DEC-11-IRSAA-A-D

RSX11A programmer's reference manual

For additional copies, order No. DEC-11-IRSAA-A-D from
Software Distribution Center, Digital Equipment Corporation,
Maynard, Massachusetts 01754

First Printing, March, 1973

Your attention is invited to the last two pages of this document. The "How to Obtain Software Information" page tells you how to keep up-to-date with DEC's software. Completion and return of the "Reader's Comments" page is beneficial to both you and DEC; all comments received are acknowledged and are considered when documenting subsequent manuals.

Copyright © 1973 by Digital Equipment Corporation

The software described in this manual is furnished to purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

The material in this document is for information purposes only and is subject to change without notice. DIGITAL assumes no responsibility for the use or reliability of software and equipment which is not supplied by it. DIGITAL assumes no responsibility for any errors which may appear in this document.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

CDP	DIGITAL	KA10	QUICKPOINT
COMPUTER LABS	EDUSYSTEM	LAB-8	RAD-8
COMTEX-11	FLIP CHIP	OMNIBUS	RSTS
DDT	FOCAL	OS/8	RSX
DEC	GLC-8		SABR
DECTAPE	IDACS	PDP	TYPESET-8
DIBOL	INDACS	PHA	UNIBUS

Teletype is a registered trademark of the Teletype Corporation.

PREFACE

RSX11A is a small real-time multiprogramming operating system for Digital Equipment Corporation's PDP-11 family of computers. It will perform well in any real-time environment where multiprogramming, time-dependent scheduling, and low core residency are prime requirements. Typical applications include laboratory automation, process control, and dedicated remote or satellite processing.

The primary intent of this manual is to provide a source of reference for this system, although specific information on how to use and generate the system is also included. Chapter 1 contains a general description of the overall system. Chapter 2 describes executive (programmed) requests. Chapter 3 explains operator communication via the operator's console. Chapter 4 contains sample tasks. Chapter 5 is dedicated to system generation. Appendices are included to summarize executive requests, error definitions, device dependent information, operating system data structures, and the structure of the 8K core-only RSX11A load module available from the Software Distribution Center.

A thorough knowledge of PDP-11 assembly language is assumed throughout the manual.

TABLE OF CONTENTS

PAGE

Chapter 1 - General Description

1.1	Introduction	1-1
1.2	Hardware Configuration	1-1
1.3	Terminology	1-2
1.4	System Overview	1-6
1.5	Manual Conventions	1-9

Chapter 2 - Executive Communication

2.1	General Description	2-1
2.1.1	Summary of Executive Requests	2-1
2.1.2	Executive Request Calling Sequence	2-3
2.2	Detailed ER Descriptions	2-4
2.2.1	Task Termination	2-4
2.2.1.1	Terminate Task Execution	2-4
2.2.1.2	Delete Task From System	2-5
2.2.2	Timer	2-6
2.2.2.1	Request Timed Wait	2-6
2.2.2.2	Request Timed Interrupt	2-8
2.2.2.3	Cancel Timed Interrupt Request	2-9
2.2.2.4	Request Time of Day	2-10
2.2.2.5	Request Date	2-11
2.2.3	Task Initiation	2-11
2.2.3.1	Request Task Execution	2-11
2.2.3.2	Request Synchronous Periodic Task Execution	2-13
2.2.3.3	Request Asynchronous Periodic Task Execution	2-14
2.2.3.4	Request Task Execution at Time of Day	2-16
2.2.4	Task Synchronization	2-17
2.2.4.1	Suspend Task	2-17
2.2.4.2	Activate Task	2-18
2.2.4.3	Test and Set Task Group Lock Return Immediate	2-19
2.2.4.4	Test and Set Task Group Lock Wait	2-21
2.2.4.5	Reset Task Group Lock	2-22
2.2.5	Intertask Communication	2-23
2.2.5.1	Send Message to Task	2-23
2.2.5.2	Receive Message From Task	2-24
2.2.6	Input/Output	2-25
2.2.6.1	Request I/O Operation	2-25
2.2.7	End Action Control	2-32
2.2.7.1	End Action Wait	2-32
2.2.7.2	End Action Return	2-33
2.2.8	List Manipulation	2-33
2.2.8.1	Define List	2-33
2.2.8.2	Remove Entry From List	2-36
2.2.8.3	Make Entry In List	2-37
2.2.9	Dynamic Storage	2-38
2.2.9.1	Request Buffer Block	2-38
2.2.9.2	Release Buffer Block	2-39
2.2.10	Miscellaneous	2-40
2.2.10.1	Set Error Trap Address	2-40
2.2.10.2	Set Alternate Stack Address	2-42
2.2.10.3	Set TRAP Trap Address	2-43

Chapter 3 - Operator Communication

3.1	General Description	3-1
3.1.1	Functions Provided	3-1
3.1.2	Error Handling	3-2

3.1.3	Command Syntax	3-2
3.2	Keyins	3-2
3.2.1	Examine Memory	3-3
3.2.2	Deposit Memory	3-3
3.2.3	Enter Date	3-4
3.2.4	Enter Time	3-5
3.2.5	Request Task Execution	3-6
3.2.6	Request Asynchronous Periodic Task Execution	3-6
3.2.7	Request Synchronous Periodic Task Execution	3-7
3.2.8	Request Task Execution at Time of Day	3-7
3.2.9	Activate Task	3-8
3.2.10	Suspend Task	3-8
3.2.11	Delete Task	3-9
3.2.12	Load Task	3-9
3.2.13	Breakpoint Trap	3-11

Chapter 4 - Sample Tasks

4.1	Sample Task #1	4-1
4.2	Sample Task #2	4-3

Chapter 5 - System Generation

5.1	General Description	5-1
5.2	Partition Table Definition	5-1
5.3	Task Table Definition	5-2
5.4	Parameter File Definition	5-5
5.5	Assembling The System	5-7
5.6	Linking The System	5-8
5.7	Sample System Generation	5-8

Appendices

A	Executive Request Summary	A-1
B	Device Dependent Information	B-1
C	Error Messages and Meaning	C-1
D	General Queuing Space	D-1
E	Device Table	E-1
F	Task Control Table	F-1
G	Resource Allocation Table	G-1
H	Partition Table	H-1
I	Sleep Queue	I-1
J	Partition Status Table	J-1
K	Operator's Console Command Data Block	K-1
L	Task Stack Frames	L-1
M	Software Configuration Parameters	M-1
N	Panic Dump Routine	N-1
O	Specification for 8K System	O-1
P	I/O Handler Interface	P-1

CHAPTER 1

GENERAL DESCRIPTION

1.1 INTRODUCTION

RSX11A is a small real-time multiprogramming operating system for Digital Equipment Corporation's PDP-11 family of computers. Major features include fixed priority scheduling, time dependent task initiation, and modest core residency requirements.

Support is provided for any number of tasks in either a core-only (core resident tasks) or core-disk environment (core and disk resident tasks).

RSX11A provides users with an operating system that is well suited for dedicated applications. Typically these include laboratory automation (single or multiple instruments), process control, and remote satellite processing.

No background capability is provided; however, any number of priority levels (one per task) are available and lower levels may be used to run nonessential or background tasks without interfering with higher priority foreground tasks.

RSX11A supports only tasks written in assembly language. System generation and actual task development must be carried out under other PDP-11 operating systems (i.e., DOS).

Tasks may be loaded into the system on-line via the operator's console task.

System generation options allow generation of systems requiring from 2 to 5K (K=1024) words of core memory.

Support is provided for RC, RF, and RK disks in a non-file structured format.

1.2 HARDWARE CONFIGURATIONS

RSX11A will run on any PDP-11 family processor. Support is provided for the extended arithmetic element (EAE), and I/O device handlers are provided for the following peripherals:

a) Teletype and Teletype compatible devices, including the LA-30, interfaced as the computer console terminal or via a KL11. The Teletype handler supports up to 16 Teletypes simultaneously (low-speed reader on ASR Teletypes is not supported).

b) PC11 high-speed paper tape reader.

c) PC11 high-speed paper tape punch.

d) RC11 fixed head disk controller.

e) RF11 fixed head disk controller.

f) RK11 disk cartridge controller.

g) AFC11 low level analog-to-digital convertor.

h) AD01-D high level analog-to-digital convertor.

i) UDC11 universal digital control unit.

j) LP11 line printer

The minimum hardware configuration required to run RSX11A is:

a) A PDP-11 family central processor.

b) 4K of read/write core memory.

c) ASR Teletype or PC11 high-speed paper tape reader.

d) KW11-L line frequency clock.

NOTE

Segmentation on the PDP-11/45 is not supported.

1.3 TERMINOLOGY

Tasks:

A task is defined as any set of program logic that, when executed, produces a desired result or accomplishes a specific job. Available core space is the only limit to the number of tasks that may be defined in RSX11A. Each task requires a minimum of eight or ten words of core resident storage. These words are used to define the task data base and must be allocated at system generation time.

The following information is stored in the task data base:

a) Periodic or Nonperiodic - A task may be periodic and executed at a regular frequency or may be nonperiodic and executed only on demand.

b) Immediate or Delayed Execution - A task may be executed immediately when the system is loaded or may be delayed until a specific request is made for its execution.

c) Core or Disk Resident - If the system is a core-only system, then tasks may only be core resident. If the system is a core-disk system, then tasks may be either core or disk resident. Core resident tasks always remain in core whereas disk resident tasks are loaded into core only when requested; the core used is then released when they terminate.

d) Privileged or Nonprivileged - A privileged task is given privileges that, if used improperly, may destroy any other task in the system or the system itself. For example the nonresident task loader is given the privilege to request I/O transfers into any part of memory. A nonprivileged task is given privileges that, if used improperly, can only destroy other tasks in its task group (see below for definition of task groups).

e) Task Group - The number assigned to the group to which a particular task belongs (see below).

f) Message Queue - A queue for receiving messages from other tasks.

g) Task Name - A 4-character ASCII name used to identify the task.

h) Execution Partition - The number assigned to the core partition from which the task executes.

i) Common Data Partition - The number assigned to the core partition used as a common area for communication with other tasks.

j) Disk Address and Device - A nonresident task in a core-disk system is stored on a particular device at a specific starting address. Only one access is required to load the task into the system.

k) Entry Address - The transfer address at which execution of the task begins.

Tasks consist of code, data, and stack segments. The entry address is also defined as the starting stack address (i.e., the value that will be loaded into register 6 when the task is executed). At least 9 stack words must immediately preceed the entry address. Since stack overflow is a serious matter and is difficult to detect, a standard task structure is recommended. This structure is not mandatory, but ensures that the task will probably destroy itself before it destroys another task or the system. The following structure is suggested:

```
Highest Core Address of Task

     Entry Point:      JMP Starting Point
                       .
                       .
                       .
                       Stack Segment
                       .
                       .
                       .
                       Data Segment
                       .
                       .
                       .
                       .
     Starting Point:   Code segment
                       .
                       .
                       .

Lowest Core Address of Task
```

Task Groups:

The concept of a task group is used in RSX11A to define a community of tasks that cooperate and share common resources. A task is given the privilege to affect the action of members of its own task group but not of other task groups. For example, a task cannot suspend the execution of another task unless that task is a member of the same task group.

The justification for task groups is not for protection alone. Groups provide the means whereby parallel tasks can be synchronized to access a common resource. Such resources include core tables, a disk file directory, or even a record within a file. The test and set executive requests are based on the group concept and provide a very general facility to accomplish synchronization.

Partitions:

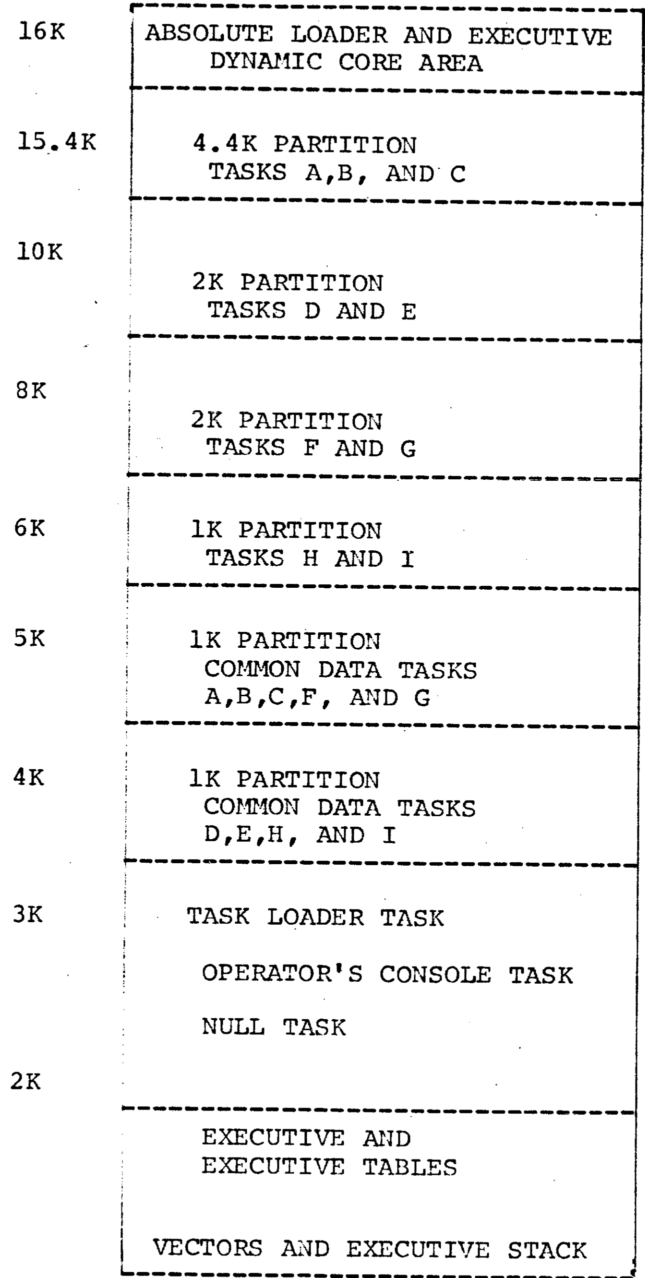
The core allocated for tasks under RSX11A is divided into fixed partitions. Each partition is defined at system generation time and its size and position never change. As many as 256 distinct partitions of any size may be defined. Partitions may not overlap each other or define core areas that are not present.

