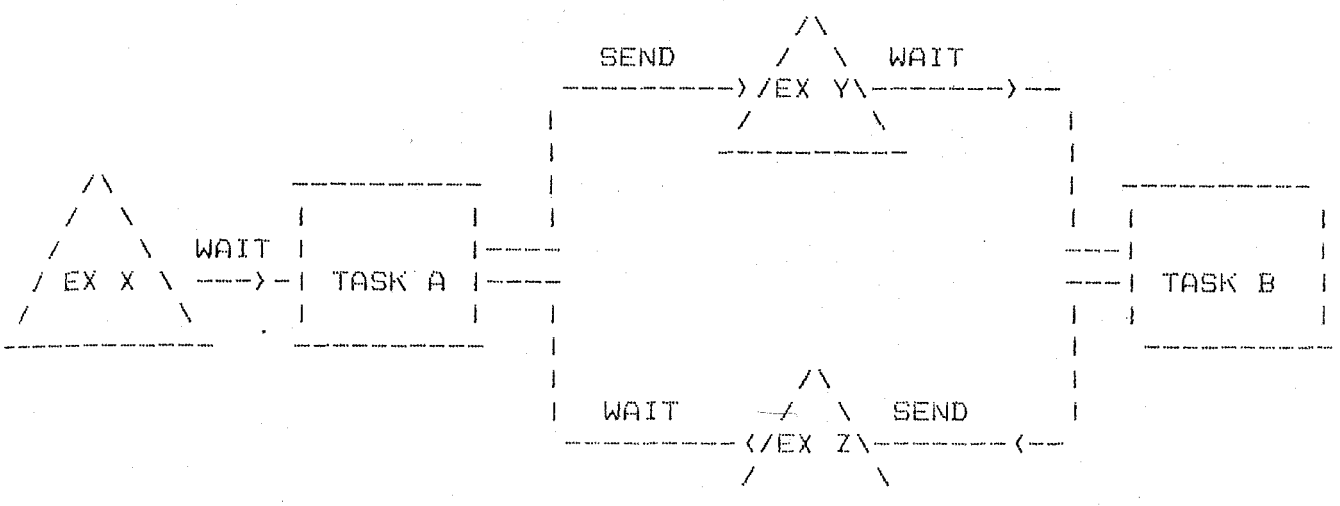


The above figure shows an example of communication between task A and task B. Task A SENDs message to exchange X and task B waits for a message at that exchange. Task A is the message producer and the task B is message consumer.

7.3 SYNCHRONIZATION

At times, there is a requirement to send a synchronization signal from one task to another. The signal can take the form of a message that might have no data bytes at all.

Let us consider the implementation of a task scheduler, used for the purpose of synchronizing another task that performs a particular function at particular time intervals. The relationship between the tasks and the exchanges is shown below.

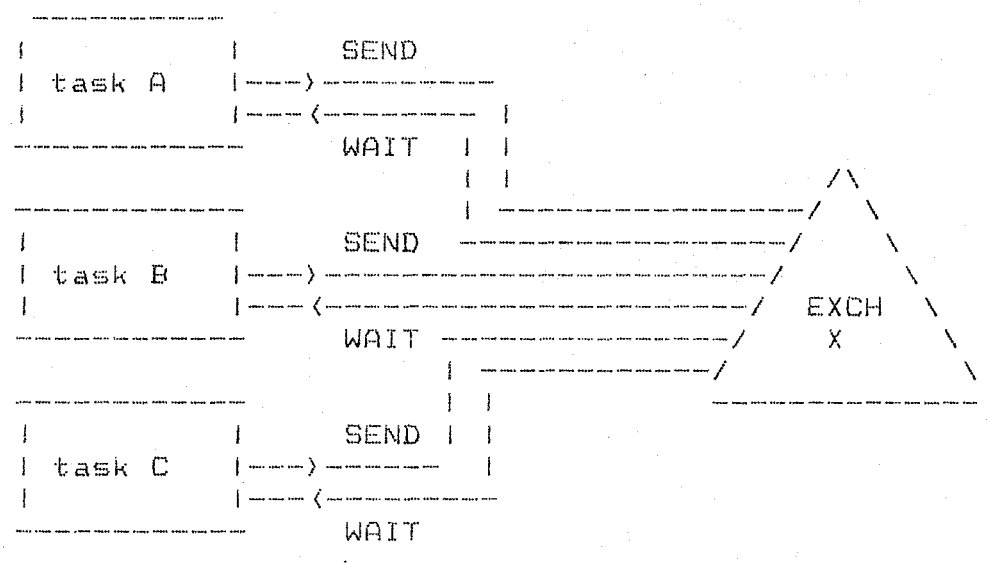


Task A, the scheduler, performs a timed wait on the exchange X. Note that the full wait period will always occur because there is no task that is sending message to exchange X. In this manner a specific timed wait by the task A precedes the passing of a synchronization message from task A to task B via an exchange Y and then returned from task B to task A via an exchange Z as a check back.

If the task B waited on the exchange X directly rather than using task A for scheduling, it would be scheduled n system time units from when it issues WAIT, instead of n system time units from the last time it was awakened. A comparison of these two methods is shown below.

7.4 MUTUAL EXCLUSION

In an environment with multi-tasking, resources, must be shared. Example of shared resources include data structures and peripherals, such as serial communication device to console. Mutual exclusion can be used to ensure that only one task has access to a shared resource at a time. The diagram below shows how one can use an exchange for mutual exclusion.



In this example the exchange X is sent a single message at system initialization. Then as tasks require the resource, they WAIT for a message from the exchange X. When a message is received, the task knows it has sole access to the resource because there is only one message associated with the exchange. After the task finishes with resource, it SENDs the message back to the exchange X. The next task waiting for the resource continues, knowing that it has exclusive access to the resource.

VIII. SYSTEM START UP

8.1 SYSTEM INITIALIZATION

FOX is an embedded or dedicated system executive. This means that it should be able to restart with minimum, or even better, with no operator intervention. FOX, at restart, initializes all of its data base and creates the IDLE task. In addition to this FOX looks for an initial task at absolute location 1000H. If this location contains a JUMP instruction (the machine code of JUMP instruction in this prototype is 0C3h) then FOX creates a second task INITSK with the following attributes. The priority of the INITSK is MAX$PRIORITY-1. The stack area is stack of the dummy task plus 20h. The task start address is 1000h.

This INITSK feature enables the system initiator to write its own initial task and run it whenever a restart occurs. All the remaining tasks, exchanges and interrupts can be created with this INITSK. At the end of the initialization period, this task should be suspended to avoid unneccessary CPU usage. An alternative is to keep it running for a high priority task.

At the initialization all processor interrupts are masked from both the processor via the disable interrupts instruction and also from the priority interrupt decoder(I8259) via the interrupt mask register. Any of the 8 interrupt lines of the interrupt priority decoder can be enabled or disabled with FOX extention calls LEVEL$ON and LEVEL$OFF. These are non-standart FOX system procedures which are supported only when the hardware supports the operation. These two extention calls accept two parameters to indicate the interrupt level to be masked or opened, and the LEVEL$ON or LEVEL$OFF system call gate entry calls. Fox masks a level whenever it receives an interrupt from that level, It is the responsibility of the interrupt service programmer to open the corresponding level at the end of the service routine or else no more interrupts can be signalled from this level.

IX. APPLICATION

9.1 SAMPLE APPLICATION

In this section, an example application of the usage of FOX will be given. Example contains system initialization, task, exchange and interrupt definitions. The example is simple in its task but illustrates most of the capabilities of the FOX.

The subject is to construct a terminal handler of an online debugger. The debugger is assumed to wait for input messages at the exchange 'CONINP' and outputs characters to the exchange 'CONOUT'. The actual task of the debugger is not discussed here.

From the above definitions, we immediately identify two exchanges "CONINP" and the "CONOUT". Through these exchanges the debugger is connected to the outside world. There are two separate tasks, one for the input channel and one for the output channel.

Input channel task waits on the console input interrupt flag for input characters. The type of the flag management is of the exchange type. Whenever the operator enters a character, the console input task will be Signaled and will read the character from the input channel. It will echo the

input character back to the console through the "CONOUT" exchange. When the input character indicates the end of the line, the input task will send the acummulated character to the debugger for processing.

Output channel task waits on the "CONOUT" exchange for the characters to be output Output channel task receives the characters to be output and then issues an Event system call on the console output interrupt flag. When the output channel becomes available, FOX Signals the output interrupt flag and the output channel task outputs the next character. This will be repeated until all of the characters are output to the channel.

Refering to the appendix C ,the source list of the sample application, one can see that the module can be partitioned into three sections. The definition , initialization and the task bodies. In the following sections each of these parts will be discussed in detail.

9.2 DEFINITION

Lines one through 35 are the definition part of the module. This part defines two external entries, the FOX entry and the DEBUG entry. The first one, FOX, defines the executive as a procedure with two parameters; the gate code and the descriptor address on which the defined operation will be applied. The second one is the address of the debugger program. It will be created as a task later in the initialization section.

Remaining part is the object definition part of the module. All of the objects are defined here. Line 23 defines the Static Task Descriptor of the debugger task. This descriptor is initiated with the following values.

a-) The Task Descriptor of the debugger task is defined to be the DEBUG$TASK$DESCRIPTOR.

b-) Name of the debugger task is "DEBUG ".

c-) Priority of the debugger task is ten.

d-) Actual body of the task starts at the address DEBUG, which is the start address of the debugger procedure.

e-) Debugger task has no exception handler.

At line 23 the static task descriptor of the console input task is defined and initialized with the followig values.

a-) The task descriptor of the console input task is defined to be CI$TASK$DESCRIPTOR.

b-) Name of the task is defined to be "CITASK".

c-) Priority of the console input task is set to be 11.

d-) Console input task starts at the address CONSOLE$INPUT$TASK.

e-) Console input task has no exception handler.

At line 25, the static task descriptor of the console output task is defined and initialized with the following values.

a-) The task descriptor of the console input task is defined to be CO$TASK$DESCRIPTOR.

b-) Name of the task is defined to be "COTASK".

c-) Priority of the console input task is set to be nine.

d-) Console input task starts at the address CONSOLE$OUTPUT$TASK.

e-) Console input task has no exception handler.

At line 26, some constants are defined. These are the EOM , the end of message character, the terminal data port address, the console input event id and the console interrupt id .

Following the constant definitions, there are the exchange descriptor definitions of the console input exchange and the console output exchange. Name of the console input exchange is defined to be CI$EXCHANGE$DESCRIPTOR and the name of the console output exchange is set to be CO$EXCHANGE$DESCRIPTOR.

At line 29, the interrupt descriptors for the console input interrupt and the console output interrupt are defined. The name of the console input interrupt descriptor is defined to be CI$INTERRUPT$DESCRIPTOR and the console output interrupt descriptor is defined to be CO$INTERRUPT$DESCRIPTOR.

At line 30, the static interrupt descriptor of the console input interrupt is defined and initialized with the following values.

a-) Interrupt descriptor is the CI$INTERRUPT$DESCRIPTOR

b-) Name of the interrupt is CI$INT

c-) Type of the exchange operation is of EXCHANGE type.

d-) No fast service address.

e-) No exception handler.

f-) Interrupt level is one.

At line 31 the static interrupt descriptor of the console output interrupt is defined and initialized with the following values.

a-) Interrupt descriptor is the CO$INTERRUPT$DESCRIPTOR

b-) Name of the interrupt is CO$INT

c-) Type of the exchange operation is of EXCHANGE type.

d-) No fast service address.

e-) No exception handler.

f-) Interrupt level is two.