Tasks may execute from only one partition. This partition is termed the task's execution partition and is initialized when the task is executed. A task may also have a common data partition that is shared with other members of its task group (and may also be shared with other task groups). If a task does not have a common data partition, then its execution partition also serves this purpose.

Besides providing execution and data sharing areas, partitions also provide protection boundaries. A task may only request I/O transfers in or out of either its execution partition or its common partition. In addition, all executive request arguments must lie within one of the task's

partitions. Thus the executive uses the partition bounds for address checking. An address check error occurs when a task makes a request that specifies information that is not in either of its partitions.

The following schematic is a typical core memory layout for a 16K system that supports both core and disk resident tasks.



1.4 SYSTEM OVERVIEW

General:

An RSX11A system has the following features:

a) Multiprogramming of as many tasks as will fit into core at once.

b) Fixed priority scheduling.

c) Small core requirements.

d) Minimum executive overhead.

e) Support of a multi-user environment.

f) Thorough error checking facilities.

g) I/O efficiency with complete user control and flexibility.

h) Support of core-only or core-disk environments.

Multiprogramming:

Most real-time systems consist of a group of programs or tasks that run at varying times or frequencies and alternate between being compute bound and I/O bound. To efficiently use the central processor, these tasks cannot be run in series since the central processor will be poorly utilized during the periods that tasks are I/O bound. In addition, real-time tasks are time-dependent in one sense or another and in general cannot wait for a slow, less important I/O or compute bound task to finish before starting execution. Multiprogramming and some sort of priority scheme for scheduling the central processor are thus required.

Multiprogramming allows many tasks to be in some state of execution simultaneously. When one task cannot use all the available central processor time because it must wait for an I/O operation or is blocked by some other condition, the central processor can be switched to another task to make use of the available time. It is possible even in a multiprogramming environment that there may not be a task in an executable state to utilize this available time. While this is certainly possible, it is less likely to occur than in a uni-programming environment.

Fixed Priority Scheduling:

A priority scheme is needed to distinguish the relative importance of the various tasks in the system. It must be possible to interrupt the execution of a less important task to execute a critical real-time program. A fixed priority scheme has been chosen. Fixed priority meets the above requirement, is simple and adequate, and requires low scheduling overhead.

Core Residency:

The low core residency requirements of RSX11A were achieved mainly in two ways. The carefully designed modularity permits modules to function for many different purposes. Sections of code that are inseparable from a module (for efficiency reasons) but are not needed if a particular service is not included, may be conditionally assembled. This allows the generation of an absolute minimum system.

Modularity in the above context is defined as the separation of functions (or groups of inseparable functions) into individual software modules. Each module then has a well defined interface and function fo perform. This structure allows functions to be easily included or excluded from the system depending on whether or not they are required. System generation then can provide maximum configuration flexibility. The maintenance and extendability of such a system is also improved.

An attempt has been made to leave the system as open ended as possible so that users may generate a system that exactly fits their needs. Various tables are controlled by the user either by direct specification or by allocation of amount of table space. Examples of such tables include the task table, the generalized queuing space, and the core allocation table. Most system constants are also user specifiable. By constucting the system in this fashion a larger spectrum of users and applications can be served.

Minimum Overhead:

RSX11A was designed to require low overhead. This requirement manifests itself in many areas of the executive and precludes the use of any "sophisticated" heuristic algorithms that might otherwise be employed. Straight forward simple algorithms have been used in an effort to obtain maximum execution speed.

A good example can be found in the task scheduler. A major reason for choosing fixed priority scheduling was because of its low overhead requirements.

Multi-User Environment:

Many real-time systems must support several simultaneous user environments, leading to a need for some kind of protection between these environments. The ideal way to accomplish this is via the appropriate protection hardware. Unfortunately most PDP-11 processors do not have this hardware protection and therefore the burden of protection is placed solely on the software. One might argue that this is impossible in an environment where each user can essentially destroy the system if he so desires, but in a cooperative controlled environment acceptable protection can be implemented and enforced.

RSX11A presents a so called "friendly" environment where tasks must follow certain rules. If these rules are followed, then the greatest possible protection (within the limits of the software) is provided. The system guarantees

that no task can cause a system failure by passing bad parameters to the executive. For example, a task cannot transfer data into another task area by passing an improper address in an I/O request.

This type of protection, although not complete, works as long as users follow the rules and does provide aid to some extent in the debugging of new tasks.

Error Handling:

Too often systems have been designed that provide insufficient error checking facilities. Errors then manifest themselves as obscure system failures usually of a nonreproducible nature. While RSX11A cannot claim to check for all possible errors, it does provide an adequate error checking facility. The executive not only checks on the user environment, but also checks on itself at critical points that have proven to be major sources of errors. If the executive detects an error that has been made by the executive itself, a fatal system error is declared and the system is gracefully shut down. This occurs at one central point in the system and thus provides a convenient place to insert user code to initiate any application dependent action necessary.

Task errors may or may not be fatal to the task that caused the error. These errors include illegal executive requests, processor errors (i.e., odd word addresses or illegal instructions), and task stack overflow. If the task is enabled to handle its own errors, the occurrence of a task error causes a trap to the erring task's error routine. Otherwise, an erring task is disabled from further execution.

At the very least, adequate error checking enhances system integrity, facilitates new task development and checkout, and helps isolate hardware errors. It is well worth the core and execution time expended.

I/O Facility:

Real-time systems typically require large amounts of specialized I/O. Therefore the I/O facilities of such a system must be efficient and place as few restrictions as possible on user flexibility and user control. For these reasons RSX11A provides a central I/O facility that is entirely under task control. Buffering may occur to any depth without the threat of system invoked I/O waits. I/O transfers can be made in series or in parallel with the execution of the requesting task. Optionally a task may also specify an end action address that is to be trapped to when the requested I/O operation has been completed. The end action routine is executed at the software priority of the requesting task and not at the hardware interrupt level of the device. All systems services are available from the end action routine.

All I/O requests are queued by the priority of the requesting task. Device handlers then empty their request queues at the

rate of the individual devices. The only limit as to the number of I/O operations that may be queued is the amount of queuing space that has been allocated for this purpose.

To summarize, the I/O facility of RSX11A has attempted to provide centralized routines with maximum possible user control and flexibility. No assumptions have been made about buffering as this is left to the individual tasks. Tasks may synchronize I/O transfers in any way they desire; however, facilities have been provided for this purpose.

Core and Disk Residency:

RSX11A may be generated as a core-only system or as a core-disk system. The core-only system supports only core resident tasks; however, mass storage device handlers may also be included for storage of data and/or task overlays on a mass storage device.

The core-disk system supports both core resident and disk resident tasks. Disk resident tasks are brought into core whenever requested and upon exit release the core they were occupying. Such tasks are designated to execute out of a particular partition. More than one disk resident task may execute out of a single partition. If multiple requests are received for a particular partition that is currently busy, they are queued by the priority of the individual tasks requesting the partition. When the partition is freed the highest priority waiting task is given ownership of the partition. The task is then swapped into core and executed. Core-disk systems must contain at least one mass storage device handler. If more than one mass storage device is present, then disk resident tasks may be stored on either or both devices.

1.5 MANUAL CONVENTIONS

Several conventions are used throughout this manual to avoid confusion. In text, octal and decimal numbers are distinguished by the presence or absence of a leading '#'. Octal numbers are preceded by '#' whereas decimal numbers are not. In the context of an assembly language example, however, the assembly rules apply for octal and decimal numbers.

The PDP-11 core memory is byte addressable and confusion often arises over the addressing of the Ith item of an array. The convention adopted herein is to use the word "index" only when the value of the index can be applied directly to address the Ith item of an array. The array so addressed may be a byte, word, or multiword array. The word "number", on the other hand, is used to refer to the Ith item of an array, but not to its address. For example the fifth member of a word array is referred to as "number" 5 and "index" #12.

All external address symbols are appended with a dollar sign ($). Equated external symbols are preceded by a dollar sign and executive request macro names have a dollar sign as their second character. These are exactly the same conventions that are used in the RSX11A software itself.

CHAPTER 2

EXECUTIVE COMMUNICATION

2.1 GENERAL DESCRIPTION

A user communicates with the executive via programmed operators. These operators are termed Executive Requests and are referred to as ER's. ER's provide a set of centralized services that are generally applicable to all real-time systems.

Services may be classified as serially reusable or reentrant; however, in either case they are shared by all users of the system. The difference between serially reusable and reentrant services is that they run on different stacks. Serially reusable services run on the system stack and are noninterruptible. Their execution will be completed before a context switch may take place. Noninterruptible does not mean that they run with interrupts locked out. On the contrary, interrupts are left on as much as possible. Reentrant services run on the user stack and therefore may require additional stack space. These services are interruptible and a context switch may take place at any time during their execution. As many services as possible are reentrant in order to provide maximum response to dynamic changes in the system.

2.1.1 Summary of Executive Requests

Executive requests are classified into ten classes. Each class represents a logically complete set of functions to afford tasks maximum flexibility. Following is a synopsis of the classes and services included therein.

1) Task Termination:

    a) Terminate task execution.

    b) Delete task from system.

2) Timer:

    a) Request timed wait.

    b) Request timed interrupt.

    c) Cancel timed interrupt request.

    d) Request time of day.

    e) Request date.

3) Task Initiation:

    a) Request task execution.

b) Request synchronous periodic task execution.

c) Request asynchronous periodic task execution.

d) Request task execution at time of day.

4) Task Synchronization:

a) Suspend task execution.

b) Activate task execution.

c) Test and set task group lock return immediate.

d) Test and set task group lock wait.

e) Reset task group lock.

5) Intertask Communication:

a) Send message to task.

b) Receive message from task.

6) Input/Output:

a) Request I/O operation.

7) End Action Control:

a) End action wait.

b) End action return.

8) List Manipulation:

a) Define list.

b) Remove entry from list.

c) Make entry in list.

9) Dynamic Storage:

a) Request buffer block.

b) Release buffer block.

10) Miscellaneous:

   a) Set error trap address.

   b) Set alternate stack address.

   c) Set TRAP trap address.

2.1.2 Executive Request Calling Sequence

One generalized calling sequence is used to call the executive. All parameters pertinent to the requested operation are passed to the executive via the task stack. The user's registers R0 to R5 are always preserved across the call.

Most ER's require one or two parameters; however, some require none. All parameters are validity checked to ensure that the user does not violate system rules or cause the executive to crash because of an illegal parameter (i.e., a nonexistent memory address).

Several ER's have skip and nonskip returns depending upon the availability of resources. The skip return is considered to be the start of an instruction sequence. The nonskip return, however, is considered to contain a "place to go" address. This enables an effective "jump" to any address in memory for a nonskip return.

ER's are actually implemented via the EMT instruction. The following instruction sequences illustrate the manner in which a user calls the executive for zero-, one-, and two-parameter ER's:

Zero-parameter ER

```
        EMT     Function Code
```

One-parameter ER

```
        MOV     #P1,-(SP)
        EMT     Function Code
```

Two-parameter ER

```
        MOV     #P2,-(SP)
        MOV     #P1,-(SP)
        EMT     Function Code
```

If the ER has a skip return, the word immediately following the EMT is interpreted as a "place to go" address for the nonskip case. This sequence is as follows for a two-parameter ER:

```
        MOV     #P2,-(SP)
        MOV     #P1,-(SP)
        EMT     Function Code
        .WORD   Place to Go Address
```

A macro definition is supplied for all ER's. Macro arguments are always taken as values and never as registers. The macro definitions for ER's are distinguishable by the existence of a dollar sign ($) as the second character in the macro name. The general form of the macro call is:

```
M$NAME  P1,P2,Place to Go
```

The above macro call represents a two-parameter ER with a skip return.

The following subsections define each ER in detail and present the macro name and calling sequence.

Performance data is included for each ER. The sizes given assume maximum size and may be smaller if certain configuration parameters are selected. All ER's are system generatable with the exception of Terminate.

ER's that are group dependent may be executed without regard to group by privileged tasks. For example, a privileged task may request the execution of any other task in the system.

2.2 DETAILED ER DESCRIPTIONS

2.2.1 Task Termination

2.2.1.1 Terminate Task Execution

Functional Description:

Terminate the execution of the requesting task.

If the tasks's execution request flag is set when this ER is executed, one or more execution requests were received for the task while it was already in execution. The flag is cleared, the task's stack is reinitialized and the task is immediately restarted from it's transfer address.

In core-disk systems the terminate ER also results in a release of the task's execution partition, if the task is a non-resident task. For this case, the partition wait queue is examined to see if any tasks are waiting to be loaded. If no tasks are waiting, the partition status is simply set to not busy. However, if there is a task waiting, it is removed from the partition wait queue and inserted in the task loader queue. An execution request is then placed for the task loader.

If the terminating task is asynchronous periodic, then its next execution time is computed as the current time of day plus the task's period. The task is then inserted in the sleep queue so its execution will be requested at the computed time.

The final step in terminating a task is to set the task inactive and to place a schedule request at the priority of the terminating task.

Request syntax:

The macro calling sequence is:

```
T$ERM
```

The assembly language generated from the macro expansion is:

```
EMT     0
```

Parameters:

No parameters are required for this ER.

Error Conditions:

Outstanding I/O requests pending.

Outstanding end action requests pending.

Performance Data:

Runs on system stack.

Required ER and not generatable.

Core requirements: 127 words.

Additional routines required: BILDS, QUEUE, LOCKS, RQLCB, REQSB, and SQUE.

User stack requirements: 19 words.

Program example:

Terminate the execution of the current task.

```
T$ERM
```

2.2.1.2 Delete Task From System

Functional Description:

Delete from the system any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be deleted.

If the specified task is currently in execution, then the requesting task's stack is not cleared and a nonskip return is executed. Otherwise all pending requests for the specified task are removed from the system. The corresponding task table entry is declared undefined thus deleting the task from the system.

The requesting task's stack is cleared and a return is executed.

Request Syntax:

The macro calling sequence is:

```
D$ELT TNAME,BUSY
```

The assembly language generated from the macro expansion is:

```
MOV #TNAME,-(SP)
EMT 2
.WORD BUSY
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be deleted.

BUSY: The nonskip "place to go" address if the task is currently in execution.

Error Conditions:

Address check on task name.

Undefined task or illegal task group if requesting task is not priviledged.

Performance Data:

Runs on system stack.

Core requirements: 41 words.

Additional routines required: GETID, RQLCB, and RMTSK.

User stack requirements: 19 words.

Program Example:

Delete the task CATR.

```
        D$ELT TNAME

TNAME:  .ASCII /CA/ ;TASK NAME
        .ASCII /TR/ ;
```

2.2.2 Timer

2.2.2.1 Request Timed Wait

Functional Description:

Delay the execution of the requesting task for a specified interval of time.

The next execution time for the task is computed as the current time of day plus the specified interval. The task is inserted into the sleep queue so it will resume execution after the interval has elapsed. The task is blocked and a schedule request is placed at the priority of the requesting task.

If the request cannot be accepted because there is no room in the sleep queue, the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is set up for when the task comes out of the sleep queue.

Request Syntax:

The macro calling sequence is:

```
W$AIT   INT,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #INT,-(SP)
EMT     3
.WORD   NORM
```

Parameters:

INT: The address of a two-word time delay expressed in line frequency units.

NORM: The nonskip "place to go" address if there is no room in the sleep queue.

Error Conditions:

Address check on time delay interval.

Performance Data:

Runs on system stack.

Core requirements: 33 words.

Additional routines required: ACHCK, REQSB, RQLCB, and SQUE.

User stack requirements: 19 words.

Program Example:

Delay the execution of the current task for one second.

```
        W$AIT   INT,NORM

INT:    .WORD   60.             ;LOW ORDER TIME
        .WORD   0               ;HIGH ORDER TIME

NORM:   Start of no room coding.
```

2.2.2.2 Request Timed Interrupt

Functional Description:

Request an interrupt trap to an end action address after a specified interval of time has elapsed.

The code executed at the end action address is executed at the software priority of the task and not at the hardware level of the clock interrupt. A return to the interrupted place in the task may be accomplished with an end action return ER.

The time of day that the trap is to occur is computed as the current time of day plus the specified interval. An entry specifying the computed time of day is placed in the sleep queue. When the current time of day becomes equal to that time, a trap to the end action address will occur.

If the request cannot be accepted because there is no room in the sleep queue, the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$TINT  INT,ENDA,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #ENDA,-(SP)
MOV     #INT,-(SP)
EMT     4
.WORD   NORM
```

Parameters:

ENDA: The end action "place to go" address to be trapped to after the specified time interval has elapsed.

INT: The address of a two-word time interval expressed in line frequency units.

NORM: The nonskip "place to go" address if there is no room in the sleep queue.

Error Conditions:

Address check on time interval.

Address check on end action address.

Performance Data:

Runs on system stack.

Core requirements: 40 words.

Additional routines required: ACKCK, RQLCB and SQUE.

User stack requirements: 19 words.

Program example:

Interrupt the current task after 100 milliseconds have elapsed.

```
        R$TINT  INT,ENDA,NORM

INT:    .WORD   6               ;LOW ORDER TIME
        .WORD   0               ;HIGH ORDER TIME
```

ENDA: Start of routine to be trapped to.

NORM: Start of no room coding.

2.2.2.3 Cancel Timed Interrupt Request

Functional Description:

Cancel all timed interrupt requests for the requesting task.

The sleep queue is searched for timed interrupt requests for the requesting task. All such entries are removed from the sleep queue and a return to the requesting task is executed.

Request Syntax:

The macro calling sequence is:

```
        C$TINT
```

The assembly language generated from the macro expansion is:

```
        EMT     5
```

Parameters:

None.

Error Conditions:

None.

Performance Data:

Runs on system stack.

Core requirements: 7 words.

Additional routines required: SQUE.

User stack requirements: 19 words.

Program Example:

Cancel all timed interrupt requests for the current task.

```
C$TINT
```

2.2.2.4 Request Time of Day

Functional Description:

Request the current time of day.

The time of day is returned to the requesting task, expressed as clock ticks past midnight, on the top two words of the task's stack.

Request Syntax:

The macro calling sequence is:

```
R$TOD
```

The assembly language generated from the macro expansion is:

```
CMP     -(SP),-(SP)
EMT     6
```

Parameters:

None.

Error Conditions:

None.

Performance Data:

Runs on user stack.

Core requirements: 10 words.

Additional routines required: None.

User stack requirements: 19 words.

Program Example:

Obtain the current time of day.

```
R$TOD
MOV     (SP)+,R0            ;LOW ORDER TIME
MOV     (SP)+,R1            ;HIGH ORDER TIME
```

2.2.2.5 Request Date

Functional Description:

Request the current date.

The date is returned to the requesting task, expressed as a Julian date relative to the year 1972, on the top two words of the task's stack.

Request Syntax:

The macro calling sequence is:

```
        R$DATE
```

The assembly language generated from the macro expansion is:

```
        CMP     -(SP),-(SP)
        EMT     7
```

Parameters:

None.

Error Conditions:

None.

Performance Data:

Runs on user stack.

Core requirements: 10 words.

Additional routines required: None.

User stack requirements: 19 words.

Program Example:

Obtain the current date:

```
        R$DATE
        MOV     (SP)+,R0            ;JULIAN DAY
        MOV     (SP)+,R1            ;YEAR RELATIVE TO
                                    ;1972
```

2.2.3 Task Initiation

2.2.3.1 Request Task Execution

Functional Description:

Request the execution of any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be executed.

If the request cannot be accepted because no queuing space is available or because the requested task is disabled from executing, then the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$QEX   TNAME,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #TNAME,-(SP)
EMT     10
.WORD   NORM
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be executed.

NORM: The nonskip "place to go" address if no queuing space is available or the requested task is disabled.

Error Conditions:

Address check on task name.

Undefined task or illegal task group if requesting task is not priviledged.

Performance Data:

Runs on system stack.

Core requirements: 27 words.

Additional routines required: GETID, REQSB, and SQUE.

User stack requirements: 19 words.

Program example:

Request the execution of the task ONCE.

```
        R$QEX   TNAME,NORM

TNAME:  .ASCII  /ON/            ;TASK NAME
        .ASCII  /CE/            ;

NORM:   Start of no room coding.
```

2.2.3.2 Request Synchronous Periodic Task Execution

Functional Description:

Request the synchronous periodic execution of any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be synchronously executed and the interval of periodicity.

Synchronous periodic execution of a task provides an execution request for that task each time the specified interval of time has elasped. An execution request is immediately placed to execute the specified task. The next request time is computed as the current time of day plus the specified time interval, and an entry is placed in the sleep queue. When the current time of day becomes equal to that time, a request will be placed to execute the specified task and another entry placed back in the sleep queue. Another request will then be placed at that time of day.

If the request cannot be accepted because the requested task is disabled from executing or there is no room in the sleep queue, then the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$QSX   TNAME,INT,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #INT,-(SP)
MOV     #TNAME,-(SP)
EMT     11
.WORD   NORM
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be synchronously executed.

INT: The address of a two-word periodic interval in line frequency units.

NORM: The nonskip "place to go" address if there is no room in the sleep queue or the requested task is disabled.

Error Conditions:

Address check on task name.

Address check on periodic interval.

Undefined task or illegal task group if requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 57 words.

Additional routines required: GETID, RQLCB, RMTSK, SQUE, and REQSB.

User stack requirements: 19 words.

Program Example:

Request the synchronous periodic execution of the task REPT with a period of 100 milliseconds.

```
        R$QSX   TNAME,INT,NORM

TNAME:  .ASCII  /RE/            ;TASK NAME
        .ASCII  /PT/            ;

INT:    .WORD   6               ;LOW ORDER TIME
        .WORD   0               ;HIGH ORDER TIME

NORM:   Start of no room coding.
```

2.2.3.3 Request Asynchronous Periodic Task Execution

Functional Description:

Request the asynchronous periodic execution of any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be asynchronously executed and the interval of periodicity.

Asynchronous periodic execution of a task provides an execution request for that task after the specified interval of time has elapsed from the termination of the task. In other words, the period of execution is specified as the time interval from the termination of one execution to the start of the next.

An execution request is immediately placed to execute the specified task. When it terminates, its next execution time will be computed as the current time of day plus the specified interval and an entry will be placed in the sleep queue. When the current time of day becomes equal to that time, another request will be placed to execute the task.

If the request cannot be accepted because the requested task is disabled from executing or there is no room in the sleep queue, then the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
        R$QAX   TNAME,INT,NORM
```

The assembly language generated from the macro expansion is:

```
        MOV     #INT,-(SP)
        MOV     #TNAME,-(SP)
        EMT     12
        .WORD   NORM
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be asynchronously executed.

INT: The address of a two-word periodic interval in line frequency units.

NORM: The nonskip "place to go" address if there is no room in the sleep queue or if the requested task is disabled.

Error Conditions:

Address check on task name.

Address check on periodic interval.

Undefined task or illegal task group if requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 37 words.

Additional routines required: ACKCK, GETID, RMTSK, and REQSB.

User stack requirements: 19 words.

Program Example:

Request the asynchronous periodic execution of the task FAST with a period of 200 milliseconds.

```
        R$QAX   TNAME,INT,NORM

TNAME:  .ASCII  /FA/            ;TASK NAME
        .ASCII  /ST/            ;

INT:    .WORD   12.             ;LOW ORDER TIME
        .WORD   0               ;HIGH ORDER TIME

NORM:   Start of no room coding.
```

2.2.3.4 Request Task Execution at Time of Day

Functional Description:

Request the execution of any task in the requestor's task group at a specific time of day. The requesting task specifies the ASCII name of the task and the time of day that it is to be executed.

The time of day is expressed in line frequency units past midnight and is used directly to make an entry in the sleep queue. When the current time of day becomes equal to that time, an execution request will be placed for the specified task.

If the request cannot be accepted because the requested task is disabled from execution or if there is no room in the sleep queue, then the requesting task's stack is not cleared and a nonskip return is executed. Otherwise, the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$QTX   TNAME,TOD,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #TOD,-(SP)
MOV     #TNAME,-(SP)
EMT     13
.WORD   NORM
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be executed at a specific time of day.

TOD: The address of a two-word time of day expressed in line frequency units.

NORM: The nonskip "place to go" address if there is no room in the sleep queue or the requested task is disabled.

Error Conditions:

Address check on task name.

Address check on time of day.

Undefined task or illegal task group if requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 40 words.

Additional routines required: ACHCK, GETID, RQLCB, and SQUE.

User stack requirements: 19 words.

Program Example:

Request the execution of the task FLOP at 8:30 AM.

```
        R$QTX   TNAME,TOD,NORM

TNAME:  .ASCII  /FL/            ;TASK NAME
        .ASCII  /OP/            ;

TOD:    .WORD   792.            ;LOW ORDER TIME
        .WORD   28.             ;HIGH ORDER TIME

NORM:   Start of no room coding.
```

2.2.4 Task Synchronization

2.2.4.1 Suspend Task

Functional Description:

Suspend the execution of any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be suspended.

The operation of suspending a task simply sets a blocking bit in the specified task's status word. This bit, when set, blocks the execution of the task and makes it unrunnable. The task will remain unrunnable until the bit is cleared (see Activate Task below). The setting and clearing of this bit is completely independent of any other state the task may already be in.

Suspending a task does not prevent execution requests from being accepted; even task loading (disk system only) can take place. The task, however, will not execute one instruction until it is reactivated by clearing the blocking bit.