At line 32, the static exchange descriptors of the console output exchange is defined and initialized with the following values.

a-) Exchange descriptor of the console output exchange is defined to be CO$EXCHANGE$DESCRIPTOR.

b-) The name of the exchange is defined to be "CONOUT".

c-) There can be at most one message at the exchange.

d-) length of the exchange is defined to be 80 characters.

At line 33, the static exchange descriptors of the console input exchange is defined and initialized with the following values.

a-) Exchange descriptor of the console input exchange is defined to be Ci$EXCHANGE$DESCRIPTOR.

b-) The name of the exchange is defined to be "CONINP".

c-) There can be at most one message at the exchange.

d-) length of the exchange is defined to be 80 characters.

At line 34 , a task link descriptor is defined to be used at the initialization for task creation.

9.3 INITIALIZATION

This part of the module starts up the system. It creates the neccessary exchanges, namely the console input exchange and the console output exchange. Then it creates the system tasks, namely the debugger task whose actual coding is not discussed here, the console input task and the console output task. At this point the reader should notice that creating a task only inserts it into the FOX system supend

queue. Becouse of that they can not be dispatched until they are inserted to the system ready queue. That will be done later by RESUME system calls.

At lines 42 and 43 the interrupts are introduced to the FOX system, namely the console input interrupt and the console output interrupt.

At this point the system is ready for start up by repetetive calls to the RESUME system procedure, all of the three tasks are resumed. But at this point, again, they can not get the CPU resource since their priorities are lower then the priotity of the initial task which is by definition MAX$PRIORITY-1.

At the end of the initialization initial task suspends itself from further execution by a call to the SUSPEND system procedure. At this point, since it is removed from the system ready queue, the remining higest priority task, which is the console input task, becomes the running task.

9.4 TASK BODIES

This part of the module contains the actual task bodies. The console input and console output tasks are defined here. The task definition of the debugger is not included here since its functioning is out of the scope of this thesis. Console input task has two parts in it, the initialization and the task loop. The initialization part links the console input task to the console output exchange and the console input exchange. It also defines the buffer addresses for the exchanges as the CI$BUF and the CHR$BUF respectively. Then the console input task goes into an infinit loop where it waits at the console input interrupt, reads the incomming caharacters, sends them to the console output exchange and when the end of message byte received, send the cummulated characters to the debugger task. Indeed that is the complete job of the console input task.

Console output task has also two parts, initialization and the task loop. It creates a link to the console output exchange and then goes into an infinite loop where it waits at the console output exchange for a message and then sends them to the console output channel as console outout interrupts arrived.

X. CONCLUSION

The aim of this study was to desig and implement a priority driven multitasking executive. The resulting system, apart from a few minor differences is quite similar to the original research proposal (see Appendix A).

In comparing the resulting system with the original proposal of FOX we see the absence of a few system calls. These are the EXIT, LINK$INTERRUPT and the complete Delete group of calls. The reason for their ommission was that they conflicted the design philosophy of FOX which is an embedded system executive. During the implementation three new system calls were thought of in order to make interrupt handler writing easier. Despite all, the spirit of the proposal remained to be the same. At the beginning it was estimated that a language mixture of five per cent assembly and 95 per cent high level language would be used. At the end it was written in PL/M. This brought the possibility of portability, because today cross compilers of PL/M is available for various microprocessors. The list contains the 8086 , 8088 and 8051 from Intel, with 100 per cent source compatibility, and Z80 , 6800, 6809, and 6502 from the PLMX corp. which are not 100 per cent software compatible, but the only restriction is that it does not support names

longer than 6 characters for public entries, and we do not have so many of them.

In this study, I tried to approach the executive as an abstract machine and tried to show that an executive can be implemented such that moving the executive from one family of microcomputers to another does not effect the software more than changing the executive to adopt to the new hardware.

Admittedly , there may be some bugs left in somewhere in the software, since testing an executive under every possible condition is hardly possible. But I am open to complaints and ready to correct any such bugs.

APPENDIX A FOX PROPOSAL

A PROPOSAL FOR

A FAST OBJECT-ORIENTED EXECUTIVE

[FOX]

SUBMITTED TO : DR. TUNC BALMAN
SUBMITTED BY : SEDAT YILMAZER
SUBJECT : MASTER THESIS
DATE : 11/20/1982

TRODUCTION

I propose to design and implement a fast object-oriented executive. It will be a collection of routines and tasks which support event driven multi-tasking on a microprocessor. I propose to design and implement a Fast Object-oriented eXecutive [FOX] . First version of FOX will run on a INTEL SBC 80/20-4 based system. In order to make it as portable as possible it will be written in a HLL . Due to the availability and highly structred nature, the INTEL PL/M is proposed. A high portion of the FOX will be written in PL/M except possibly a very small portion has to be written in the assembly language of the target system. The overall system will satisfy the %95 PL/M %05 assembly ratio.

FOX will be a Real-Time Event-driven Object-oriented Multi-tasking executive. It will be Real-Time since the main objective of FOX is to be a contol application executive , where Real-time scheduling is a must . It will be Event-driven to decrease unnesseccary polling of external peripherals and provide a structured method of interfacing with the hard-ware. It will be object oriented to ease the user interface and possible future extentions , such as a File Manager, and finaly it will be multi-tasking to increase the throughput for each peny paid for the hardware.

First version of FOX will run on a system called MSCP in NETAS. It is a sub-unit of a national telephone center under the development of R&D department. It has all the neccessary hard-ware needed for an executive , and so for FOX. It consists of a processor board (SBC 80/20-4) with 4Kb of RAM , 8Kb of EPROM , an RS 232 communication interface , a programmable interrupt decoder , 3 programable timers and some parallel I/O and multi-bus interface logic. It also encapsulate an I/O , RAM EPROM extention board (SBC 116). So the complete system has 2 SBC boards 20Kb of RAM and upto 40Kb of EPROM , almost more than enough for a simple control application.

DESIGN METHODE

I will try to design the FOX as flexible as possible so it will be easily customized for any control application. I will take the INTEL iRMX-80 , INTEL iRMX-86 , NETAS MTEX ,NETAS REMTEX and the DIGITAL RESERCH MPM/II as bases to FOX, (I don't want to invent the wheel again !).

In the design phase , I would like to follow the rules listted below

a -) There will be two design reviews one after the acceptance of this proposal , and one after one month after the first design review. In first meeting the first level definition of the FOX will be made. Possible objects will be discussed , scheduling methodes will be defined. In the second design review, the dificulties and possible contradictions in the first definitions will be discoused and solved.

b -) The design methode will be Top-down program development and Bottom-up object definition. First the objects that the executive will heavily depend on , such as process descriptors and queue structures ,will be defined and ,then on, the program development and coding will start.

c` -) If not strickly neccessary , machine depended features will not be used neither in HW nor in SW , including the machine language of the target machine.

DEFINITION OF FOX

FOX will be a nucleus to be costumized for any specific multi-tasking system. It will have a minimal and sufficient user interface routines to handle the tasks and the communication between them. The unit of executable code collections under the FOX is a TASK. There can be any number of tasks competing for the CPU and for the peripherals at any instance of time. FOX will supply sufficient mechanizms to synchronize the activities of all of the peripherals. Messages between the tasks will be handled in the EXCHANGEs. Both the tasks and the exchanges can be created and deleted with simple system calls. FOX will also support a unified interrupt structure. User can either ask the FOX to suspend it until a specific interrupt arrives or define his own interrupt service routine for faster interrupt servicing.

FOX will use linked queues to schedule the tasks and manupulate the exchanges. For task scheduling , there will be three system queues

a -) System ready task queue : For the tasks that can immediatly gasp the CPU and execute. This queue will be held in high priority first and round-roubin between the equal priority tasks.

b -) System dormant task queue : For the tasks which are neither waiting for peripherals nor for the CPU nor for the exchange operations and nor for a time-out condition (they are

merely in the system but not working at all for the time being) they can only be activated via a create_task system call. The queue has no implied structure , but first in last out one seems to be more convenient.

c -) System time-wait queue : The tasks which are intentionally waiting for a time-out condition or waiting for the exchange operation, when the user defines a time-out in the exchange , will be held in this queue. Since there can be many tasks in this queue the structure of the queue should minimize the job of the delay manager so that it does not have to scane all time-wait queue at each system clock tick.

In addition , tasks can be queued at the exchange descriptors for send or receive operations. If an exchange has no message when a task requests one , or , if the exchange has no room for a new message when a task sends one , the task will be queued at the exchange descriptor. There can be any number of tasks waiting at an exchange either for receive operation or for a send operation (mixture of send waiting and receive waiting task at any exchange queue is , of course , impossible).

Tasks can be queued at the exchage queue and at the system time wait queue at the same time , if the user had asked for a time out at the exchange. In such a case, the task will be removed from both of the queues when the time out condition is met or when the exchange operation is completed.

The following sections desicribe the proposed system routines for the above mentioned utilities.

a -) CREATE_TASK

CREATE_TASK(TASK_DESCRIPTOR) ;

This procedure will insert the designated tasks process descriptor into the system ready task queue. Then on the task competes for the CPU and the peripherals as other tasks do.

b -) DELETE_TASK

DELETE_TASK(TASK_DESCRIPTOR) ;

This procedure removes the designated task from all of the following queues and puts into the system dormant task queue.

i -) System ready task queue

ii -) System time wait queue

iii -) Exchange wait queues

c -) EXIT

EXIT ;

This procedure removes the calling task from the system ready task queue.

d -) LINK_TASK

LINK_TASK(TASK_LINK_DESCRIPTOR);

This procedure links the designated task to the calling task.

e -) CREATE_EXCHANGE

CREATE_EXCHANGE(EXCHANGE_DESCRIPTOR) ;

This procedure insets the designated exchange descriptor into the system exchage link.

f -) DELETE_EXCHANGE

DELETE_EXCHANGE(EXCHANGE_DESCRIBTION) ;

This procedure removes the designated exchange from the system exchange link

g -) LINK_EXCHANGE

LINK_EXCHANGE(EXCHANGE_LINK_DESCRIPTOR) ;

This procedure links the designated exchange to the calling task for further exchange operations

h -) SEND

SEND(EXCHANGE_LINK_DESCRIPTOR) ;

This procedure sends the designated message to the designated exchange. Task can ask for a time out facility under abnormal conditions and or define an exception handler. Also the FOX defines a dummy exchange for task wait operations which is almost a must in an event driven system.

i -) REQUEST

REQUEST(EXCHANGE_LINK_DESCRIPTOR);

This procedure asks for a message at the designated exchange. User can define time out and or exception handler for abnormal conditions , as in the SEND case.

j -) DEFINE_INTERRUPT

DEFINE_INTERRUPT(INTERRUPT_DESCRIPTOR)

This procedure defines an interrupt at specified level and type.

k -) DELETE_INTERRUPT

DELETE_INTERRUPT(INTERRUPT_DESCRIPTOR) ;

This procedure deletes the interrupt defined before .

l -) LINK_INTERRUPT

LINK_INTERRUPT(INTERRUPT_LINK_DESCRIPTOR) ;

This procedure links the task to designated interrupt descriptor for further references.

m -) WAIT_INTERRUPT

WAIT_INTERRUPT(INTERRUPT_LINK_DESCRIPTOR) ;

This procedure puts the task into time wait queue until the designated interrupt (or event) occures.

n -) CREATE_INTERRUPT

CREATE_INTERRUPT(INTERRUPT_LINK_DESCRIPTOR)

This procedure sets the designated interrupt (or event).