The requesting task's stack is cleared and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        S$PND   TNAME
```

The assembly language generated from the macro expansion is:

```
        MOV     #TNAME,-(SP)
        EMT     14
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be suspended.

Error Conditions:

Address check on the task name.

Undefined task or illegal task group if the requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 9 words.

Additional routines required: GETID and REQSB.

User stack requirements: 19 words.

Program Example:

Suspend the execution of the task STOP.

```
        S$PND   TNAME

TNAME:  .ASCII  /ST/            ;TASK NAME
        .ASCII  /OP/            ;
```

2.2.4.2 Activate Task

Functional Description:

Activate the execution of any task in the requestor's task group. The requesting task specifies the ASCII name of the task to be activated.

The operation of activating a task simply clears the blocking bit that is set by the suspend task ER. Removing this blocking condition makes the specified task immediately eligible to run on the processor provided it is already in execution and not blocked by any other blocking condition.

The requesting task's stack is cleared and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        A$CTV   TNAME
```

The assembly language generated from the macro expansion is:

```
        MOV     #TNAME,-(SP)
        EMT     15
```

Parameters:

TNAME: The address of the 4-character ASCII name of the task to be activated.

Error Conditions:

Address check on task name.

Undefined task or illegal task group if requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 8 words.

Additional routines required: GETID and REQSB.

User stack requirements: 19 words.

Program Example:

Activate the execution of the task REAL.

```
        A$CT    TNAME

TNAME:  .ASCII  /RE/            ;TASK NAME
        .ASCII  /AL/            ;
```

2.2.4.3 Test and Set Task Group Lock Return Immediate

Functional Description:

Test and set a task group lock and return control immediately regardless of whether the operation is successful. The requesting task specifies an 8-bit key which identifies the lock to be tested and set.

The use of this and other group lock ER's may be best understood if group locks themselves are explained. Task group locks provide the means whereby a task group may share up to 256 group specified resources. Each resource is assigned a unique 8-bit key. Example resources might be a core buffer pool in a common area or a file on a disk. Nonsimultaneous access to a shared resource (one that has been assigned a key) can be guaranteed by using the task group lock ER's (see also Test and Set Wait and Reset ER's).

A lock can be owned by only one task at a time. Thus when a lock is set, it is considered to be owned and no other task may become owner until the current owner resets the lock. Tasks may only retain ownership of a lock during their execution. Executing a terminate ER results in the resetting of all locks that the task owns.

The relationship between a particular key and a resource may be decided ahead of time or tasks may dynamically specify the

correspondence at execution time. The system places no definition upon a key. The definition is the task group's alone. The usage of the task group lock ER's to ensure nonsimultaneous access to a shared device is very simple. Each time a member of the task group wishes to gain access to a shared resource, it specifies the corresponding key and executes one of the test and set ER's. If the operation is successful, then the task has gained ownership of the designated lock and may access the resource. When the task has finished its operation, a reset lock ER is executed, thus making the lock available for ownership by other tasks in the group. If, on the other hand, the operation was not successful, then the lock is already owned by another task and the requesting task may not safely use the shared resource.

The understanding of the group lock ER's can be increased further if an important distinction is drawn. The group lock ER's do not themselves prevent simultaneous access to shared resources. Rather they provide the means necessary whereby tasks themselves may guarantee nonsimultaneous access to a shared resource.

The test and set task group lock return immediate ER enables a task to test and set a particular lock and regain control immediately. If the operation is successful, then the requesting task's stack is cleared and a skip return is executed. Otherwise the requesting task's stack is not cleared and a nonskip return is executed.

Request Syntax:

The macro calling sequence is:

```
T$SETI  KEY,BUSY
```

The assembly language generated from the macro expansion is:

```
MOV     #KEY,-(SP)
EMT     20
.WORD   BUSY
```

Parameters:

KEY: An 8-bit group lock identifier.

BUSY: The nonskip "place to go" address if the operation is unsuccessful.

Error Conditions:

None.

Performance Data:

Runs on system stack.

Core requirements: 1 word.

Additional routines required: LOCKS.

User stack requirements: 19 words.

Program Example:

Test and set the task group lock whose key is 99.

```
        T$SETI  99.,BUSY

BUSY:   Start of lock busy coding.
```

2.2.4.4 Test and Set Task Group Lock Wait

Functional Description:

Test and set a task group lock and suspend the requestor's execution if the operation is unsuccessful. The requesting task specifies an 8-bit key which identifies the lock to be tested and set.

The specified group lock is tested and set. If the operation is successful (task has become owner of lock), then the task stack is cleared and a skip return is executed. Otherwise an attempt is made to insert the requesting task into the group lock wait queue for the specified lock. If there is no room in the queue, the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is set up for when the task becomes owner of the lock.

Request Syntax:

The macro calling sequence is:

```
        T$SETW  KEY,NORM
```

The assembly language generated form the macro expansion is:

```
        MOV     #KEY,-(SP)
        EMT     21
        .WORD   NORM
```

Parameters:

KEY: An 8-bit group lock identifier.

NORM: The nonskip "place to go" address if no queuing space is available.

Error Conditions:

None.

Performance Data:

Runs on system stack.

Core requirements: 6 words.

Additional routines required: LOCKS.

User stack requirements: 19 words.

Program Example:

Test and set the task group lock whose key is 45.

```
        T$SETW  45.,NORM

NORM:   Start of no room coding.
```

2.2.4.5 Reset Task Group Lock

Functional Description:

Reset a task group lock. The requesting task specifies an 8-bit key which identifies the lock to be reset.

The specified lock is reset. If there are any tasks waiting to become owner of the lock, then the highest priority (first) task is removed from the corresponding group lock wait queue. This task is given ownership of the lock and is reactivated by clearing its resource wait bit.

The requesting task's stack is cleared and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        R$SET   KEY
```

The assembly language generated from the macro expansion is:

```
        MOV     #KEY,-(SP)
        EMT     22
```

Parameters:

KEY: An 8-bit group lock identifier.

Error Conditions:

None.

Performance Data:

Runs on system stack.

Core requirements are: 10 words.

Additional routines required: LOCKS.

User stack requirements: 19 words.

Program Example:

Reset the task group lock whose key is 29.

```
R$SET   29.
```

2.2.5 Intertask Communication

2.2.5.1 Send Message To Task

Functional Description:

Send a message to any task in the requestor's task group. The requesting task specifies a two-word message and the ASCII name of the task that is to receive it.

An attempt is made to insert the message in the specified task's message queue. If sufficient queuing space is not available, the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the requesting task's stack is cleared and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
S$END   TNAME,MPKT,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #MPKT,-(SP)
MOV     #TNAME,-(SP)
EMT     23
.WORD   NORM
```

Parameters:

MPKT: The address of a two-word message.

TNAME: The address of the 4-character ASCII name of the task that is to receive the message.

NORM: The nonskip "place to go" address if no queuing space is available.

Error Conditions:

Address check on message.

Address check on task name.

Undefined task or illegal task group if requesting task is not privileged.

Performance Data:

Runs on system stack.

Core requirements: 23 words.

Additional routines required: ACHCK, GETID, and QUEUE.

User stack requirements: 19 words.

Program Example:

Send a message to the task GARB.

```
        S$ENT   TNAME,MPKT,NORM

TNAME:  .ASCII  /GA/            ;TASK NAME
        .ASCII  /RB/            ;

MPKT:   .WORD   55.             ;TWO WORD
        .WORD   46.             ;MESSAGE

NORM:  Start of no room coding.
```

2.2.5.2 Receive Message From Task

Functional Description:

Remove a message from the requestor's message queue.

An attempt is made to remove a message from the requesting task's message queue. If there are no entries in the queue, the requesting task's stack is cleared and a nonskip return is executed. Otherwise the first message is removed and placed on the top of the requesting task's stack and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
        R$CEIV  NONE
```

The assembly language generated from the macro exapnsion is:

```
        CMP     -(SP),-(SP)
        EMT     24
        .WORD   NONE
```

Parameters:

NONE: The nonskip "place to go" address if there are no messages in the task's message queue.

Error Conditions:

None.

Performance Data:

Runs on user stack.

Core requirements: 17 words.

Additional routines required: QUEUE.

User stack requirements: 14 words.

Program Example:

Remove a message from current task's message queue.

```
        R$CEIV  NONE
        MOV     (SP)+,R0        ;FIRST WORD
        MOV     (SP)+,R1        ;SECOND WORD
```

NONE: Start of empty queue coding.

2.2.6 Input/Output

2.2.6.1 Request I/O Operation

Functional Description:

Request an I/O operation (i.e., a device control function or transfer of information between core memory and a peripheral device). One general interface is provided for all devices. This interface is termed an I/O packet, the general form of which is nine words. Six words are device independent and the remaining three are device dependent. All device dependent information is given in Appendix B.

An I/O packet specifies control information, the function to be performed, the I/O channel on which the function is to be performed, and an item count and buffer address if the specified function involves a data transfer. Depending on the device, additional device dependent information may also be required.

I/O operations may be processed in series or in parallel with the execution of the requesting task. This is termed with or without "hold" respectively. An I/O operation that is to be performed in series with the execution of the requesting task results in the I/O operation being queued and the task's execution suspended until the operation has been completed. I/O operations that are to be processed in parallel with the execution of the requesting task result in the I/O operation being queued and control returned immediately to the requesting task. In either case, the status byte in the I/O packet will remain busy until the I/O operation is actually completed (see general I/O packet format below).

An end action "place to go" address may be specified for all I/O operations. When the I/O operation is complete, the execution of the requesting task is trapped to the end action

address. The code executed at the end action address is termed end action coding. This code is executed at the software priority of the task and not at the interrupt level of the specified device. Control is returned to the interrupted place in the task via the end action return ER (see 2.2.7.2).

Different end action "place to go" addresses may be specified with each I/O request. All end action requests, as they occur, are processed "last-in-first-out", and therefore end action coding may itself be interrupted to process other end action requests. Multiple I/O requests that specify the same end action address must be processed by reentrant end action coding.

End action and its purpose may be more fully understood if the sequence of operations that occur is described. The first operation that must occur is for a task to make one or more I/O requests specifying an end action address. As each of these I/O requests is completed, a new program state is stored on the requesting task's stack. This program state consists of a set of register values, the end action "place to go" address, and a processor status word of zero. Register R0 is loaded with the address of the original I/O packet, R1 with the final software status of the operation, and R2 with the untransferred item count. Registers R3 thru R5 are set to zero. When the task becomes the highest priority task that is runnable, the first set of context is loaded and execution begins in the task at the end action address. The task returns control to the next set of context by executing an end action return ER. This ER effectively discards the current set of context and causes the next set to be loaded.

Nine stack words must be allocated on the task stack for each nested level of end action that is desired. If the stack is also used for temporary operands in the end action coding, this space must also be allocated. The nine words are necessary for each context push. Before a context push is executed, the task stack is address checked to see if it would underflow either the task's execution or common partition. If stack overflow would occur, then a switch is made to the task's alternate stack and a trap to the error routine is set up.

Before an end action return ER is executed, the task must ensure that the task stack is cleared of all temporary operands that were placed on the stack during end action coding (i.e., the task stack must point exactly to the same address that it did when the end action coding was entered). If this restriction is not followed, unpredictable results will occur.

It should be noted that I/O hold and end action are distinct functions and are not dependent upon each other in any way. I/O hold blocks the execution of the requesting task until that specific I/O request is completed. End action does not block the execution of the requesting task and may occur at any time the task is in a runnable state (not blocked in any way). In order to understand the distinction an example is in order. Suppose that four I/O requests were made that

specified only end action and a fifth request was made that specified both end action and hold. None of the end action coding could actually be executed until the task becomes runnable at the completion of the I/O request specifying hold. Assuming that the four end action requests were all completed before the hold request, each set of end action coding would be executed one after another. Each end action return ER would return control to the next set of end action coding. The fifth end action return ER would return control to the interrupted place in the task.

Now that an understanding of how end action works has been established, the questions may be asked: why end action; how does one use it? Why may be answered very simply; it enables a task to implement parallel processing of I/O devices. In other words, each transfer propagates the next transfer by trapping the task whenever a transfer has been completed.

The answer to the second question of how does one use it requires examples. Suppose that one task is to handle input from more than one terminal simultaneously and that it is undesirable to solicit input from each terminal without hold and then poll the I/O packets periodically to see if any lines of input have been typed. Rather input is solicited from each terminal without hold but with end action. An end action wait ER (see subsection 2.2.7.1) is then executed. The end action wait ER causes the execution of the task to be blocked until one of its end action requests occurs (i.e., one of the terminals finishes typing a line of input). The task is then unblocked and its execution is trapped to the end action address. The end action coding queues the line of input for processing, solicits more input with end action, and executes an end action return ER. Control is then regained in normal coding whereupon the task processes the line of input and executes another end action wait ER to wait for the next line of input.

As a second example, suppose that multiple level buffering is desired on output to a device. One way that this could be accomplished is to have a pool of I/O packets and buffers. Each time a buffer full of information is ready to be transferred, the I/O packet pool is searched for a packet that is not busy. The I/O packet is then initialized and an I/O request is made without hold. This could continue until all I/O packets are busy, since multilevel I/O queuing is provided in the executive. When this condition occurs, one of two alternatives can be executed. An I/O request to the same device for a transfer of zero items could be executed with hold. Executing this alternative would cause the execution of the task to be blocked until all I/O packets were not busy. This would accomplish the desired result, but would defeat the purpose of multilevel buffering when all I/O packets were busy. The second alternative would be to execute a timed wait ER with a time interval equal to the time it takes to unload one buffer. When control is regained, another search of the I/O packet pool is conducted. This continues until one of the I/O requests is complete. The desired multilevel buffering is accomplished in all conditions, but unnecessary execution time is expended to search the packets periodically until one becomes free. A second way to handle multilevel buffering is to use end

action. As each I/O request is executed, end action is specified. As each end action request is processed, the I/O packet and buffer are returned to a free pool to be used again. If the situation is encountered in which no I/O packets or buffers are available, an end action wait ER is executed. This guarantees that the task will only regain control when a buffer is free. No unnecessary execution time is expended and the desired multilevel buffering is accomplished. This example can be carried to as many devices simultaneously as desired.

One problem should be recognized and fully understood before attempting multilevel end action (i.e., more than one simultaneous I/O request that specifies the same end action address). The end action coding must be reentrant with respect to both itself and normal task coding. The most common reentrancy problem in end action coding is that of list manipulation. For this reason several ER's have been provided to ease this problem. These ER's are explained in subsections 2.2.8.1, 2.2.8.2, and 2.2.8.3.

It should be noted that a task may execute any ER from end action coding, since the task is executing at its software priority level with a complete set of context. Care should be exercised, however, to ensure that the exit from end action coding is via the end action return ER.

I/O packets, once verified, are not copied into auxiliary storage in the executive; therefore extreme caution should be exercised so as not to modify an I/O packet before the operation is complete. Ignoring this restriction will cause unpredictable results.

An I/O request, after being verified, may or may not be accepted depending on whether queuing space is available. If a request cannot be accepted, the requesting task's stack is not cleared and a nonskip return is executed. Otherwise the I/O request is queued, the requesting task's stack is cleared, and a skip return is setup. If I/O hold was specified, the task is placed in I/O wait and a schedule request is placed at the priority of the requesting task. Otherwise the skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$QIO   PKT,NORM
```

The assembly language generated from the macro expansion is:

```
MOV     #PKT,-(SP)
EMT     30
.WORD   NORM
```

Parameters:

PKT: The address of a 9-word I/O packet. Not all I/O packets require nine words.

NORM: The nonskip "place to go" address if no queuing space is available.

Actual packet formats are given in Appendix B. The general form of the 9-word I/O packet is as follows:

Word 0 (PKT-4)

Bits 15 to 0 - The end action address if end action is specified. If end action is not specified, this word does not have to be present.

Word 1 (PKT-2)

Bits 15 to 0 - The final untransferred item count. This word is set at the completion of an I/O transfer. For control functions this word is meaningless but must be present.

Word 2 (PKT)

Bits 15 to 8 - The current packet status. This byte reflects the current state of the I/O packet and is set by the various I/O routines. The status of a packet may be:

0 = The packet is not busy and the I/O operation has been completed without errors.

1 = The packet is busy and the I/O operation has not been completed.

2 = The packet is not busy and the I/O operation ended in error. Words 7 and 8 of the I/O packet contain the final hardware status for those devices that have more than one error state.

3 = The packet is not busy and the I/O operation timed out before being completed.

4 = The packet is not busy and the specified device is down.

Bit 7 - I/O hold specification.

0 = The I/O operation is to be performed in parallel with the execution of the requesting task.

1 = The I/O operation is to be performed in series with the execution of the requesting task.

Bit 6 - End action specification.

0 = No end action address is provided in word 0.

1 = Word 0 contains an end action "place to go" address that is trapped to when the I/O operation is complete.

Bits 5 to 4 - Empty and not used, but must be zero.

Bit 3 - Control function specification.

0 = The function provided in bits 2 to 0 is a data transfer function and the block of information to be transferred must be address checked.

1 = The function provided in bits 2 to 0 is a control function and no address check need be performed.

Bits 2 to 0 - The function code of the I/O operation to be performed.

Word 3 (PKT+2)

Bits 15 to 8 - Immediate mode specification.

0 = Immediate mode is not specified and the item count and buffer address are contained in words 4 and 5 respectively.

Not equal to 0 = Immediate mode is specified and this byte contains the number of items to be transferred. The buffer address is the address of word 4.

NOTE

Immediate mode is not legal for all devices. Appendix B lists those devices.

Bits 7 to 0 - The I/O channel on which the I/O operation is to be performed. Specify the absolute device number.

Word 4 (PKT+4)

Bits 15 to 0 - If immediate mode is specified, this word is the start of the input or output buffer. Otherwise this word contains the number of items to be transferred. The size of an item is dependent upon the device on which the transfer is to take place. For character devices such as the paper tape reader, the item count is in bytes. For word devices such as the various disks, the item count is in words.

Word 5 (PKT+6)

Bits 15 to 0 - If immediate mode is specified, this is the second word of the input or output

buffer. Otherwise this word contains the starting address of the input or output buffer. The address must be on a boundary that is in agreement with the type of device. For example, transfers to and from word-oriented devices cannot occur on byte boundaries.

Words 6 to 8 (PKT+8 to PKT+12)

Bits 15 to 0 - Device dependent information.

Error Conditions:

Address check on I/O packet.

Address check on end action address.

Illegal device specification.

Illegal I/O function.

Illegal immediate mode specification.

Performance Data:

Runs on system stack.

Generatable ER.

Core requirements: 132 words.

Additional routines required: ACHCK, QUEUE, and REQSB.

User stack requirements: 19 words.

Program Example:

Transfer a buffer of data to device number 2 (i.e., a paper tape punch) with hold and end action.

```
         R$QIO   PKT,NORM

         .WORD   ENDA             ;END ACTION ADDRESS
         .WORD   0                ;
PKT:     .BYTE   $HOLD+$EA+$WR,0  ;FUNCTIONS
         .BYTE   2,0              ;DEVICE
         .WORD   100              ;ITEM COUNT
         .WORD   BUF              ;BUFFER ADDRESS

NORM:    Start of no room coding.

ENDA:    Start of end action coding.
```

2.2.7 End Action Control

2.2.7.1 End Action Wait

Functional Description:

Suspend the execution of the requesting task until one of its outstanding end action requests is satisfied.

The end action request count is tested. If the count is nonzero, the task is placed in end action wait and a skip return is set up for when the task is reactivated. Otherwise a nonskip return is executed.

Request Syntax:

The macro calling sequence is:

```
E$ACTW  NONE
```

The assembly language generated from the macro expansion is:

```
EMT     34
.WORD   NONE
```

Parameters:

NONE: The nonskip "place to go" address if there are no outstanding end action requests.

Error Conditions:

None.

Performance Data:

Runs on user stack.

Core requirements: 17 words.

Additional routines required: REQSB.

User stack requirements : 21 words.

Program Example:

Wait for first outstanding end action request.

```
        E$ACTW  NONE
NONE:   Start of no outstanding requests coding.
```

2.2.7.2 End Action Return

Functional Description:

Return control to the interrupted place in the requestor from end action coding.

The current context is discarded and the next context is loaded from the task's stack. A return is executed.

Request Syntax:

The macro calling sequence is:

```
E$ACTR
```

The assembly language generated from the macro expansions is:

```
EMT     35
```

Parameters:

None.

Error Conditions:

None.

Performance Data:

Runs on user stack.

Core requirements: 6 words.

Additional routines required: None.

User stack requirements: 10 words.

Program Example:

Return from end action coding.

```
E$ACTR
```

2.2.8 List Manipulation

2.2.8.1 Define List

Functional Description:

Generate a list structure. The requesting task specifies a descriptor that describes the list to be generated.

Lists generated via this ER may be manipulated with the make entry in and remove entry from list ER's.

A list descriptor contains five words. Three words are specified by the requesting task and contain the length of each list entry, the starting address of the list, and the length of the list area. The other two words are filled in by the ER routine with the addresses of the first and last entries in the generated list.

Each entry in the generated list is linked to the next entry in the list via an address in the first word of the entry. The last entry in the list contains a link of zero. As many list entries are generated as will fit in the defined list area.

This ER can be used to generate a buffer pool. The buffer pool would be considered as a list of empty buffers. The make entry in and remove entry from list ER's can then be used for the operations of returning buffers to and requesting buffers from the pool.

The specified list is generated, the requesting task's stack is cleared, and a return is executed.

Request Syntax:

The macro calling sequence is:

```
D$LIST  DESC
```

The assembly language generated from the macro expansion is:

```
MOV     #DESC,-(SP)
EMT     36
```

Parameters:

DESC: The address of a five-word list descriptor with the following format:

Word 1 (DESC)

Bits 15 to 0 - Filled by the ER code with the address of the first entry in the list.

Word 2 (DESC+2)

Bits 15 to 0 - Filled by the ER code with the address of the last entry in the list.

Word 3 (DESC+4)

Bits 15 to 0 - The number of words in each list entry.

Word 4 (DESC+6)

Bits 15 to 0 - The length of the core area allocated for the list in bytes.

Word 5 (DESC+10)

Bits 15 to 0 - The starting address of the list (must be on a word boundary).

Error Conditions:

Address check on list description.

Address check on list area.

Performance data:

Runs on user stack.

Core requirements: 34 words.

Addition routines required: ACHCK.

User stack requirements: 21 words.

Program Example:

Define a list with four-word entries starting at the address LT. The length of the list area is 050 bytes.

```
        D$LIST  DESC

DESC:   .WORD   0                   ;FILLED WITH LT
        .WORD   0                   ;FILLED WITH LT+40
        .WORD   4                   ;LENGTH OF ENTRY
        .WORD   50                  ;LENGTH OF LIST AREA
        .WORD   LT                  ;START OF LIST
```

The resulting list would appear as follows:

```
LT:     .WORD   LT+10
          .
          .
LT+10:  .WORD   LT+20
          .
          .
LT+20:  .WORD   LT+30
          .
          .
LT+30:  .WORD   LT+40
          .
          .
LT+40:  .WORD   0
```

NOTE

For the special cases where there is insufficient space specified to generate any list entries or the length of an entry is defined as zero, the filled descriptor words take on a different format. The address of the first entry in the list is returned as zero indicating there are no entries in the list. The address of the last entry in

the list is returned as the address of the first word of the descriptor. This format is perfectly legal and is used to describe an empty list for all of the list manipulation ER's.

2.2.8.2 Remove Entry From List

Functional Description:

Remove the first entry from a list. The requesting task specifies a two-word list descriptor.

This ER runs on the system stack and is therefore noninterruptible, making it especially useful for manipulating lists from end action coding where reentrancy may be a problem.

An attempt is made to remove the first entry from the specified list. If there are no entries in the list, the requesting task's stack is cleared and a nonskip return is executed. Otherwise the address of the entry is placed on the top of the requesting task's stack and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
R$MOV   DESC,NONE
```

The assembly language generated from the macro expansion is:

```
MOV     #DESC,-(SP)
EMT     37
.WORD   NONE
```

Parameters:

NONE: The nonskip "place to go" address if there are no entries in the list.

DESC: The address of a two-word list descriptor. The format of these words is as defined for the first two words of the define list ER (i.e., the addresses of the first and last entries in the list).

Error Conditions:

Address check on list descriptor.

Address check on address of first entry in list.

Address check on address of last entry in list.

Performance Data:

Runs on system stack.

Core requirements: 26 words.

Additional routines required: ACHCK.

User stack requirements: 19 words.

Program Example:

Remove the first entry from the list described by the descriptor DESC.

```
        R$MOV   DESC,NONE

        MOV     (SP)+,R0        ;ADDRESS OF FIRST

DESC:   .WORD   FIRST           ;ADDRESS OF FIRST
        .WORD   LAST            ;ADDRESS OF LAST

NONE:   Start of empty list coding.
```

2.2.8.3 Make Entry In List

Functional Description:

Add a new entry to the end of a list. The requesting task specifies a two-word list descriptor.

This ER runs on the system stack and is therefore noninterruptible, making it especially useful for manipulating lists from end action coding where reentrancy may be a problem.

The new entry is placed at the end of the specified list. The requesting task's stack is cleared and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        M$ENT   DESC,ENT
```

The assembly language generated from the macro expansion is:

```
        MOV     #ENT,-(SP)
        MOV     #DESC,-(SP)
        EMT     40
```

Parameters:

ENT: The address of the entry to be added to the end of the list. The first word of this entry must be free as it will be used to link the entry into the list.

DESC: The address of a two-word list descriptor. The format of these words is as defined for the first two words of the define list ER (i.e. the addresses of the first and last entries in the list).

Error Conditions:

Address check on list descriptor.

Address check on address of first entry in list.

Address check on address of last entry in list.

Address check on address of new entry.

Performance Data:

Runs on system stack.

Core requirements: 26 words.

Additional routines required: ACHCK.

User stack requirements: 19 words.

Program Example:

Add the entry at address ENT to the end of the list described by the descriptor D1.