APPENDIX B SOURCE LISTINGS

```
SIS-II PL/M-80 V3.0 COMPILATION OF MODULE INITIALIZATION
BJECT MODULE PLACED IN START.OBJ
OMPILER INVOKED BY:  PLM80 START PRINT(:LP:)


        $DEBUG WORKFILES(:F0:.:F0:)
        /*************************************************/
        /**       FOX INITIALIZATION MODULE             **/
        /**                                             **/
        /**   THIS MODULE INITIATES ALL POINTERS AND    **/
        /**      CREATES THE INITIAL AND IDLE TASKS     **/
        /**                                             **/
        /**   MODULE NAME : INITIALIZATION              **/
        /**   FILE   NAME : START                       **/
        /**   AUTHOR      : SEDAT YILMAZER              **/
        /**   DATE        : 15.12.1983                  **/
        /**                                             **/
        /*************************************************/
 1      INITIALIZATION:
        DO:

        $NOLIST

16  1   DECLARE JUMP            LITERALLY       '0C3H':

17  1   DECLARE INT$VECTOR  BYTE    EXTERNAL:

18  1   DECLARE (CREATE$TASK.CREATE$EXCHANGE.CREATE$INTERRUPT.
                 LINK$TASK.LINK$EXCHANGE.
                 KILL.RESUME.DISPATCH.
                 SEND$EXCHANGE.WAIT$EXCHANGE.
                 EVENT.SIGNAL.
                 LEVEL$ON.LEVEL$OFF.END$INT.
                 ERROR                      )   ADDRESS EXTERNAL:

19  1   DECLARE INI$STACK(20)       BYTE:
20  1   DECLARE INI$TASK$DESCRIPTOR     TASK$DESCRIPTOR:
21  1   DECLARE IDLE$TASK$DESCRIPTOR    TASK$DESCRIPTOR:

22  1   DECLARE INI$S$TASK$DESCRIPTOR   STATIC$TASK$DESCRIPTOR
         DATA(.INI$TASK$DESCRIPTOR.
                 'INITSK'.
                 250.
                 1000H.
                 .INI$STACK+20.
                 NIL):

        /********** FOX GATE CODE DEFINITIONS ***************/
                 ~
        /*DECLARE   CREATE$TASK        LITERALLY   '01'.
                 CREATE$EXCHANGE    LITERALLY   '02'.
                 CREATE$INTERRUPT   LITERALLY   '03'.
                 LINK$TASK          LITERALLY   '04'.
                 LINK$EXCHANGE      LITERALLY   '05'.
                 SUSPEND            LITERALLY   '06'.
                 RESUME             LITERALLY   '07'.
```

```
            DISPATCH            LITERALLY   '08'.
            SEND                LITERALLY   '09'.
            WAIT                LITERALLY   '10'.
            EVENT               LITERALLY   '11'.
            SIGNAL              LITERALLY   '12'.
            LEVEL$ON            LITERALLY   '20'.
            LEVEL$OFF           LITERALLY   '21'.
            END$INT             LITERALLY   '22':
        */
23  1   TIMER$INTERRUPT: PROCEDURE EXTERNAL:
24  2   END:

        /*=======================================================*/
```

```
            $EJECT
            /*******************************************************/
            /**    INITIALIZE THE QUEUE POINTERS AND CURRENT      **/
            /**     TASK POINTER                                  **/
            /**                                                   **/
            /**  PROCEDURE : SW$INITIALIZATION                    **/
            /**  FUNCTION  : INITIALIZE THE FOX POINTERS          **/
            /**  CALL      : CALL SW$INITIALIZATION               **/
            /**  HISTORY   : CREATED AT 12.25.1984                **/
            /**                                                   **/
            /*******************************************************/
25   1      SW$INITIALIZATION: PROCEDURE :

26   2          RQ$HEAD=.IDLE$TASK$DESCRIPTOR:
27   2          SQ$HEAD=NIL:
28   2          DQ$HEAD=NIL:
29   2          FL$HEAD=NIL:
30   2          CP=RQ$HEAD:
31   2      END:
```

```
        $EJECT
        /*****************************************************/
        /**    INITIALIZE THE FOX HARDWARE                  **/
        /**                                                 **/
        /**  PROCEDURE : HW$INITIALIZATION                  **/
        /**  FUNCTION  : INITIALIZE THE FOX HARDWARE        **/
        /**  CALL      : CALL HW$INITIALIZATION             **/
        /**  HISTORY   : CREATED AT 12.25.1984              **/
        /**                                                 **/
        /*****************************************************/
32  1   HW$INITIALIZATION: PROCEDURE :

33  2   DECLARE TIMER$CONTROL$PORT      LITERALLY   '0DFH',
              TIMER$COUNT$PORT        LITERALLY   '0DDH',
              PIC$ADDRESS$PORT        LITERALLY   '0D8H',
              COUNT                   LITERALLY   '25000',
              INITIALIZE$TIMER        LITERALLY   '036H';

34  2       OUTPUT(TIMER$CONTROL$PORT)=INITIALIZE$TIMER;
35  2       OUTPUT(TIMER$COUNT$PORT)=LOW(COUNT);
36  2       OUTPUT(TIMER$COUNT$PORT)=HIGH(COUNT);
37  2       OUTPUT(PIC$ADDRESS$PORT)=LOW(.INT$VECTOR)+16H;
38  2       OUTPUT(PIC$ADDRESS$PORT)=HIGH(.INT$VECTOR);

39  2   END;
```

```
$EJECT
/*****************************************************/
/**       FOX ENTRY PROCEDURE                       **/
/**                                                 **/
/**  PROCEDURE : FOX                                **/
/**  FUNCTION  : DISPATCH FOX SYSTEM CALL           **/
/**  CALL      : CALL FOX(FUNCTION$ID.PARAMETER)    **/
/**  HISTORY   : CREATED AT 25.12.1983              **/
/**                                                 **/
/*****************************************************/

40  1   FOX: PROCEDURE(GATE$VALUE.PARAMETER) PUBLIC REENTRANT:
41  2   DECLARE GATE$VALUE  BYTE.
            PARAMETER   ADDRESS:
42  2       DECLARE CALL$ADR    ADDRESS:
43  2       DECLARE FUNCTION(*) POINTER DATA(.ERROR.
                                            .CREATE$TASK.
                                            .CREATE$EXCHANGE.
                                            .CREATE$INTERRUPT.
                                            .LINK$TASK.
                                            .LINK$EXCHANGE.
                                            .KILL.
                                            .RESUME.
                                            .DISPATCH.
                                            .SEND$EXCHANGE.
                                            .WAIT$EXCHANGE.
                                            .EVENT.
                                            .SIGNAL.
                                            .ERROR.
                                            .ERROR.
                                            .ERROR.
                                            .ERROR.
                                            .ERROR.
                                            .ERROR.
                                            .ERROR.
                                            .LEVEL$ON.
                                            .LEVEL$OFF.
                                            .END$INT
                                                    ):
44  2       DECLARE MAX$GATE$VALUE  LITERALLY   '22':

45  2       DISABLE:
46  2       IF  GATE$VALUE (= MAX$GATE$VALUE THEN
47  2       DO:
48  3           CALL$ADR=FUNCTION(GATE$VALUE):
49  3           CALL    CALL$ADR(PARAMETER):
50  3       END:
51  2       ENABLE:
52  2   END:

    /*****************************************************/
    /**       INTERRUPT SERVICE ROUTINES                **/
    /**    CURRENTLY 8 INTERRUPT LEVELS ARE HANDLED     **/
    /**                                                 **/
    /*****************************************************/

53  1   EVENT0: PROCEDURE   INTERRUPT 0:
```

```
54  2          CALL    FOX(12.0):
55  2      END:

56  1      EVENT1: PROCEDURE  INTERRUPT 1:
57  2          CALL    FOX(12.1):
58  2      END:

59  1      EVENT2: PROCEDURE  INTERRUPT 2:
60  2          CALL    FOX(12.2):
61  2      END:

62  1      EVENT3: PROCEDURE  INTERRUPT 3:
63  2          CALL    FOX(12.3):
64  2      END:

65  1      EVENT4: PROCEDURE  INTERRUPT 4:
66  2          CALL    FOX(12.4):
67  2      END:

68  1      EVENT5: PROCEDURE  INTERRUPT 5:
69  2          CALL    FOX(12.5):
70  2      END:

71  1      EVENT6: PROCEDURE  INTERRUPT 6:
72  2          CALL    FOX(12.6):
73  2      END:

74  1      EVENT7: PROCEDURE INTERRUPT 7:
75  2          CALL    TIMER$INTERRUPT:
76  2      END:
```

```
            $EJECT
            /*******************************************************/
            /**           INITIALIZATION PROCEDURE AND            **/
            /**                 IDLE TASK                         **
            /**                                                   **/
            /**  PROCEDURE : START                                **/
            /**  FUNCTION  : INITIATE SOFTWARE AND HARDWARE       **/
            /**  CALL      : N/A                                  **/
            /**  HISTORY   : CREATED 12.25.1984                   **/
            /**                                                   **/
            /*******************************************************/

 77   1     START: PROCEDURE    PUBLIC:

 78   2     DECLARE I$TASK$LINK$DESCRIPTOR  TASK$LINK$DESCRIPTOR:
 79   2     DECLARE START$JUMP  BYTE        AT(1003):

 80   2        DISABLE:
 81   2        CALL    SW$INITIALIZATION:  /* IDLE TASK IS INITIATED */
 82   2        CALL    HW$INITIALIZATION:
 83   2        IF      START$JUMP=JUMP THEN
 84   2        DO:
 85   3           CALL    FOX(CREATE$TASK..INI$S$TASK$DESCRIPTOR):
 86   3        END:
 87   2        ENABLE:
 88   2        CALL    MOVE(.('INITSK')..I$TASK$LINK$DESCRIPTOR.NAME,6):
 89   2        CALL    FOX(LINK$TASK..I$TASK$LINK$DESCRIPTOR):
 90   2        CALL    FOX(RESUME..I$TASK$LINK$DESCRIPTOR):
 91   2        DO WHILE 1:
                 /* IDLE TASK */
 92   3        END:
 93   2     END:


            /*************** END OF MODULE ***********************/

 94   1     END:


MODULE INFORMATION:

    CODE AREA SIZE     = 0183H    387D
    VARIABLE AREA SIZE = 006BH    107D
    MAXIMUM STACK SIZE = 000FH     15D
    501 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.0 COMPILATION OF MODULE USERINTERFACE
NO OBJECT MODULE REQUESTED
COMPILER INVOKED BY:  PLM80 USER PRINT(:LP:) NOOBJECT


            $WORKFILES(:F0:,:F0:) DEBUG
            /*****************************************************/
            /**          MASTER THESIS CODE SECTION             **
            /**                                                 **
            /**   THIS MODULE DEFINES ALL USER INTERFACE OF FOX  **
            /**                                                 **
            /**  MODULE NAME : USER$INTERFACE                   **
            /**  FILE   NAME : USER                             **
            /**  AUTHOR      : SEDAT YILMAZER                   **
            /**  DATE        : 5.29.1983                        **
            /**                                                 **
            /*****************************************************/

  1         USER$INTERFACE: DO:
            $NOLIST
```

```
$EJECT
/*************************************************************/
/*          CREATE A TASK DESCRIPTOR IN SYSTEM LINK          */
/*                                                           */
/*   PROCEDURE : CREATE$TASK                                 */
/*   FUNCTION  : INITATE TASK DESCRIPTOR AND INSERT SYSTEM   */
/*   CALL      : CREATE$TASK(S$T$D):                         */
/*   HISTORY   : CREATED AT 06.11.1983                       */
/*************************************************************/

CREATE$TASK: PROCEDURE(S$T$D) PUBLIC :

DECLARE S$T$D   POINTER.
    S$TASK  BASED S$T$D STATIC$TASK$DESCRIPTOR.
    T$D POINTER.
    TASK    BASED   T$D TASK$DESCRIPTOR:

    T$D=S$TASK.L:
    CALL    MOVE(6..S$TASK.NAME(0)..TASK.NAME(0)):
    TASK.P=S$TASK.P:
    TASK.SP=S$TASK.SP:
    TASK.EH=S$TASK.EH:
    TASK.L.TASK.D=NIL:
    TASK.ST=IDLE:
    TASK.C=0:
    RAM(TASK.SP-9)=LOW(.S$TASK.PC):
    RAM(TASK.SP-10)=HIGH(.S$TASK.PC):
    CALL    INSERT$SYSTEM$LINK(T$D):
    CALL    INSERT$QUEUE(T$D.SD$HEAD):
END:
```

```
        $EJECT
        /*************************************************************/
        /*          CREATE EXCHANGE IN SYTEM LINK                    */
        /*                                                           */
        /*   PROCEDURE : CREATE$EXCHANGE                             */
        /*   FUNCTION  : INTRODUCE AN EXCHANGE INTO SYSTEM           */