```
        M$ENT   D1,ENT

D1:     .WORD   FIRST           ;ADDRESS OF FIRST
        .WORD   LAST            ;ADDRESS OF LAST

ENT:    .WORD   LINK            ;LINK WORD
          .     DATA            ;ENTRY DATA
          .     DATA            ;
        .WORD   DATA            ;
```

2.2.9 Dynamic Storage

2.2.9.1 Request Buffer Block

Functional Description:

Request a buffer block from the dynamic storage pool. The requesting task specifies the size of the block required in 8-word units.

If the request cannot be accepted because a contiguous block of memory the size requested is not available, then the requesting task's stack is cleared and a nonskip return is executed. Otherwise the address of the block is placed on the top of the requesting task's stack and a skip return is executed.

Request Syntax:

The macro calling sequence is:

```
G$BUF   SIZE,NOBF
```

The assembly language generated from the macro expansion is:

```
MOV     #SIZE,-(SP)
EMT     41
.WORD   NOBF
```

Parameters:

SIZE: The size of the buffer block desired expressed in 8-word units.

NOBF: The nonskip "place to go" address if enough contiguous core is not available.

Error Conditions:

Requesting task not privileged.

Performance Data:

Runs on system stack.

Core requirements: 15 words.

Additional routines required: RQLCB.

User stack requirements: 19 words.

Program Example:

Request a buffer block 16 words in length.

```
        G$BUF   2,NOBF
        MOV     (SP)+,R0        ;BUFFER ADDRESS

NOBF:   Start of no available buffer coding.
```

2.2.9.2 Release Buffer Block

Functional Description:

Release a buffer block to the dynamic storage pool. The requesting task specifies the address and size of the block to be released in 8-word units.

The specified buffer block is released, the requesting task's stack is cleared, and a return is executed.

Request Syntax:

The macro calling sequence is:

```
R$BUF   SIZE,BUF
```

The assembly language generated from the macro expansion is:

```
MOV     #BUF,-(SP)
MOV     #SIZE,-(SP)
EMT     42
```

Parameters:

SIZE: The size of the buffer block to be released in 8-word units.

BUF: The starting address of the buffer block to be released.

Error Conditions:

Requesting task not privileged.

Illegal buffer block address.

Performance Data:

Runs on system stack.

Core requirements: 33 words.

Additional routines required: RQLCB.

User stack requirements: 19 words.

Program Example:

Release a buffer block 32 words in length whose address is ALFA.

```
R$BUF   4,ALFA
```

2.2.10 Miscellaneous

2.2.10.1 Set Error Trap Address

Functional Description:

Set error trap address so requestor may handle its own errors. The requesting task specifies the address that is to be trapped to when an error occurs.

Errors that may cause traps include illegal ER's (parameters, codes, etc.), processor errors (odd word address, illegal instruction, or nonexistent memory), and stack overflow. All error traps occur at the software priority of the task. When

an illegal ER or processor error occurs, the task is trapped to with the stack in a state such that it can determine why the trap occurred. The task is given control with register R0 loaded with the error code (see Appendix C) and the stack in the following state:

SP+14	PS-processor status at time of error.
SP+12	PC-program address at time of error.
SP+10	R0-register 0 at time of error.
SP+8	R1-register 1 at time of error.
SP+6	R2-register 2 at time of error.
SP+4	R3-register 3 at time of error.
SP+2	R4-register 4 at time of error.
SP	R5-register 5 at time of error.

Stack overflow is handled in much the same manner; however, the state of the stack when the overflow occurs is somewhat indeterminate. The alternate stack is switched to when stack overflow is detected and a trap to the task is constructed on this stack. The task gains control with register R0 loaded with the error code 030 (stack overflow) and register R1 loaded with the address of the top of the old stack just before the overflow occurred. In general the task will probably not be able to recover properly from this situation.

The error trap address is inserted into the partition status table, the requesting task's stack is cleared, and a return is executed.

Request Syntax:

The macro calling sequence is:

```
E$RSET  ERRA
```

The assembly language generated from the macro expansion is:

```
MOV     #ERRA,-(SP)
EMT     51
```

Parameters:

ERRA: The address to be trapped to when the requesting task commits an error.

Error Conditions:

Address check on error trap address.

Performance Data:

Runs on user stack.

Core requirements: 11 words.

Additional routines required: None.

User stack requirements: 21 words.

Program Example:

Set the error trap address to GOGO.

```
        E$RSET  GOGO

GOGO:   Start of error handling code.
```

2.2.10.2 Set Alternate Stack Address

Functional Description:

Define an alternate stack to be used in case stack overflow is detected while the requestor is in execution. The requesting task specifies the starting address of the alternate stack. The alternate stack must be at least 9 words in length.

When a task is initialized the alternate stack address for the task is defined to be the same as its starting stack address. Therefore, if stack overflow is detected, the overflow will occur onto the stack already in use. This destroys information on the stack and is probably undesirable in most cases. Stack overflow should be avoided if at all possible. However, if it can occur, this ER should be used to define an alternate stack so information on the stack in use will not be lost.

The alternate stack address is set, the requesting task's stack is cleared, and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        A$LSET  ASTK
```

The assembly language generated from the macro expansion is:

```
        MOV     #ASTK,-(SP)
        EMT     52
```

Parameters:

ASTK: The starting address of the alternate stack.

Error Conditions:

Address check on alternate stack.

Performance Data:

Runs on user stack.

Core requirements: 12 words.

Additional routines required: ACHCK.

User stack requirements: 21 words.

Program Example:

Set alternate stack address to ALTS.

```
        A$LSET  ALTS

.=.+22                          ;RESERVE SPACE
ALTS:                           ;START OF STACK
```

2.2.10.3 Set TRAP Trap Address

Functional Description:

Set address to be trapped to if a TRAP instruction is executed from the requestor. The requesting task specifies the address to be trapped to.

When a task is initialized the TRAP trap address is set to the illegal instruction error exit. A task may enable the trap to itself by executing this ER. The trap occurs directly to the task with the old processor status and program counter on the top of the stack.

The TRAP trap address is set, the requesting task's stack is cleared, and a return is executed.

Request Syntax:

The macro calling sequence is:

```
        S$TRAP  TRPA
```

The assembly language generated from the macro expansion is:

```
        MOV     #TRPA,-(SP)
        EMT     53
```

Parameters:

TRPA: Address to be trapped to if a TRAP instruction is executed from the requesting task.

Error Conditions:

Address check on trap address.

Performance Data:

Runs on user stack.

Core requirements: 11 words.

Additional routines required: ACHCK.

User stack requirements: 21 words.

Program Example:

Set TRAP trap address to TERP.

```
        S$TRAP  TERP

TERP:   Start of TRAP trap coding.
```

CHAPTER 3

OPERATOR COMMUNICATION

3.1 GENERAL DESCRIPTION

A user may communicate directly with the system via the operator's console task. Communications may occur on any terminal (Teletype, VT05, etc.) that has been defined as an operator's console (see device dependent information in Appendix B). Commands (keyins) are processed serially and thus only one terminal may communicate with the operator's console task at a given time.

A communication is begun by typing a character (any character will suffice) on a terminal. If no other task is currently soliciting input on the terminal, then a message is sent to the operator's console task specifying that terminal.

The operator's console task receives the message, outputs an asterisk (*), and solicites one line of input. The keyin is then entered followed by a carriage return.

3.1.1 Functions Provided

The following functional capabilities are provided by the operator's console task via keyins:

a) Examine memory.

b) Deposit memory.

c) Enter date.

d) Enter time of day.

e) Request execution of a task.

f) Request asynchronous periodic execution of a task.

g) Request synchronous periodic execution of a task.

h) Request execution of a task at a specific time of day.

i) Activate execution of a task.

j) Suspend execution of a task.

k) Delete a task from the system.

l) Load a new task into the system from paper tape.

m) Execute a breakpoint trap (for ODT).

3.1.2 Error Handling

All input parameters for a particular keyin are verified before the command is actually executed. An appropriate error message is printed if an error is detected. The following error messages are defined.

a) INSF PRM - An insufficient number of parameters has been supplied.

b) LATER - The requested function cannot be executed at the current time (i.e., queue full, no buffers, etc.).

c) ILL PARM NN - NN represents an illegal parameter. It may be of the wrong type or of improper magnitude. Parameter 00 is defined as the command mnemonic itself.

d) ILL REQ - The requested function caused an illegal operation to be executed (i.e., undefined task, nonexistent memory, etc.).

e) LOAD ERR - An error was encountered during task loading (checksum, bad transfer, etc.).

3.1.3 Command Syntax

A command consists of a command mnemonic and may be followed by one or more parameter fields. Fields are separated from one another by a comma; a blank or carriage return at the end of a field terminates the scan of the keyin.

Command parameters may be either numeric or alphanumeric and are error checked to ensure that the proper type of parameter occurs within a given field. Alphanumeric parameters may contain alphabetic (A thru Z) or numeric (0 thru 9) characters and may start with either. Numeric parameters may contain only the characters 0 thru 9 (7 if octal field). Decimal fields may be positive or negative; octal fields are distinguished by a leading '#' sign.

The following example illustrates a command with 3 parameters. The first parameter is alphanumeric and the remaining two are numeric.

```
CMND,ALPF,+1000,#177776
```

3.2 KEYINS

The following subsections describe each keyin that is available via the operator's console task.

3.2.1 Examine Memory

Functional Description:

Examine the contents of one or more memory locations. The address and contents of each memory location are printed in octal format.

Keyin Syntax:

EXAM,FIRST

or

EXAM,FIRST,LAST

or

EXAM,FIRST,COUNT

Parameters:

EXAM: The command mnemonic.

FIRST: The address of the first memory location to be examined.

LAST: The address of the last memory location to be examined.

COUNT: The number of memory locations to be examined.

Error Conditions:

Nonexistent memory address.

Odd memory address.

Keyin Example:

Examine the contents of memory locations #10, #12, #14, and #16.

EXAM,#10,#16.

or

EXAM,#10,4

The resultant printout:

```
000010  xxxxxx
000012  xxxxxx
000014  xxxxxx
000016  xxxxxx
```

3.2.2 Deposit Memory

Functional Description:

Deposit one or more values into consecutive memory locations. The address and stored contents of each memory location are

printed in octal format. Up to 15 values may be specified with a single keyin.

Keyin Syntax:

DEPO,ADDR,V1,V2,V3,...,Vn

Parameters:

DEPO: The command mnemonic.

ADDR: The address of the memory location in which the first value is to be stored. Other values are stored in consecutive locations thereafter.

V1,V2,V3,...VN: A series of values that are to be stored in memory.

Error Conditions:

Nonexistent memory address.

Odd memory address.

Keyin Example:

Deposit the values 100, #10, #15, 199, and 205 into consecutive memory locations starting at #5776.

DEPO,#5576,100,#10,#15,199,205

The resultant printout:

```
005776  000144
006000  000010
006002  000015
006004  000307
006006  000315
```

3.2.3 Enter Date

Functional Description:

Enter the Julian date into the system.

Keyin Syntax:

DATE,DAY,YEAR

Parameters:

DATE: The command mnemonic.

DAY: The Julian day of the year (i.e., 1 thru 365 or 366 if leap year).

YEAR: The current year expressed as a two-digit number (i.e., 72 for 1972).

Error Conditions:

Illegal day.

Illegal year.

Keyin Example:

Set the system date to March 23, 1972.

DATE,83,72

3.2.4 Enter Time

Functional Description:

Enter the time of day into the system in military format.

Keyin Syntax:

TIME,HOUR

or

TIME,HOUR,MINUTE

or

TIME,HOUR,MINUTE,SECOND

Parameters:

TIME: The command mnemonic.

HOUR: The hour of the day.

MINUTE: The minute of the hour.

SECOND: The second of the minute.

Error Conditions:

Illegal hour.

Illegal minute.

Illegal second.

Keyin Example:

Set the system time of day to 1:45 PM and 30 seconds.

TIME,13,45,30

3.2.5 Request Task Execution

Functional Description:

Request the execution of any task in the system.

Keyin Syntax:

RQEX,TNAME

Parameters:

RQEX: The command mnemonic.

TNAME: The alphanumeric name of the task to be executed.

Error Conditions:

Undefined task.

Keyin Example:

Request the execution of the task FLOP.

RQEX,FLOP

3.2.6 Request Asynchronous Periodic Task Execution

Functional Description:

Request the asynchronous periodic execution of any task in the system.

Keyin Syntax:

RQAX,TNAME,PERIOD

Parameters:

RQAX: The command mnemonic.

TNAME: The alphanumeric name of the task to be executed asynchronous periodic.

PERIOD: The period of execution expressed in line frequency units.

Error Conditions:

Undefined task.

Keyin Example:

Request the asynchronous periodic execution of the task REPT with a period of 2 minutes.

RQAX,REPT,7200

3.2.7 Request Synchronous Periodic Task Execution

Functional Description:

Request the synchronous periodic execution of any task in the system.

Keyin Syntax:

RQSX,TNAME,PERIOD

Parameters:

RQSX: The command mnemonic.

TNAME: The alphanumeric name of the task to be executed synchronous periodic.

PERIOD: The period of execution expressed in line frequency units.

Error Conditions:

Undefined task.

Keyin Example:

Request the synchronous periodic execution of the task AGAN with a period of 100 milliseconds.

RQSW,AGAN,6

3.2.8 Request Task Execution at Time of Day

Functional Description:

Request the execution of any task at a specific time of day.

Keyin Syntax:

RQTX,TNAME,TIME

Parameters:

RQTX: The command mnemonic.

TNAME: The alphanumeric name of the task to be executed at the specified time of day.

TIME: The time of day that the task is to be executed expressed as time past midnight in line frequency units.

Error Conditions:

Undefined task.

Keyin Example:

Request the execution of the task SCHD at 8:45 PM.

RQTX,SCHD,4482000

NOTE: 4482000=(20 hours*60 minutes+45 minutes)*60 seconds*60 line frequency units.

3.2.9 Activate Task

Functional Description:

Activate the execution of any task in the system:

Keyin Syntax:

ACTV,TNAME

Parameters:

ACTV: The command mnemonic.

TNAME: The alphanumeric name of the task to be activated.

Error Conditions:

Undefined task.

Keyin Example:

Activate the execution of the task FOOL.

ACTV,FOOL

3.2.10 Suspend Task

Functional Description:

Suspend the execution of any task in the system.

Keyin Syntax:

SPND,TNAME

Parameters:

SPND: The command mnemonic.

TNAME: The alphanumeric name of the task to be suspended.

Error Conditions:

Undefined task.

Keyin Example:

Suspend the execution of the task CROW.

SPND,CROW

3.2.11 Delete Task

Functional Description:

Delete any task from the system. The specified task cannot currently be in execution or the request will be denied.

Keyin Syntax:

DELT,TNAME

Parameters:

DELT: The command mnemonic.

TNAME: The alphanumeric name of the task to be deleted.

Error Conditions:

Undefined task.

Keyin Example:

Delete the task GONE from the system.

DELT,GONE

3.2.12 Load Task

Functional Description:

Load a new task into the system from the high-speed paper tape reader. The load module must be in absolute loader format and may have been produced by either the assembler or linker.

Keyin Syntax:

If core-only task system:

LOAD,TNAME,PRI,GRP,EXP,CMP,PRV,IMD

If core-disk task system:

LOAD,TNAME,PRI,GRP,EXP,CMP,PRV,IMD,RES(,DADR,DEV)

Parameters:

LOAD: The command mnemonic.

TNAME: The alphanumeric name of the task to be loaded.

PRI: The priority of the task expressed as the position in the task table (i.e., priority 2 for the second position in the table).

GRP: The group of which the task is to become a member.

EXP: The number of the partition that the task is to execute out of.

CMP: The number of the task's common data partition. If the task does not have a common data partition, CMP should be set equal to EXP.

PRV: The alphanumeric designation of the task's privilege status. 'PV' means privileged and anything else means nonprivileged.

IMD: The alphanumeric designation of the task's execution status. 'IM' means immediate execution at system startup and anything else means no execution at startup.

RES: The alphanumeric designation of the task's residency status. 'RS' means the task is core resident and anything else means the task is disk resident. This parameter is required only in core-disk task systems.

DADR: The starting disk address of the area on the disk to which that the task is to be written. This parameter is required only if RES is not equal to 'RS'.

DEV: The device number of the disk on which the task is to be written. This parameter is required only if RES is not equal to 'RS'.

Error Conditions:

Task name already defined.

Task table entry occupied.

Undefined priority (i.e., no such position in table).

Undefined execution of common partition (i.e., no such position in table).

Illegal device number (not a disk or undefined).

Keyin Examples :

Load the task DEMO into the system. The task is to have a priority of 3 (occupies 3rd task table entry), be a member of group 25, have an execution and common partition of 6, be nonprivileged, no immediate execution and be core resident. A core-only task system is assumed.

LOAD,DEMO,3,25,6,6,NP,NI

Load the task SWAP into the system. The task is to have a priority of 4, be a member of group 31, have an execution partition of 5, have a common partition of 10, be privileged, no immediate execution, and be disk resident. It is to be

written to device 8 starting at disk address 5. A core-disk task system is assumed.

LOAD,SWAP,4,31,5,10,PV,NI,NR,5,8

PROGRAMMING NOTE

The user must allocate all disk space used for nonresident tasks. When a nonresident task is to be loaded into the system, it is first loaded into the actual execution partition and then written to the disk. The entire execution partition is written, thus the task can be read into core via a single disk access. Care should be taken to ensure that enough disk space has been allocated since the entire execution partition is written to the disk. Disks are addressable by logical sector and space should be allocated in these units (256 words/logical sector).

3.2.13 Breakpoint Trap

Functional Description:

Execute a breakpoint trap instruction (code 3) to trap to ODT.

This keyin provides a controlled entry into ODT. Execution of the system may be continued via a proceed keyin to ODT.

Keyin Syntax:

BKPT

Parameters:

BKPT: The command mnemonic.

Error Conditions:

None.

Keyin Example:

Break to ODT.

BKPT

CHAPTER 4

SAMPLE TASKS

4.1 SAMPLE TASK #1

The following task is a paper tape copy program using hold I/O.

```
        .TITLE  PAPER
        .CSECT
;
;               VERSION V001A
;
;       MINIMUM SYSTEM PAPER TAPE COPY PROGRAM
;
;       DEFINITIONS
;

CON=0                                   ;CONSOLE UNIT NUMBER
PTR=1                                   ;PAPER TAPE READER UNIT NUMBER
PTP=2                                   ;PAPER TAPE PUNCH UNIT NUMBER

;
;       I/O PACKET DEFINITIONS
;

        .WORD   0                       ;
        .WORD   0                       ;
PKT1:   .BYTE   $WR+$HOLD,0             ;SOLICITATION OUTPUT
        .BYTE   CON,PKT3-PKT2           ;
PKT2:   .ASCII  /PUT TAPE IN READER AND DEPRESS CR!/;
        .BYTE   15,12                   ;
PKT3:                                   ;
        .EVEN

        .WORD   0                       ;
PKT4:   .BYTE   $RD+$HOLD,0             ;SOLICITE INPUT
        .BYTE   CON,1                   ;
        .WORD   0                       ;

        .WORD   0                       ;
PKT5:   .BYTE   $RD+$HOLD,0             ;READ PAPER TAPE
        .BYTE   PTR,0                   ;
        .WORD   BUFL                    ;
        .WORD   BUF                     ;

        .WORD   0                       ;
PKT6:   .BYTE   $WR+$HOLD,0             ;WRITE PAPER TAPE
        .BYTE   PTP,0                   ;
        .WORD   BUFL                    ;
        .WORD   BUF                     ;
```

```
        .WORD   0                   ;
PKT7:   .BYTE   $WR+$HOLD,0         ;PAPER TAPE TRANSFER ERROR
        .BYTE   CON,PKT9-PKT8       ;
PKT8:   .ASCII  /PUNCH ERROR/       ;
        .BYTE   15,12               ;
PKT9:                               ;
        .EVEN

        .WORD   0                   ;
PKT10:  .BYTE   $WR+$HOLD,0         ;FINAL MESSAGE
        .BYTE   CON,PKT12-PKT11     ;
PKT11:  .ASCII  /THAT'S ALL FOLKS!/;
        .BYTE   15,12               ;
PKT12:                              ;
        .EVEN

;
;       PAPER TAPE INPUT OUTPUT BUFFER
;

BUFL=200

BUF:    .=.+BUFL

;
;       STACK AREA
;

.=.+40.                             ;20 WORDS OF STACK SPACE (DECIMAL)

;
;       START OF COPY PROGRAM
;

COPY:   R$QIO   PKT1,COPY+4         ;OUTPUT SOLICITATION MESSAGE
COPY1:  R$QIO   PKT4,COPY1+4        ;SOLICITE INPUT
COPY2:  R$QIO   PKT5,COPY2+4        ;READ AN INPUT BUFFER
        MOV     #BUFL,R0            ;GET BUFFER LENGTH
        SUB     PKT5-2,R0           ;SUB BYTES NOT TRANSFERED
        BEQ     COPY4               ;IF EQ NONE
        MOV     R0,PKT6+4           ;SET BYTES TO PUNCH
COPY3:  R$QIO   PKT6,COPY3+4        ;PUNCH OUT BUFFER
        TSTB    PKT6+1              ;TEST FINAL PUNCH STATUS
        BNE     COPY5               ;IF NE ERROR
        TSTB    PKT5+1              ;TEST FINAL READ STATUS
        BEQ     COPY2               ;IF EQ MORE TO PUNCH
COPY4:  R$QIO   PKT10,COPY4+4       ;OUTPUT FINAL MESSAGE
        TSERM                       ;TERMINATE TASK
COPY5:  R$QIO   PKT7,COPY5+4        ;OUTPUT ERROR MESSAGE
        TSERM                       ;TERMINATE TASK
        .END    COPY                ;
```

4.2 SAMPLE TASK #2

The following task is a paper tape copy program which makes use of end action ERs.

```
          .TITLE  PAPCOP
          .CSECT
;
;
;         PAPER TAPE COPY TASK
;
;         USES END ACTION ER'S
;
;         DEFINITIONS
;

BUFL=1000
CN=0                                    ;CONSOLE TTY
PR=1                                    ;PAPER TAPE READER
PP=2                                    ;PAPER TAPE PUNCH

;
;         I/O PACKETS
;

          .WORD   0                     ;FOR ADDRESS CHECK ERROR
          .WORD   0                     ;
PKT1:     .WORD   $HOLD+$WR             ;OUTPUT SOLICITATION
          .BYTE   CN,PKT3-PKT2          ;
PKT2:     .ASCII  'PUT TAPE IN READER AND TYPE (CR)';
          .BYTE   12,15                 ;
PKT3:                                   ;
          .EVEN
          .WORD   0                     ;
PKT4:     .WORD   $HOLD+$RD             ;SOLICITE INPUT
          .BYTE   CN,1                  ;
PKT5:     .WORD   0                     ;
          .WORD   COP5                  ;
          .WORD   0                     ;
PKT6:     .WORD   $EA+$RD               ;READ TAPE WITH END ACTION
          .BYTE   PR,0                  ;
          .WORD   176                   ;
          .WORD   0                     ;
          .WORD   COP4                  ;
          .WORD   0                     ;
PKT7:     .WORD   $EA+$WR               ;WRITE TAPE WITH END ACTION
          .BYTE   PP,0                  ;
          .WORD   176                   ;
          .WORD   0                     ;
          .WORD   0                     ;
PKT8:     .WORD   $HOLD+$WR             ;END OF JOB
          .BYTE   CN,PKT10-PKT9         ;
PKT9:     .ASCII  'TAPE COPIED'         ;
          .BYTE   12,15                 ;
PKT10:                                  ;
          .EVEN
```

```
;
;       LIST DESCRIPTORS
;

LSTD1:  .WORD   0               ;
        .WORD   0               ;
        .WORD   100             ;SIZE OF ENTRIES IN WORDS
        .WORD   BUFL            ;LENGTH OF LIST AREA
        .WORD   BUF             ;STARTING ADDRESS OF LIST AREA
LSTD2:  .WORD   0               ;
        .WORD   LSTD2           ;

;
;       READ/WRITE BUFFER
;

BUF:    .=.+BUFL                ;

;
;       ROUTINE BUSY FLAGS
;

RFLG:   .WORD   -1              ;READ ROUTINE
WFLG:   .WORD   -1              ;WRITE ROUTINE

;
;       STACK SPACE
;

.=.+200.

;
;       START OF TASK
;

START:  R$QIO   PKT1,START+4    ;SOLICITE INPUT
COP0:   R$QIO   PKT4,COP0+4     ;
        D$LIST  LSTD1           ;DEFINE LIST
        MOV     #-1,RFLG        ;RESET FLAGS
        MOV     #-1,WFLG        ;
        JSR     PC,READ         ;READ FIRST BUFFER
COP2:   E$ACTW  COP3            ;WAIT FOR END ACTION
        BR      COP2            ;
COP3:   R$QIO   PKT8,COP3+4     ;OUTPUT FINAL MESSAGE
        T$ERM                   ;TERMINATE TASK
```

```
;
;       END ACTION CODE FOR PUNCH REQUEST
;

COP4:   MOV     PKT7+6,-(SP)       ;RELEASE BUFFER TO POOL
        SUB     #2,(SP)            ;ADJUST FOR LINK WORD
        MOV     #LSTD1,-(SP)       ;
        EMT     40                 ;
        JSR     PC,READ            ;START NEXT READ
        JSR     PC,WRIT1           ;START NEXT WRITE
        E$ACTR                     ;RETURN FROM END ACTION
WRIT:   INC     WFLG               ;ROUTINE BUSY?
        BNE     WRIT3              ;IF NE YES
WRIT1:  R$MOV   LSTD2,WRIT4        ;REMOVE NEXT BUFFER
        ADD     #2,(SP)            ;STEP OVER LINK WORD
        MOV     (SP)+,PKT7+6       ;INSERT BUFFER ADDRESS
WRIT2:  R$QIO   PKT7,WRIT2+4       ;PUNCH BUFFER OUT
WRIT3:  CLR     WFLG               ;DON'T ALLOW OVERFLOW
        RTS     PC                 ;RETURN
WRIT4:  MOV     #-1,WFLG           ;RESET BUSY FLAG
        RTS     PC                 ;RETURN

;
;       END ACTION CODE FOR READ REQUEST
;

COP5:   MOV     PKT6+6,-(SP)       ;MAKE LIST ENTRY
        SUB     #2,(SP)            ;ADJUST FOR LINK WORD
        TSTB    PKT6+1             ;GOOD TRANSFER?
        BNE     COP6               ;IF NE NO
        MOV     #LSTD2,-(SP)       ;SPECIFY PUNCH LIST
        EMT     40                 ;
        JSR     PC,WRIT            ;START PUNCHING
        JSR     PC,READ1           ;READ NEXT BUFFER
        E$ACTR                     ;END ACTION RETURN
COP6:   MOV     #LSTD1,-(SP)       ;SPECIFY BUFFER POOL
        EMT     40                 ;
        E$ACTR                     ;RETURN FROM END ACTION
READ:   INC     RFLG               ;ROUTINE BUSY?
        BNE     READ3              ;IF NE YES
READ1:  R$MOV   LSTD1,READ4        ;GET A FREE BUFFER
        ADD     #2,(SP)            ;STEP OVER LINK WORD
        MOV     (SP)+,PKT6+6       ;INSERT BUFFER ADDRESS
READ2:  R$QIO   PKT6,READ2+4       ;READ NEXT BUFFER
READ3:  CLR     RFLG               ;DON'T ALLOW OVERFLOW
        RTS     PC                 ;RETURN
READ4:  MOV     #-1,RFLG           ;RESET BUSY FLAG
        RTS     PC                 ;RETURN

        .END    START              ;
```

CHAPTER 5

SYSTEM GENERATION

5.1 GENERAL DESCRIPTION

A wide variety of hardware/software configurations may be generated for RSX11A. These configurations are selected via the definition of appropriate system generation parameters. The interactive program GENFIL has been provided to aid in the definition process. The output of this program is a parameter definition file (CONFIG) and specific directions for assembling and linking the desired system.