        /*   CALL      : CREATE$EXCHANGE(S$E$D):                     */
        /*   HISTORY   : CREATED AT 06.11.1983                       */
        /*************************************************************/

72  1   CREATE$EXCHANGE: PROCEDURE(S$E$D) PUBLIC :

73  2   DECLARE S$E$D   POINTER.
        S$EXCHANGE  BASED   S$E$D   STATIC$EXCHANGE$DESCRIPTOR.
        E$D POINTER.
        EXCHANGE    BASED   E$D EXCHANGE$DESCRIPTOR:

74  2       E$D=S$EXCHANGE.L:
75  2       CALL MOVE(6..S$EXCHANGE.NAME(0)..EXCHANGE.NAME(0)):
76  2       EXCHANGE.LNG=S$EXCHANGE.LNG:
77  2       EXCHANGE.SZ=S$EXCHANGE.SZ:
78  2       EXCHANGE.BSZ=EXCHANGE.SZ*EXCHANGE.LNG:
79  2       EXCHANGE.EH=NIL:
80  2       EXCHANGE.MF.EXCHANGE.MB=0:
81  2       CALL    INSERT$SYSTEM$LINK(E$D):
82  2   END:
```

```
$EJECT
/*****************************************************/
/**       CREATE INTERRUPT SERVICE                  **/
/** PROCEDURE : CREATE$INTERRUPT                    **/
/** FUNCTION  : INTRODUCE AN INTERRUPT SERVICE      **
/** INPUTS    : INTERRUPT DESCRIPTOR POINTER        **/
/** OUTPUTS   : NONE                                **/
/** CALL      : USER TASK                           **/
/** HISTORY   : CREATED AT 11.12.1983               **/
/*****************************************************/

83  1     CREATE$INTERRUPT: PROCEDURE(S$I$D$P) PUBLIC:
84  2     DECLARE S$I$D$P PTR.
            S$I$D   BASED   S$I$D$P STATIC$INTERRUPT$DESCRIPTOR.
            I$D$P   PTR.
            I$D     BASED   I$D$P   INTERRUPT$DESCRIPTOR:

85  2           I$D$P=S$I$D.L:
86  2           CALL MOVE(6..S$I$D.NAME(0).I$D.NAME(0)):
87  2           I$D.T=S$I$D.T:
88  2           I$D.F=EMPTY:
89  2           I$D.LVL=S$I$D.LVL:
90  2           I$D.EH=S$I$D.EH:
91  2           INTERRUPT$VECTOR(I$D.LVL)=I$D$P:
92  2           CALL INSERT$SYSTEM$LINK(I$D$P):
93  2     END:
```

```
$EJECT
/*************************************************************/
/*         LINK TASK OR EXCHANGE TO THE CALLING TASK         */
/*                                                           */
/*   PROCEDURE : LINK$SYSTEM                                 */
/*   FUNCTION  : LINK TASK OR EXCHANGE TO CALLER             */
/*   CALL      : LINK$SYSTEM(L$P):                           */
/*   HISTORY   : CREATED AT 06.11.1983                       */
/*   NOTICE    : IF THE DESIGNATED ENTRY IS NOT IN THE       */
/*               SYSTEM THEN THE EXCEPTION HANDLER OF THE    */
/*               TASK WILL BE EXECUTED                       */
/*************************************************************/

 94  1      LINK$SYSTEM: PROCEDURE(L$P) PUBLIC :

 95  2      DECLARE         L$P     POINTER .           /* LINK POINTER */
            LINK        BASED   L$P     DESCRIPTOR. /* LINK DESCRIPTOR */
            S$P         POINTER.                    /* SYSTEM LINK POINTER */
            S$L         BASED   S$P     DESCRIPTOR. /* SYSTEM LINK DESCRIPTOR */
            TASK        BASED   C$P     TASK$DESCRIPTOR:
 96  2          S$P=FL$HEAD:
 97  2          LINK.L=NIL:                         /* INITIALIZE FOR NO SUCCESS */
 98  2          DO WHILE S$P <> NIL:
 99  3              IF COMPARE(.S$L.NAME(0)..LINK.NAME(0).6) THEN
100  3              DO :
101  4                  LINK.L=S$P:
102  4                  S$P=NIL:
103  4              END:
104  3              ELSE    S$P=S$L.SL:
105  3          END:
106  2          IF (LINK.L=NIL) AND (TASK.EH<>NIL) THEN CALL TASK.EH:
108  2      END:
```

```
        $EJECT
        /*******************************************************/
        /*                                                     */
        /*                                                     */
        /*   PROCEDURE : KILL                                  */
        /*   FUNCTION  : KILL [SUSPEND] DESIGNATED TASK        */
        /*   CALL      : CALL KILL(T$L$D):                     */
        /*   HISTORY   : CREATED AT 06.05.1983                 */
        /*******************************************************/
109  1  KILL: PROCEDURE(T$L$D) PUBLIC :
110  2  DECLARE T$L$D POINTER .
            TASK$LINK BASED T$L$D TASK$LINK$DESCRIPTOR:

111  2      CALL    REMOVE$READY$QUEUE(TASK$LINK.L):
112  2      CALL    INSERT$QUEUE(.SQ$HEAD.TASK$LINK.L):
113  2      CALL    DISPATCH:
114  2  END:
```

```
                $EJECT
                /*************************************************************/
                /*                                                           */
                /*                                                           */
                /*   PROCEDURE : RESUME                                      */
                /*   FUNCTION  : RESUME A SUSPENDED TASK                     */
                /*   CALL      : CALL RESUME(T$L$D)                          */
                /*   HISTORY   : CREATED AT 06.05.1983                       */
                /*                                                           */
                /*************************************************************/
115   1         RESUME: PROCEDURE(T$L$D) PUBLIC :

116   2         DECLARE T$L$D POINTER.
                    TASK$LINK BASED T$L$D TASK$LINK$DESCRIPTOR:

117   2             CALL    REMOVE$QUEUE(.SQ$HEAD.TASK$LINK.L):
118   2             CALL    INSERT$READY$QUEUE(TASK$LINK.L):
119   2             CALL    DISPATCH:
120   2         END:
```

```
$EJECT
/*****************************************************/
/**     INTERRUPT FLAG WAIT                         **/
/**                                                 **/
/** PROCEDURE : EVENT                               **/
/** FUNCTION  : WAIT FOR AN INTERRUPT OR EVENT      **/
/** INPUTS    : INTERRUPT LINK DESCRIPTOR           **/
/** OUTPUTS   : NONE                                **/
/** CALL      : USER TASKS                          **/
/** HISTORY   : CREATED AT 11.12.1983               **/
/** NOTICE    :                                     **/
/*****************************************************/

1  1    EVENT: PROCEDURE(LEVEL) PUBLIC REENTRANT:

2  2    DECLARE LEVEL   BYTE.       /* POINTER TO INTERRUPT LINK DESCRIPTOR */
           I$D$P   PTR.
           I$D     BASED   I$D$P   INTERRUPT$DESCRIPTOR:

3  2        I$D$P=INTERRUPT$VECTOR(LEVEL):
4  2        IF I$D.F = EMPTY THEN
5  2        DO:
6  3            I$D.F=FULL:
7  3            I$D.TASK=C$P:
8  3            DISABLE:
9  3            CALL REMOVE$READY$QUEUE(C$P):
0  3            CALL INSERT$QUEUE(.SD$HEAD.C$P):
1  3        END:
2  2        ELSE DO:
3  3            IF I$D.F=FREE THEN I$D.F=EMPTY:
5  3                        ELSE IF I$D.EH <> NIL THEN CALL I$D.EH:
           END:
8  2        CALL    DISPATCH:
9  2    END:
```

```
        $EJECT
        /*****************************************************/
        /**              SIGNAL EVENT                       **/
        /** PROCEDURE : SIGNAL                              **/
        /** FUNCTION  : RELEASE ANY TASK WAITING FOR THE    **/
        /**           EVENT LEVEL SPECIFIED. OR ACTIVE      **/
        /**           THE INTERRUPT SERVICE ROUTINE         **/
        /** INPUTS    : EVENT LEVEL                         **/
        /** OUTPUTS   : NONE                                **/
        /** HISTORY   : CREATED 11.12.1983                  **/
        /** NOTICE    :                                     **/
        /*****************************************************/

140  1  SIGNAL: PROCEDURE(LEVEL) PUBLIC:
141  2  DECLARE LEVEL   BYTE:
142  2  DECLARE I$D$P   PTR.
               I$D BASED   I$D$P   INTERRUPT$DESCRIPTOR:
143  2  DECLARE TEMP    PTR:

144  2      I$D$P=INTERRUPT$VECTOR(LEVEL):      /* GET INTERRUPT DESCRIPTOR ADDRESS */
145  2      IF I$D.F = FULL THEN
146  2      DO:
147  3          I$D.F= EMPTY:
148  3          IF I$D.T = EX THEN
149  3          DO:
150  4              CALL    REMOVE$QUEUE(.SQ$HEAD_I$D.TASK):
151  4              CALL    INSERT$READY$QUEUE(I$D.TASK):
152  4          END:
153  3          ELSE    IF I$D.SRV () NIL THEN CALL I$D.SRV:
           END:
156  2      ELSE IF I$D.F= EMPTY THEN I$D.F=FREE:
158  2                  ELSE    IF I$D.EH () NIL THEN CALL I$D.EH:
           CALL    DISPATCH:
161  2  END:
```

```
        $EJECT
        /*************************************************************/
        /**       GET HEAD OF ANY EXCHANGE                          **/
        /**                                                         **/
        /**  PROCEDURE : GET$EXCHANGE$HEAD                          **/
        /**  FUNCTION  : GET HEAD OF ANY EXCHANGE                   **/
        /**  CALL      : CALL GET$EXCHANGE$HEAD(E$D.WHERE)          **/
        /**  HISTORY   : CREATED AT 06.11.1983                      **/
        /**                                                         **/
        /*************************************************************/

52  1   GET$EXCHANGE$HEAD: PROCEDURE(E$D.WHERE) PUBLIC:
53  2   DECLARE (E$D.WHERE) POINTER .
              D$D BASED   WHERE POINTER.
              I$D BASED E$D DESCRIPTOR:
54  2       D$D=I$D.L:
55  2   END:
```

```
$EJECT
/*************************************************************/
/*          PUT MESSAGE INTO EXCHANGE BUFFER                 */
/*                                                           */
/*   PROCEDURE : PUT$EXCHANGE$MESSAGE                        */
/*   FUNCTION  : PUT MESSAGE INTO EXCHANGE BUFFER            */
/*   CALL      : PUT$EXCHANGE$MESSAGE(E$L$D);                */
/*   HISTORY   : CREATED AT 06.05.1983                       */
/*************************************************************/

166  1    PUT$EXCHANGE$MESSAGE: PROCEDURE(E$L$D) PUBLIC ;

167  2    DECLARE (E$L$D.                  /*POINTER TO EXCHANGE LINK DESCRIPTOR*/
                  E$D.                    /*POINTER TO EXCHANGE DESCRIPTOR */
                  D$D                     /* DUMMY TASK DESCRIPTOR */
                        ) POINTER.

             E$LINK  BASED E$L$D EXCHANGE$LINK$DESCRIPTOR.
             E       BASED E$D   EXCHANGE$DESCRIPTOR;


168  2        E$D=E$LINK.L;
169  2        E.C=E.C+1;
170  2        IF E.LNG()0 THEN    CALL MOVE(E.LNG..E.BUF(E.MF).E$LINK.M);
172  2        IF (E.MF=(E.MF+E.L))=E.BSZ THEN E.MF=0;
174  2        IF CHECK$EXCHANGE(E$D) IS EMPTY THEN
175  2        DO;
176  3            CALL GET$EXCHANGE$HEAD(E$D..D$D);
177  3            CALL REMOVE$QUEUE(.E.L.D$D);
178  3            CALL REMOVE$DELAY$QUEUE(D$D);
179  3            CALL INSERT$READY$QUEUE(D$D);
180  3        END;
181  2        ELSE IF E.C=E.SZ    THEN    CALL SET$EXCHANGE(E$D.FULL);
183  2                            ELSE    CALL SET$EXCHANGE(E$D.FREE);

184  2    END;
```

```
        $EJECT
        /*************************************************************/
        /*          GET MESSAGE FROM EXCHANGE BUFFER                 */
        /*                                                           */
        /*   PROCEDURE : GET$EXCHANGE$MESSAGE                        */
        /*   FUNCTION  : PUT MESSAGE INTO EXCHANGE BUFFER            */
        /*   CALL      : GET$EXCHANGE$MESSAGE(E$L$D):                */
        /*   HISTORY   : CREATED AT 06.11.1983                       */
        /*************************************************************/
185  1  GET$EXCHANGE$MESSAGE: PROCEDURE(E$L$D) PUBLIC :

186  2  DECLARE (E$L$D.                  /*POINTER TO EXCHANGE LINK DESCRIPTOR*/
                E$D.                    /*POINTER TO EXCHANGE DESCRIPTOR */
                D$D                     /* DUMMY TASK DESCRIPTOR */
                        ) POINTER.

                E$LINK  BASED E$L$D EXCHANGE$LINK$DESCRIPTOR.
                E       BASED E$D   EXCHANGE$DESCRIPTOR:


187  2          E$D=E$LINK.L:
188  2          E.C=E.C-1:
189  2          IF E.LNG () 0 THEN  CALL MOVE(E.L.E$LINK.M..E.BUF(E.MB)):
191  2          IF (E.MB:=(E.MB+E.L))=E.BSZ THEN E.MB=0:
193  2          IF CHECK$EXCHANGE(E$D) IS FULL THEN
194  2          DO:
195  3              CALL GET$EXCHANGE$HEAD(E$D..D$D):
196  3              CALL REMOVE$QUEUE(.E.L.D$D):
197  3              CALL REMOVE$DELAY$QUEUE(D$D):
198  3              CALL INSERT$READY$QUEUE(D$D):
199  3          END:
200  2          ELSE IF E.C=0   THEN CALL SET$EXCHANGE(E$D.EMPTY):
202  2                          ELSE CALL SET$EXCHANGE(E$D.FREE):

203  2      END:
```

```
$EJECT

/*****************************************************/
/**      EXCHANGE SEND   SERVICE                    **/
/**                                                 **/
/**   PROCEDURE : SEND$GET$EXCHANGE                 **/
/**   FUNCTION  : SERVICE TO SEND & WAIT REQUESTS   **/
/**   CALL      : CALL SEND$GET$EXCHANGE(E.SG.F):   **/
/**   HISTORY   : CREATED AT 05.31.1983             **/
/*****************************************************/

204  1    SEND$GET$EXCHANGE: PROCEDURE(E$L$D.SEND$GET.EMPTY$FULL) PUBLIC :

205  2    DECLARE     E$L$D   POINTER .
            EXCHANGE$LINK BASED E$L$D EXCHANGE$LINK$DESCRIPTOR.
            E$D POINTER.
            EXCHANGE    BASED E$D EXCHANGE$DESCRIPTOR.
            T$D POINTER .
            I$T BASED T$D TASK$DESCRIPTOR.
            SEND$GET POINTER.
            EMPTY$FULL  BYTE:

206  2        E$D=EXCHANGE$LINK.L:
207  2        DO CASE EXCHANGE$LINK.K AND 3 :

         /*--------------- WAIT UNTIL SEND---------------------- */
208  3           DO:
209  4               IF CHECK$EXCHANGE(E$D) IS EMPTY$FULL THEN
210  4               DO:
211  5                   CALL REMOVE$READY$QUEUE(C$P):
212  5                   CALL INSERT$QUEUE(.EXCHANGE.L.C$P):
213  5                   CALL DISPATCH:
214  5               END:
215  4               CALL SEND$GET(E$L$D):
216  4           END:

         /*------------ DO NOT WAIT IF NOT SEND ----------------*/
217  3           IF CHECK$EXCHANGE(E$D) IS EMPTY$FULL THEN EXCHANGE$LINK.S=FAIL:
219  3           ELSE DO :
220  4               CALL SEND$GET(E$L$D):
221  4               EXCHANGE$LINK.S=SUCCESS:
222  4           END:

         /*----------- WAIT TIME-OUT OR SEND -------------------*/
223  3           IF CHECK$EXCHANGE(E$D) IS EMPTY$FULL THEN
224  3           DO :
225  4               CALL REMOVE$READY$QUEUE(C$P):
226  4               CALL INSERT$QUEUE(.EXCHANGE.L.C$P):
227  4               CALL INSERT$DELAY$QUEUE(C$P):
228  4               CALL DISPATCH:
229  4               IF CHECK$EXCHANGE(E$D) IS EMPTY$FULL
231  5                   THEN DO:
233  5                       IF EXCHANGE$LINK.EH <> NIL  THEN CALL EXCHANGE$LINK.EH:
234  4                   END:
235  4                   ELSE CALL SEND$GET(E$L$D):
236  3           END:
            ELSE CALL SEND$GET(E$L$D):
```

```
        /*--------- RESERVED ACTION ------------------------*/
237  3          :

238  3        END:
239  2      END:

        /*************************************************************/
        /*       SEND EXCHANGE MESSAGE                               */
        /*                                                           */
        /*  PROCEDURE : SEND$EXCHANGE                                */
        /*  FUNCTION  : SEND EXCGANGE MESSAGE TO  EXCHANGE           */
        /*  CALL      : CALL SEND$EXCHANGE(E$L$D):                   */
        /*  HISTORY   : CREATED AT 05.06.1983                        */
        /*************************************************************/
240  1      SEND$EXCHANGE: PROCEDURE(E$L$D) PUBLIC:
241  2      DECLARE E$L$D POINTER:

242  2          CALL    SEND$GET$EXCHANGE(E$L$D..PUT$EXCHANGE$MESSAGE.FULL):
243  2      END:
```

```
$EJECT
/*************************************************************/
/*       WAIT EXCHANGE MESSAGE                               */
/*                                                           */
/*  PROCEDURE : WAIT$EXCHANGE                                */
/*  FUNCTION  : GET EXCGANGE MESSAGE FROM EXCHANGE           */
/*  CALL      : CALL WAIT$EXCHANGE(E$L$D):                   */
/*  HISTORY   : CREATED AT 05.06.1983                        */
/*************************************************************/

244  1    WAIT$EXCHANGE: PROCEDURE(E$L$D) PUBLIC :
245  2    DECLARE E$L$D POINTER :

246  2        CALL    SEND$GET$EXCHANGE(E$L$D..GET$EXCHANGE$MESSAGE.EMPTY):
247  2    END:
```

```
$EJECT
/*************************************************************/
/**        REMOVE INTERRUPT MASK OF A GIVEN LEVEL           **/
/**                                                         **/
/**  PROCEDURE : LEVEL$ON                                   **/
/**  FUNCTION  : REMOVE INTERRUPT MASK OF A LEVEL           **/
/**  CALL      : CALL FOX(LEVEL$ON.LEVEL)                   **/
/**  HISTORY   : CREATED 12.25.1983                         **/
/**                                                         **/
/*************************************************************/

248  1    LEVEL$ON: PROCEDURE(LEVEL) PUBLIC REENTRANT;
249  2        DECLARE LEVEL   BYTE;

250  2        OUTPUT(MASK$PORT)=INPUT(MASK$PORT) AND NOT BITS(LEVEL);
251  2    END;
```

```
$EJECT
/*********************************************************/
/**      MASK INTERRUPTS OF A GIVEN LEVEL               **/
/**                                                     **/
/**  PROCEDURE : LEVEL$OFF                              **/
/**  FUNCTION  : MASK INTERRUPTS OF A GIVEN LEVEL       **/
/**  CALL      : CALL FOX(LEVEL$OFF,LEVEL)              **/
/**  HISTORY   : CREATED AT 12.25.1983                  **/
/**                                                     **/
/*********************************************************/

LEVEL$OFF: PROCEDURE(LEVEL) PUBLIC REENTRANT;
    DECLARE LEVEL   BYTE;

    OUTPUT(MASK$PORT)=INPUT(MASK$PORT) OR BITS(LEVEL);
END;
```

```
$EJECT
/*********************************************************/
/**      ISSUE END OF INTERRUPT TO INTERRUPT DECODER     **/
/**                                                      **/
/**  PROCEDURE : END$INT                                 **/
/**  FUNCTION  : ISSUE AND OF INTERRUPT TO PIC [8259]    **/
/**  CALL      : CALL FOX(END$INT.NIL)                   **/
/**  HISTORY   : CREATED AT 12.25.1983                   **/
/**                                                      **/
/*********************************************************/

256  1      END$INT: PROCEDURE PUBLIC:
257  2          OUTPUT(PIC$CONTROL$PORT)=EOI:
258  2      END:
```

```
            $EJECT
            /*******************************************************/
            /**      TIMER INTERRUPT SERVICE                      **/
            /**                                                   **/
            /**   PROCEDURE : TIMER$INTERRUPT                     **/
            /**   FUNCTION  : SERVICE TO TIMER INTERRUPTS         **/
            /**   CALL      : VIA TIMER INTERRUPT                 **/
            /**   HISTORY   : CREATED AT 05.29.1983               **/
            /*******************************************************/

259   1     TIMER$INTERRUPT: PROCEDURE INTERRUPT 7 :

260   2     DECLARE D$T BASED DQ$HEAD TASK$DESCRIPTOR:
261   2     DECLARE T$D POINTER:

262   2         IF  DQ$HEAD <> NIL THEN
263   2         DO:
264   3             D$T.C=D$T.C-1:
265   3             DO WHILE (D$T.C = 0) AND (DQ$HEAD <> NIL ):
266   4                 CALL REMOVE$QUEUE(D$T.E.DQ$HEAD):
267   4                 CALL INSERT$READY$QUEUE(DQ$HEAD):
268   4                 CALL REMOVE$DELAY$QUEUE(DQ$HEAD):
269   4             END:
270   3         END:
271   2         CALL    REMOVE$READY$QUEUE(C$P):        /*REMOVE CURRENT TASK*/
272   2         CALL    INSERT$READY$QUEUE(C$P):        /*REINSSERT */
273   2         CALL    DISPATCH:
274   2     END:

            /*****************************************************************/
            /************** END OF USER MODULE ***************************/
            /*****************************************************************/

275   1     END:



MODULE INFORMATION:

    CODE AREA SIZE     = 0850H   2128D
    VARIABLE AREA SIZE = 0048H     72D
    MAXIMUM STACK SIZE = 000CH     12D
    820 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
-II PL/M-80 V3.0 COMPILATION OF MODULE OBJECTDEFINITIONS
BJECT MODULE REQUESTED
ILER INVOKED BY:  PLM80 OBJECT WORKFILES(:F0:,:F0:) PRINT(:LP:) NOOBJECT


        /*****************************************************/
        /**          MASTER THESIS CODE SECTION             **
        /**                                                 **
        /**   THIS MODULE DEFINES ALL OF THE FOX OBJECTS     **
        /**                                                 **
        /**  MODULE NAME : OBJECT$DEFINITIONS               **
        /**  FILE   NAME : OBJECT                           **
        /**  AUTHOR      : SEDAT YILMAZER                   **
        /**  DATE        : 5.12.1983                        **
        /**                                                 **
        /*****************************************************/

1       OBJECT$DEFINITIONS:
        DO :
        $NOLIST

6  1    DECLARE BITS(*) BYTE PUBLIC DATA(01H,02H,04H,08H,10H,20H,40H,80H);
```

```
$EJECT
/*******************************************************/
/**         DISPATCH CPU BETWEEN TASKS                **/
/**                                                   **/
/** PROCEDURE : DISPATCH                              **/
/** FUNCTION  : GIVE THE CPU TO THE HIGHEST PRIORITY**/
/**             READY TASK. WHICH IS BY DEFINITION    **/
/**             THE HEAD OF READY QUEUE               **/
/** CALL      : DISPATCH                              **/
/** HISTORY   : 09.20.1983                            **/
/**                                                   **/
/*******************************************************/

17  1   DISPATCH: PROCEDURE PUBLIC :

18  2   DECLARE CURRENT$TASK    BASED C$P TASK$DESCRIPTOR.
           NEW$TASK    BASED RQ$HEAD   TASK$DESCRIPTOR:

19  2       CURRENT$TASK.SP=STACK$PTR:
20  2       STACK$PTR=NEW$TASK.SP:
21  2   END:
```

```
        $EJECT
        /*******************************************************/
        /*              COMPARE TWO STRINGS                    */
        /*                                                     */
        /*  PROCEDURE  : COMPARE                               */
        /*  FUNCTION   : COMPARE TWO STRINGS                   */
        /*  CALL       : COMPARE(.S1..S2.LENGTH)               */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

22  1   COMPARE: PROCEDURE(SPTR1.SPTR2.LNGTH) BOOLEAN PUBLIC REENTRANT;

23  2   DECLARE (SPTR1.SPTR2)   POINTER .
                LNGTH           BYTE .
                I               BYTE .

            (S1 BASED SPTR1)(1) BYTE .
            (S2 BASED SPTR2)(1) BYTE ;

24  2       DO I=0 TO LNGTH-1 ;
25  3           IF S1(I) <> S2(I) THEN RETURN FALSE ;
27  3       END ;
28  2       RETURN TRUE ;
29  2   END ;
```

```
$EJECT
/*******************************************************/
/*              SET PROCESS STATUS                     */
/*                                                     */
/*  PROCEDURE  : SET$STATUS                            */
/*  FUNCTION   : SET THE STATUS OF THE CURRENT TASK    */
/*               TO INCOMMINGSTATUS                    */
/*  CALL       : SET$STATUS(STATUS)                    */
/*  HISTORY    : CREATED AT 05.14.1983                 */
/*                                                     */
/*******************************************************/

30  1   SET$STATUS: PROCEDURE(T$D.STAT) PUBLIC REENTRANT:

31  2   DECLARE (STATE.STAT) BYTE.
        T$D POINTER .
        I$T BASED T$D TASK$DESCRIPTOR :

32  2       STATE=STAT AND 1:
33  2       STAT=BITS(SHR(STAT.1)):
34  2       IF STATE    THEN I$T.ST=I$T.ST OR STAT :
36  2              ELSE I$T.ST=I$T.ST AND (NOT STAT):
37  2   END :
```

```
        $EJECT
        /*******************************************************/
        /*              CHECK PROCESSOR STATUS                 */
        /*                                                     */
        /*  PROCEDURE  : CHECK$STATUS                          */
        /*  FUNCTION   : CHECK PROCESS STATUS OF THE CURRENT   */
        /*               TASK                                  */
        /*  CALL       : CHECK$STATUS(STATUS)                  */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

E8  1   CHECK$STATUS: PROCEDURE(T$P.STAT) BYTE PUBLIC REENTRANT:

39  2   DECLARE (STATE.STAT) BYTE .
            T$P POINTER.
            I$T BASED T$P TASK$DESCRIPTOR :

40  2       STATE=STAT AND 1 :
41  2       STAT=(BITS(SHR(STAT.1)) AND I$T.ST)()0:
42  2       IF STATE    THEN RETURN STAT:
44  2              ELSE RETURN NOT STAT:
45  2   END :
```

```
        $EJECT
        /*******************************************************/
        /*              SET EXCHANGE STATUS                    */
        /*                                                     */
        /*  PROCEDURE  : SET$EXCHANGE                          */
        /*  FUNCTION   : SET THE STATUS OF THE EXCHANGE        */
        /*               TO INCOMMINGSTATUS                    */
        /*  CALL       : SET$EXCHANGE(E$P.STATUS)              */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/
46  1   SET$EXCHANGE:.PROCEDURE(E$P.STATUS) PUBLIC REENTRANT:
47  2   DECLARE E$P PTR.
            STATUS  BYTE.
            EXCHANGE    BASED   E$P EXCHANGE$DESCRIPTOR:

48  2       EXCHANGE.S=STATUS:

49  2   END:


        /*******************************************************/
        /*            CHECK EXCHANGE STATUS                    */
        /*                                                     */
        /*  PROCEDURE  : CHECK$EXCHANGE                        */
        /*  FUNCTION   : CHECK THE STATUS OF THE EXCHANGE      */
        /*  CALL       : CHECK$EXCHANGE(E$P)                   */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/
50  1   CHECK$EXCHANGE: PROCEDURE(E$P) BYTE PUBLIC REENTRANT:
51  2   DECLARE E$P PTR.
            EXCHANGE    BASED   E$P EXCHANGE$DESCRIPTOR:

52  2       RETURN  EXCHANGE.S:
53  2   END:
```

```
        $EJECT
        /*******************************************************/
        /**     INSERT A DESCRIPROR INTO ANY OF THE QUEUES    **/
        /**                                                   **/
        /**  PROCEDURE : INSERT$QUEUE                         **/
        /**  FUNCTION  : INSERT A DESCRIPTOR IN TO A QUEUE    **/
        /**  INPUTS    : QUEUE HEADER AND THE DESCRIPTOR ADR  **/
        /**  OUTPUTS   : DESCRIPTOR INSERTTED TO THE QUEUE    **/
        /**  PLM CALL  : CALL INSERT$QUEUE(.QUEUE$HEAD.D):    **/
        /**  HISTORY   : CREATED AT 11.06.1983                **/
        /**  NOTICE    : CURRENTLY ONLY SUSPEND  QUEUE        **/
        /**                            EXCHANGE QUEUE         **/
        /**              ARE SUPPORTED                        **/
        /*******************************************************/
54  1   INSERT$QUEUE: PROCEDURE(Q$P,I$P) PUBLIC REENTRANT;

55  2   DECLARE (Q$P          /* POINTER TO QUEUE DESCRIPTOR */
            I$P         /* POINTER TO INCOMMING DESCRIPTOR */
               ) POINTER;
56  2   DECLARE Q$D BASED   Q$P DESCRIPTOR; /* QUEUE DESCRIPTOR */
57  2   DECLARE I$D BASED   I$P DESCRIPTOR; /* INCOMMING DESCRIPTOR */

58  2       CALL    SET$STATUS(I$P.SUSPENDED);
59  2       I$D.L=Q$D.L;
60  2       Q$D.L=I$P;

61  2   END INSERT$QUEUE;
```

```
$EJECT
/*******************************************************/
/**     REMOVE A DESCRIPROR FROM ANY OF THE QUEUES     **/
/**                                                    **/
/**  PROCEDURE : REMOVE$QUEUE                          **/
/**  FUNCTION  : REMOVE A DESCRIPTOR FROM  A QUEUE     **/
/**  INPUTS    : QUEUE HEADER AND THE DESCRIPTOR ADR   **/
/**  OUTPUTS   : DESCRIPTOR REMOVED   TO THE QUEUE     **/
/**  PLM CALL  : CALL REMOVE$QUEUE(.QUEUE$HEAD.D):     **/
/**  HISTORY   : CREATED AT 11.06.1983                 **/
/**  NOTICE    : CURRENTLY ONLY READY QUEUE.           **/
/**                              SUSPEND QUEUE.        **/
/**                              EXCHANGE QUEUE.       **/
/**              ARE SUPPORTED                         **/
/*******************************************************/
62  1   REMOVE$QUEUE: PROCEDURE(Q$P.I$P) PUBLIC REENTRANT:

63  2   DECLARE (Q$P.           /* PIONTER TO QUEUE DESCRIPTOR */
            I$P                 /* POINTER TO INCOMMING DESCTRIPTOR */
                ) POINTER:
64  2   DECLARE Q$D BASED Q$P DESCRIPTOR:   /* QUEUE DESCRIPTOR */
65  2   DECLARE I$D BASED I$P DESCRIPTOR:   /* INCOMMING DESCRIPTOR */

66  2       DO WHILE (Q$P()NIL) AND (Q$D.L()I$P):
67  3           Q$P=Q$D.L:
68  3       END:
69  2       IF  Q$P() NIL THEN
70  2       DO:
71  3           Q$D.L=I$D.L:
72  3           CALL SET$STATUS(I$P.NOTSUSPENDED):
73  3       END:

74  2   END REMOVE$QUEUE:
```

```
        $EJECT
        /*******************************************************/
        /*              INSERT INTO SYSTEM LINK                */
        /*                                                     */
        /*  PROCEDURE  : INSERT$SYSTEM$LINK                    */
        /*  FUNCTION   : INSERT INCOMMING TASK OR EXCAHANGE    */
        /*               DESCRIPTOR INTO THE SYSTEM LINK       */
        /*  CALL       : INSERT$SYSTEM$LINK(.DESCRIPTOR)       */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

75  1   INSERT$SYSTEM$LINK: PROCEDURE(P)  PUBLIC REENTRANT :

76  2   DECLARE P POINTER .         /* DESCRIPTOR POINTER */
            I$D BASED   P   DESCRIPTOR :

77  2       I$D.SL=FL$HEAD:
78  2       FL$HEAD=P:

79  2   END :
```

```
        $EJECT
        /*******************************************************/
        /*              REMOVE FROM SYSTEM LINK                */
        /*                                                     */
        /*  PROCEDURE  : REMOVE$SYSTEM$LINK                    */
        /*  FUNCTION   : REMOVE INCOMMING DESCRIPTOR           */
        /*               FROM THE SYSTEM LINK                  */
        /*  CALL       : REMOVE$SYSTEM$LINK(.DESCRIPTOR)       */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

80  1   REMOVE$SYSTEM$LINK: PROCEDURE(I$P) PUBLIC REENTRANT;

81  2   DECLARE (I$P,P) POINTER ,
                I$D BASED   I$P DESCRIPTOR ,
                P$D BASED   P   DESCRIPTOR ;

82  2       IF FL$HEAD = I$P THEN FL$HEAD=I$D.SL ;
84  2       ELSE DO ;
85  3           P=FL$HEAD;
86  3           DO WHILE (P$D.SL <> P ) AND (P$D.SL <> NIL ) ;
87  4               P=P$D.SL ;
88  4           END ;
89  3           IF P<>NIL THEN  P$D.SL=I$D.SL;
91  3       END ;
92  2   END ;
```

```
        $EJECT
        /*******************************************************/
        /*              INSERT INTO READY QUEUE                */
        /*                                                     */
        /*  PROCEDURE  : INSERT$READY$QUEUE                    */
        /*  FUNCTION   : INSERT INCOMMING TASK DESCRIPTOR      */
        /*               INTO THE SYSTEM READY TASK QUEUE      */
        /*  CALL       : INSERT$READY$QUEUE(.TASK$DESCRIPTOR) */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

93  1   INSERT$READY$QUEUE: PROCEDURE(T$P) PUBLIC REENTRANT;

94  2   DECLARE (T$P.       /*INCOMMING TASK DESCRIPTOR POINTER*/
                D$P.        /*DUMMY TASK DESCRIPTOR (TEMP) POINTER */
                P$P         /* PREVIOUS TASK POINTER */
                    )   POINTER;

95  2   DECLARE I$T BASED T$P   TASK$DESCRIPTOR .
                D$T BASED D$P   TASK$DESCRIPTOR .
                P$T BASED P$P   TASK$DESCRIPTOR;
96  2       IF CHECK$STATUS(T$P.NOTREADY) THEN
97  2       DO;
98  3           D$P.P$P=RQ$HEAD;        /*START AT THE BEGINING */
99  3           DO WHILE I$T.P (= D$T.P;    /*WHILE DUMMY P IS HIGHER */
100 4               P$P=D$P;
101 4               D$P=D$T.L;      /*GET THE NEXT TASK */
102 4           END;
103 3           I$T.L=P$T.L;            /* ADJUST FORFARD LINK */
104 3           P$T.L=T$P;
105 3           CALL SET$STATUS(T$P.READY);
106 3       END;
107 2   END;
```

```
        $EJECT
        /*******************************************************/
        /*              REMOVE FROM READY QUEUE                */
        /*                                                     */
        /*  PROCEDURE  : REMOVE$READY$QUEUE                    */
        /*  FUNCTION   : REMOVE INCOMMING TASK DESCRIPTOR      */
        /*               FROM THE SYSTEM READY TASK QUEUE      */
        /*  CALL       : REMOVE$READY$QUEUE(.TASK$DESCRIPTOR)  */
        /*  HISTORY    : CREATED AT 05.14.1983                 */
        /*                                                     */
        /*******************************************************/

108  1  REMOVE$READY$QUEUE: PROCEDURE(T$P) PUBLIC :

109  2  DECLARE (T$P.   /*INCOMMING TASK DESCRIPTOR POINTER*/
                D$P     /*DUMMY TASK DESCRIPTOR (TEMP)*/
                ) POINTER :

110  2  DECLARE I$T BASED T$P TASK$DESCRIPTOR .
              D$T BASED D$P TASK$DESCRIPTOR :

111  2     IF CHECK$STATUS(T$P.READY) THEN
112  2     DO :
113  3        D$P=.RQ$HEAD:
114  3        DO WHILE D$P() NIL AND D$T.L()T$P:
115  4           D$P=D$T.L:
116  4        END:
117  3        IF D$P () NIL THEN
118  3        DO:
119  4           D$T.L=I$T.L:
120  4           CALL SET$STATUS(T$P.NOTREADY):
121  4        END:
122  3     END:
123  2  END REMOVE$READY$QUEUE:
```

```
$EJECT
/*******************************************************/
/*              INSERT INTO DELAY QUEUE                */
/*                                                     */
/*  PROCEDURE  : INSERT$DELAY$QUEUE                    */
/*  FUNCTION   : INSERT INCOMMING TASK DESCRIPTOR      */
/*               INTO THE SYSTEM DELAYED TASK QUEUE    */
/*  CALL       : INSERT$DELAY$QUEUE(.TASK$DESCRIPTOR)  */
/*  HISTORY    : CREATED AT 05.15.1983                 */
/*                                                     */
/*******************************************************/

124  1    INSERT$DELAY$QUEUE: PROCEDURE(T$D) PUBLIC :

125  2    DECLARE (T$D.                /*INCOMMING TASK DESCRIPTOR */
          P$D.                      /*PREVIOUS TASK DESCRIPTOR */
          N$D                    /*NEXT TASK DESCRIPTOR */
              ) POINTER .
          TOTAL   INTEGER .

          I$T BASED T$D TASK$DESCRIPTOR .
          P$T BASED P$D TASK$DESCRIPTOR .
          N$T BASED N$D TASK$DESCRIPTOR :

126  2       IF CHECK$STATUS(T$D.NOTDELAYED) THEN
127  2       DO :
128  3           TOTAL=0:
129  3           N$D.P$D=DQ$HEAD :
130  3           DO WHILE (TOTAL ( I$T.C ) AND (N$D () NIL ):
131  4               TOTAL=TOTAL+N$T.C:
132  4               P$D=N$D:
133  4               N$D=N$T.D:
134  4           END:
135  3           IF N$D () NIL  THEN TOTAL=TOTAL-N$T.C:
137  3           IF P$D = DQ$HEAD
                 THEN DO:
139  4               I$T.D=DQ$HEAD:
140  4               DQ$HEAD=T$D:
141  4           END:
142  3           ELSE DO :
143  4               I$T.D=P$T.D:
144  4               P$T.D=I$T.D:
145  4           END:
146  3           I$T.C=I$T.C-TOTAL :
147  3           CALL SET$STATUS(T$D.DELAYED):
148  3       END:
149  2   END INSERT$DELAY$QUEUE:
```

```
        $EJECT
        /*******************************************************/
        /*              REMOVE FROM DELAY QUEUE                */
        /*                                                     */
        /*  PROCEDURE  : REMOVE$DELAY$QUEUE                    */
        /*  FUNCTION   : REMOVE INCOMMING TASK DESCRIPTOR      */
        /*               FROM THE SYSTEM DELAYED TASK QUEUE    */
        /*  CALL       : REMOVE$DELAY$QUEUE(.TASK$DESCRIPTOR)  */
        /*  HISTORY    : CREATED AT 05.15.1983                 */
        /*                                                     */
        /*******************************************************/

50  1   REMOVE$DELAY$QUEUE: PROCEDURE(T$D) PUBLIC :

51  2   DECLARE (T$D.               /*INCOMMING TASK DESCRIPTOR */
                P$D                 /*PREVIOUS TASK DESCRIPTOR  */
                ) POINTER.
            I$T BASED T$D TASK$DESCRIPTOR .
            P$T BASED P$D TASK$DESCRIPTOR :

152 2       IF CHECK$STATUS(T$D.DELAYED) THEN
153 2       DO:
154 3           IF T$D=DQ$HEAD
                THEN DO :
156 4               DQ$HEAD=I$T.D :
157 4               P$D=DQ$HEAD :
158 4               P$T.C=P$T.C+I$T.C:
159 4           END :
160 3           ELSE DO :
161 4               P$D=DQ$HEAD :
162 4               DO WHILE (P$T.D <> T$D) AND (P$T.D <> NIL ):
163 5                   P$D=P$T.D :
164 5               END:
165 4               IF P$D = NIL THEN RETURN :
167 4               P$T.D=I$T.D:
168 4               P$D=P$T.D:
169 4               P$T.C=P$T.C+I$T.C:
170 4           END:
171 3           CALL SET$STATUS(T$D.NOT$DELAYED):
172 3       END:
173 2   END REMOVE$DELAY$QUEUE:

        /*******************************************************/

174 1   END :
```

```
MODULE INFORMATION:

  CODE AREA SIZE     = 0558H   1368D
  VARIABLE AREA SIZE = 0020H     32D
  MAXIMUM STACK SIZE = 0010H     16D
  695 LINES READ
  0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

APPENDIX C SAMPLE APPLICATION

```
ISIS-II PL/M-80 V3.0 COMPILATION OF MODULE INITIALIZATION
OBJECT MODULE PLACED IN :F1:THESIS.OBJ
COMPILER INVOKED BY:  P :F1:THESIS.EXP



                /*********************************************************/
                /**           SYSTEM INITIALIZATION                     **/
                /**                                                     **/
                /** PROCEDURE : INITSK                                  **/
                /** FUNCTION  : INITIALIZE THE SAMPLE SYSTEM            **/
                /** AUTHOR    : SEDAT YILMAZER                          **/
                /*********************************************************/
1               INITIALIZATION: DO:

                $NOLIST
14   1          FOX: PROCEDURE(GATE$CODE.DESCRIPTOR$ADR) EXTERNAL:
15   2          DECLARE   DESCRIPTOR$ADR ADDRESS:
16   2          DECLARE   GATE$CODE      BYTE:
17   2          END:

                   /*-------- DEFINE TASK STACK POINTERS -----------*/

18   1          DECLARE (DEBUG$STACK.CI$STACK.CO$STACK)(40) BYTE PUBLIC:

                   /*-------- DEFINE TASK DESCRIPTORS --------------*/

19   1          DECLARE (DEBUG$TASK$DESCRIPTOR.      /* FOR DEBUGGER TASK */
                        CI$TASK$DESCRIPTOR.         /* FOR CONSOLE INPUT TASK */
                        CO$TASK$DESCRIPTOR          /* FOR CONSOLE OUTPUT TASK */
                                        ) TASK$DESCRIPTOR:

20   1          DECLARE C$P ADDRESS EXTERNAL:

21   1          DEBUG: PROCEDURE EXTERNAL:
22   2          END:

23   1          DECLARE DEBUG$S$TASK$DESCRIPTOR STATIC$TASK$DESCRIPTOR
                    DATA(.DEBUG$TASK$DESCRIPTOR.    /* POINTER TO ACTUAL TASK DESCRIPTOR */
                        'DEBUG '.                   /* NAME OF THE TASK                  */
                        10.                         /* DEBUGER TASK PRIORITY             */
                        .DEBUG.                     /* ADDRESS OF THE TASK START         */
                        NIL                         /* EXCEPTION HANDLER IS NOT AVAILABLE*/
                                    ):


24   1          DECLARE CI$S$TASK$DESCRIPTOR        STATIC$TASK$DESCRIPTOR
                    DATA(.CI$TASK$DESCRIPTOR.       /* POINTER TO ACTUAL TASK DESCRIPTOR */
                        'CITASK'.                   /* NAME OF THE TASK                  */
                        11.                         /* CI TASK PRIORITY                  */
                        .CONSOLE$INPUT$TASK.        /* ADDRESS OF THE TASK START         */
                        NIL                         /* EXCEPTION HANDLER IS NOT AVAILABLE*/
                                    ):


25   1          DECLARE CO$S$TASK$DESCRIPTOR        STATIC$TASK$DESCRIPTOR
                    DATA(.CO$TASK$DESCRIPTOR.   /* POINTER TO ACTUAL TASK DESCRIPTOR */
```

```
            'COTASK'.               /* NAME OF THE TASK                  */
            09.                         /* CO TASK PRIORITY                  */
            .CONSOLE$OUTPUT.            /* ADDRESS OF THE TASK START         */
            NIL                         /* EXCEPTION HANDLER IS NOT AVAILABLE*/
                        ):


26  1   DECLARE EOM                     LITERALLY   '0DH'.
        TERMINAL$DATA$PORT$ADDRESS  LITERALLY   '0CCH'.
        CONSOLE$INPUT$EVENT$ID      LITERALLY   '00'.
        CONINT                      LITERALLY   '01':

27  1   DECLARE CO$EXCHANGE$DESCRIPTOR EXCHANGE$DESCRIPTOR:

28  1   DECLARE CI$EXCHANGE$DESCRIPTOR EXCHANGE$DESCRIPTOR:


29  1   DECLARE (CI$INTERRUPT$DESCRIPTOR.   /* FOR CONSOLE INPUT INTERRUPTS */
            CO$INTERRUPT$DESCRIPTOR     /* FOR CONSOLE OUTPUT INTERRUPTS*/
                                    ) INTERRUPT$DESCRIPTOR:

30  1   DECLARE CI$S$INTERRUPT$DESCRIPTOR   STATIC$INTERRUPT$DESCRIPTOR
        DATA(.CI$INTERRUPT$DESCRIPTOR.  /* LINK TO INTERRUPT DESCRIPTOR */
            'CI$INT'.                   /* NAME OF THE INTERRUPT        */
            EXCHANGE$TYPE.              /* TYPE OF THE INTERRUPT SERVICE*/
            NIL.                        /* SINCE NOT A FAST TYPE INTERRUPT*/
            NIL.                        /* NO EXCEPTION HANDLER         */
            1                           /* INTERRUPT LEVEL              */
                    ):

31  1   DECLARE CO$S$INTERRUPT$DESCRIPTOR   STATIC$INTERRUPT$DESCRIPTOR
        DATA(.CO$INTERRUPT$DESCRIPTOR.  /* LINK TO INTERRUPT DESCRIPTOR */
            'CO$INT'.                   /* NAME OF THE INTERRUPT        */
            EXCHANGE$TYPE.              /* TYPE OF THE INTERRUPT SERVICE*/
            NIL.                        /* SINCE NOT A FAST TYPE INTERRUPT*/
            NIL.                        /* NO EXCEPTION HANDLER         */
            2                           /* INTERRUPT LEVEL              */
                    ):


32  1   DECLARE CO$S$EXCHANGE$DESCRIPTOR STATIC$EXCHANGE$DESCRIPTOR
            DATA(.CO$EXCHANGE$DESCRIPTOR.'CONOUT'.1.80):


33  1   DECLARE CI$S$EXCHANGE$DESCRIPTOR STATIC$EXCHANGE$DESCRIPTOR
            DATA(.CI$EXCHANGE$DESCRIPTOR.'CONINP'.1.80):

34  1   DECLARE X$TASK$LINK$DESCRIPTOR  TASK$LINK$DESCRIPTOR:

35  1   DECLARE I BYTE:


36  1   INITASK: PROCEDURE PUBLIC:

37  2       CALL FOX    (CREATE$EXCHANGE..CI$S$EXCHANGE$DESCRIPTOR):
38  2       CALL FOX    (CREATE$EXCHANGE..CO$S$EXCHANGE$DESCRIPTOR):
```

```
2           CALL FOX   (CREATE$TASK..DEBUG$S$TASK$DESCRIPTOR):
2           CALL FOX   (CREATE$TASK..CI$S$TASK$DESCRIPTOR):
2           CALL FOX   (CREATE$TASK..CO$S$TASK$DESCRIPTOR):

2           CALL FOX   (CREATE$INTERRUPT..CI$S$INTERRUPT$DESCRIPTOR):
2           CALL FOX   (CREATE$INTERRUPT..CO$S$INTERRUPT$DESCRIPTOR):

2           CALL MOVE  (.('CITASK')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL MOVE  (.('COTASK')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL MOVE  (.('DEBUG ')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL FOX   (SUSPEND.C$P):
2       END:

        /*******************************************************/
        /**          CONSOLE INPUT TASK BODY                  **/
        /**                                                   **/
        /**  PROCEDURE : CONSOLE$INPUT$TASK                   **/
        /**  FUNCTION  : COLLECT CHARACTERS FROM THE CONSOLE  **/
        /**              DEVICE AND AT THE AND SEND TO THE    **/
        /**              DEBUGGER TASK FOR FURTHER PROCESSING **/
        /**  INPUTS    : NONE                                 **/
        /**  OUTPUTS   : NONE                                 **/
        /*******************************************************/


1       CONSOLE$INPUT$TASK: PROCEDURE PUBLIC:

2       DECLARE   CONOUT$EXCHANGE$LINK  EXCHANGE$LINK$DESCRIPTOR:
2       DECLARE   CHR$BUF(2)        BYTE:

2       DECLARE   CI$EXCHANGE$LINK    EXCHANGE$LINK$DESCRIPTOR:
2       DECLARE   CI$BUF(80)     BYTE:


2           CALL MOVE(6..('CONOUT').CONOUT$EXCHANGE$LINK.NAME(0)):
2           CONOUT$EXCHANGE$LINK.M=.CHR$BUF(1):      /* MESSAGE ADDRESS */
2           CHR$BUF(0)=1:
2           CALL FOX   (LINK$EXCHANGE..CONOUT$EXCHANGE$LINK):
2           CALL MOVE(6..('CONINP')..CI$EXCHANGE$LINK.NAME(0)):
2           CALL FOX   (LINK$EXCHANGE..CI$EXCHANGE$LINK):
2           CI$EXCHANGE$LINK.M=.CI$BUF:
2           CI$BUF(0)=0:        /* USE AS CHARACTER COUNT */
2           DO WHILE 1:
3               CALL FOX  (EVENT.CONSOLE$INPUT$EVENT$ID):
3               CHR$BUF(1).CI$BUF(CI$BUF(0))=INPUT(TERMINAL$DATA$PORT$ADDRESS):
3               CALL FOX  (SEND..CONOUT$EXCHANGE$LINK):
3               IF CHR$BUF(1)=EOM THEN
```

```
2           CALL FOX   (CREATE$TASK..DEBUG$S$TASK$DESCRIPTOR):
2           CALL FOX   (CREATE$TASK..CI$S$TASK$DESCRIPTOR):
2           CALL FOX   (CREATE$TASK..CO$S$TASK$DESCRIPTOR):

2           CALL FOX   (CREATE$INTERRUPT..CI$S$INTERRUPT$DESCRIPTOR):
2           CALL FOX   (CREATE$INTERRUPT..CO$S$INTERRUPT$DESCRIPTOR):

2           CALL MOVE  (.('CITASK')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL MOVE  (.('COTASK')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL MOVE  (.('DEBUG ')..X$TASK$LINK$DESCRIPTOR.NAME.6):
2           CALL FOX   (LINK$TASK..X$TASK$LINK$DESCRIPTOR):
2           CALL FOX   (RESUME..X$TASK$LINK$DESCRIPTOR):

2           CALL FOX   (SUSPEND.C$P):
2       END:

        /*******************************************************/
        /**          CONSOLE INPUT TASK BODY                  **/
        /**                                                   **/
        /**  PROCEDURE : CONSOLE$INPUT$TASK                   **/
        /**  FUNCTION  : COLLECT CHARACTERS FROM THE CONSOLE  **/
        /**              DEVICE AND AT THE AND SEND TO THE    **/
        /**              DEBUGGER TASK FOR FURTHER PROCESSING **/
        /**  INPUTS    : NONE                                 **/
        /**  OUTPUTS   : NONE                                 **/
        /*******************************************************/


1       CONSOLE$INPUT$TASK: PROCEDURE PUBLIC:

2       DECLARE   CONOUT$EXCHANGE$LINK  EXCHANGE$LINK$DESCRIPTOR:
2       DECLARE   CHR$BUF(2)        BYTE:

2       DECLARE   CI$EXCHANGE$LINK    EXCHANGE$LINK$DESCRIPTOR:
2       DECLARE   CI$BUF(80)      BYTE:


2           CALL MOVE(6..('CONOUT').CONOUT$EXCHANGE$LINK.NAME(0)):
2           CONOUT$EXCHANGE$LINK.M=.CHR$BUF(1):      /* MESSAGE ADDRESS */
2           CHR$BUF(0)=1:
2           CALL FOX   (LINK$EXCHANGE..CONOUT$EXCHANGE$LINK):
2           CALL MOVE(6..('CONINP')..CI$EXCHANGE$LINK.NAME(0)):
2           CALL FOX   (LINK$EXCHANGE..CI$EXCHANGE$LINK):
2           CI$EXCHANGE$LINK.M=.CI$BUF:
2           CI$BUF(0)=0:        /* USE AS CHARACTER COUNT */
2           DO WHILE 1:
3               CALL FOX  (EVENT.CONSOLE$INPUT$EVENT$ID):
3               CHR$BUF(1).CI$BUF(CI$BUF(0))=INPUT(TERMINAL$DATA$PORT$ADDRESS):
3               CALL FOX  (SEND..CONOUT$EXCHANGE$LINK):
3               IF CHR$BUF(1)=EOM THEN
```

```
 73   3              DO:
 74   4                  CALL  FOX (SEND..CI$EXCHANGE$LINK): /* SENT TO DEBUGGER */
 75   4                  CI$BUF(0)=0:   /* AFTER SEND OPERATION BUFFER CAN BE MODIFIED */
 76   4              END:
 77   3              ELSE CI$BUF(0)=CI$BUF(0)+1:
 78   3          END:
 79   2      END:

          /*******************************************************/
          /**         CONSOLE OUTPUT TASK BODY                  **/
          /**                                                   **/
          /**  PROCEDURE : CONSOLE$OUTPUT$TASK                  **/
          /**  FUNCTION  : SENDS CHARACTERS TO THE CONSOLE      **/
          /**              DEVICE .                             **/
          /**  INPUTS    : NONE                                 **/
          /**  OUTPUTS   : NONE                                 **/
          /*******************************************************/

 80   1      CONSOLE$OUTPUT: PROCEDURE PUBLIC:


 81   2      DECLARE   CO$EXCHANGE$LINK    EXCHANGE$LINK$DESCRIPTOR:
 82   2      DECLARE   CO$BUF(80)     BYTE:

 83   2          CALL MOVE(6..('CONOUT')..CO$EXCHANGE$LINK.NAME(0)):
 84   2          CALL FOX   (LINK$EXCHANGE..CO$EXCHANGE$LINK):
 85   2          CO$EXCHANGE$LINK.M=.CO$BUF:

 86   2          DO WHILE 1:
 87   3              CALL FOX  (WAIT..CO$EXCHANGE$LINK):
 88   3              IF CO$EXCHANGE$LINK.S=SUCCESS THEN
 89   3              DO I=0 TO CO$BUF(0):
 90   4                  CALL FOX  (EVENT.CONINT):
 91   4                  OUTPUT(TERMINAL$DATA$PORT$ADDRESS)=CO$BUF(I+1):
 92   4              END:
 93   3          END:
 94   2      END:


 95   1      END: /* OF DEMOSTRATION MODULE */


MODULE INFORMATION:

    CODE AREA SIZE     = 0217H    535D
    VARIABLE AREA SIZE = 020EH    526D
    MAXIMUM STACK SIZE = 0002H      2D
    463 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

REFERENCES

Brinch, Hansen. Operating System Principles.

Henry, Katzan. Operating Systems.

Peter, Wagner. Introduction to System Programming.

Wayne, Black. Design of Real Time Applications.

Dionysios, C. Tsichritizis. Operating System Principles.

Semih Pekol, Sedat Yilmazer. MTEX user guide.

Intel. PL/M Programmer User Manual.

Intel. SBC 80/20 and SBC 80/20-4 Single Board Computers Hardware Reference Manual.

Intel. Intellec Series II Microcomputer Development System Hardware Interface Manual.

Intel. Intellec Series II Microcomputer Development System Hardware Reference Manual.

Intel. PL/M-80 Programming Manual.

Intel. ISIS-II PL/M Compiler Operator's Manual.

Intel. Component Data Catalog.

Digital Research. MP/M-II System programmer guide.