System generation is considered a four-step process.

1) The definition of memory partitions and any tasks that are to be linked directly into the system load module.

2) The execution of the interactive program GENFIL to generate the configuration parameter file and obtain assembly and linking directions.

3) The assembly of all specified modules.

4) The linking of the resultant object modules to form the desired load module of RSX11A.

5.2 PARTITION TABLE DEFINITION

The partition table (see Appendix H) describes the allocation of core memory in fixed partitions. Up to 256 distinct partitions of any size may be defined. Entries in the partition table must be in ascending order of core address; care must be taken to ensure its accuracy.

The system is supplied with definitions for three partitions (i.e. null, operator's console, and nonresident task loader tasks). User-defined partitions must immediately follow.

Definition Syntax:

The partition table is located in the module TABLS and the following macro call is used to define entries:

```
        PART    BSY,BAS
```

The assembly language generated from the macro expansion is:

```
        .CSECT  AREA1
        .BYTE   $NQ+1,BSY+177

$NQ=$NQ+1
        .CSECT  AREA2
        .WORD   BAS
        .CSECT  AREA3
        .WORD   0
        .CSECT
```

Where:

BSY: The partition status flag. If a core resident task is to be linked directly into the partition, then BSY=1; otherwise, BSY=0 (i.e., the partition is free).

BAS: The base address of the partition (i.e., the lowest address in the partition).

The following example illustrates the generation of partition table entries.

Example:

Four partitions are to be defined and one will be occupied by a core resident task that is linked directly into the system load module. The partition sizes are to be: a) the size of the core resident task (starting address is STADR$ and ending address is ENADR$), b) 1K, c) 2K, d) a partition that occupies the remainder of core memory. The following macro calls define these partitions.

```
        .GLOBL  ENADR$
        .GLOBL  STADR$

        PART    1,STADR$
        PART    0,ENADR$+2
        PART    0,ENADR$+4002
        PART    0,ENADR$+14002
```

5.3 TASK TABLE DEFINITION

Undefined (empty) entries may be generated in the task table via the configuration parameter T$$SLT. Subsequently tasks may be loaded into these entries using the LOAD keyin of the operator's console task. If, however, tasks are to be linked directly into the system load module, the task table must be modified and an appropriate entry placed in the partition table (see above).

Definition Syntax:

The task table is located in the module TABLS and the following macro call is used to define entries in this table:

```
        TASK    TLBL,STAT,ENTR,GRP,NAM1,NAM2,EXPT,
                     CMPT,DEVN,DADR
```

The assembly language generated from the macro expansion is:

```
        .GLOBL  ENTR
        .GLOBL  TLBL
        .CSECT  TSK1
TLBL=.-TSK1$
        .WORD   STAT
        .CSECT  TSK2
        .WORD   ENTR
        .CSECT  TSK5
        .BYTE   GRP,$NQ+1
```

```
$NQ=$NQ+1
        .CSECT  TSK6
        .ASCII  NAM1
        .CSECT  TSK7
        .ASCII  NAM2
        .CSECT  TSK8
        .WORD   0
        .CSECT  TSK9
        .WORD   0
        .CSECT  TSK11
        .BYTE   EXPT,CMPT
        .IFDF   D$$ISC
        .CSECT  TSK12
        .BYTE   0,DEVN
        .CSECT  TSK13
        .WORD   DADR
        .ENDC
```

Where:

TLBL: An external label that is to be defined as the index of the task (i.e., $CNTSK for the operator's console task).

STAT: The initial status of the task (see Appendix F for bit definitions).

ENTR: The external entry point of the task.

GPP: The number (0 to 255) of the group of which the task is to be a member.

NAM1: The first two ASCII characters of the task name.

NAM2: The second two ASCII characters of the task name.

EXPT: The number of the partition in which the task is to execute.

CMPT: The number of the partition in which the task is to have access as a common data area.

DEVN: The number of the device on which the task resides. This parameter need only be defined in core-disk task systems. If the task is core resident, a definition of zero will suffice.

DADR: The logical starting disk address of the disk area in which the task is stored. This parameter need only be defined in core-disk task systems. If the task is core resident, then a definition of zero will suffice.

PROGRAMMING NOTE

The position of an entry in the task table determines the priority of the task defined by that entry; the closer to the front of the table, the higher the priority. Care must be taken to ensure that the property priority is assigned to tasks via their placement in

the task table. Undefined (empty) entries may be defined via the following macro call. This call should be used to space task table entries where appropriate (priority considerations):

```
TASK $NLTSK,$TDF,NULL$,0,↑/';;'/,↑/';;'/
          ,0,0,0,0
```

The mapping of tasks into execution and common data partitions is also very important. A task may have only one execution and common data partition. The mapping is via the position of the target partition in the partition table. For example, if a task's execution partition is the fourth entry and its common data partition is the tenth entry in the partition table, then the parameters EXPT and CMPT are defined as 4 and 10 respectively. If a task does not have a common data partition, then CMPT is defined equal to EXPT.

This mapping seems very complicated at first, but it provides a high degree of flexibility. The following example helps to clarify this mapping. A complete task and partition table is given. The entries that are provided with the system are marked as SYS; additional entries added for the example are marked EXP.

Example:

A system is to be generated that will contain the operator's console task, the nonresident task loader, and three user-defined tasks. Two of the user-defined tasks are to be linked directly into the system load module. These tasks are to occupy partitions of 3K and 4K respectively. The remaining task will be loaded as needed via the LOAD keyin of the operator's console task and will occupy the partition formed from the remaining memory. None of the tasks are to be members of group 5.

Partition Table Definitions:

```
        PART    1,NLST$                                     SYS
        .IFDF   C$$NSL                                      SYS
CP=1                                                        SYS
        PART    1,CNST$                                     SYS
        .ENDC                                               SYS
        .IFDF   D$$ISC                                      SYS
LP=2                                                        SYS
        PART    1,LDST$                                     SYS
        .ENDC                                               SYS
P1=3                                                        EXP
        PART    1,INITL$                                    EXP
P2=4                                                        EXP
        PART    1,INITL$+14000                              EXP
P3=5                                                        EXP
        PART    1,INITL$+14000+4000                         EXP
```

Task Table Definitions:

```
         .IFDF   D$$ISC                                      SYS
         TASK    $LDTSK,0,LOAD$,0,↑/';;'/,↑/';;'/,          SYS
                      LP,LP,0,0                              SYS
         .ENDC                                               SYS
         .IFDF   C$$NSL                                      SYS
         TASK    $CNTSK,0,CONS$,0,↑/';;'/,↑/';;'/,          SYS
                      CP,CP,0,0
         .ENDC                                               SYS
         TASK    $TASK1,0,ENT1$,5,'TS','K1',                 EXP
                      P1,P1,0,0
         TASK    $TASK2,0,ENT2$,5,'TS','K2',                 EXP
                      P2,P2,0,0
         TASK    $NLTSK,$TDF,NULL$,0,↑/';;'/,↑/';;'/,       EXP
                      0,0,0,0
         TASK    $NLTSK,$TEX,NULL$,0,↑/';;'/,N/';;'/,       SYS
                      0,0,0,0
```

The task not linked into the load module would be loaded into the system via the following LOAD keyin:

LOAD,NAME,4,5,5,5,NP,NI,RS

and executed with the keyin:

RQEX,NAME

NOTE

All tables are numbered from zero upward. Thus the first entry in a table is number 0.

5.4 PARAMETER FILE GENERATION

A parameter file is required to define system generation parameters. The interactive program GENFIL has been provided to aid in the generation and definition of this file.

GENFIL runs under DOS and is supplied in object module form. It must be linked for the target DOS configuration and then executed. As GENFIL executes, it solicits the definition of parameters that will become the body of the parameter file. Solicitation responses may be 'Y', a number, or a carriage return. A 'Y' or number response (whichever is appropriate) implies an affirmative answer while carriage return always implies a negative answer.

The command input/output of GENFIL is to/from the logical devices CMI/CMO respectively with a physical default of KB for both. Two output files are generated: 1) TXTOUT to logical device TXT with physical default LP, 2) CONFIG to logical device PFL with physical default KB. The file TXTOUT contains instructions specifying which modules to assemble and link to generate the desired system. CONFIG is the parameter file and contains the parameter definitions that were specified during the dialog.

GENFIL asks four sets of questions.

The first set deals with executive requests and are answered with 'Y' or carriage return.

The second set deals with I/O drivers. All but two of these questions are also answered with 'Y' or carriage return. The exceptions are: 1) Multi-terminal driver - answered with the number of terminals minus one or carriage return, 2) UDC11 driver - answered with the number of functional modules or carriage return.

The third set of questions deals with table sizes and additional functional capabilities. All but two of these questions are answered with a number or carriage return. The exceptions are: 1) nonresident task support - answered with 'Y' or carriage return, 2) EAE support - answered with a 'Y' or carriage return 3) console task answered with a 'Y' or carriage return.

The fourth set of questions concerns operator's console task keyins. These questions are answered with 'Y' or carriage return.

The questions in all four sets are self explanatory (see example below) except in the case of the third set. The responses to this set are numbers and need further explanation. The solicitation dialog and a detailed explanation of each response follows (for numeric responses only).

PANIC DUMP (DEV CSR OR CR)

> If the panic dump routine is desired, answer with the address of the control status register of the dump device. Legal devices are: 1) Line printer, 2) Paper tape punch, 3) Terminal devices (i.e., Teletype, VT05, or LA30).
>
> Otherwise, answer with carriage return.

NUM 8 WD BLKS IN DYNAMIC POOL (NUM OR CR)

> If the default value (32 blocks) is not desired, answer with the number of 8-word blocks that are to be allocated to the dynamic storage pool. Care must be taken to ensure the proper value of this parameter. This area is allocated in the initialization code from the top of memory down and is taken out of the last (highest) partition. Sleep queue entries (8 words), partition status tables (16 words), and operator's console buffers (72 words) are allocated from this storage pool. If there is either an overlap between the next-to-last partition and the dynamic storage pool (last partition too small or dynamic area too big) or if there is insufficient space to allocate partition status tables for resident tasks that have been linked into the system load module, the initialization routine will print out an error message and halt.

Otherwise, answer with a carriage return.

DEV TIME OUT CYCLE (NUM OR CR)

If the default value (100 milliseconds) is not desired, answer with the time interval in line frequency units between time out cycles (i.e., 6 for 100ms at 60 cycle. The time out cycle is defined as the frequency at which an attempt will be made to time out devices (i.e., a scan through the device list to find and time out active devices).

Otherwise, answer with a carriage return.

NUM QUEUE SLOTS (NUM OR CR)

If the default value (10 slots) is not desired, answer with the number of queuing slots desired. Care must be exercised in selecting this parameter. Queue slots are used to queue I/O requests, intertask messages, and tasks waiting for resources. In general there should be enough available queue slots to satisfy the dynamic requirements of the system approximately 95% of the time (may be higher in certain systems).

Otherwise, answer with a carriage return.

SZ RESOURCE ALLOC TBL (NUM OR CR)

If the default value (1 entry) is not desired, answer with the number of entries that are to be generated for the resource allocation table. The resource allocation table is used to hold group lock information of active keys and their owners. The number of entries generated for this table should be equal to the maximum number of group lock keys that are to be active simultaneously.

Otherwise, answer with a carriage return.

EXTRA TASK SLOTS (NUM OR CR)

If additional undefined (empty) task slots are desired, answer with the number of undefined (empty) task slots that are to be generated. The generation of empty task slots allocates space so tasks may be loaded into the system at a later time.

Otherwise, answer with a carriage return.

5.5 ASSEMBLING THE SYSTEM

The TXTOUT output file of GENFIL specifies the modules that must be assembled to generate the desired system (see Section 5.7 for an example of system generation).

Source module names are specified as

NAME.SRC

Where:

NAME: The name of the module.

SRC: The extension of the module name. 'SRC' signifies a source file.

All source modules must be assembled with the MACRO-11 assembler under DOS.

5.6 LINKING THE SYSTEM

The object modules produced from the assembly process must be linked according to the directions output by GENFIL to create a load module of the desired system. This is accomplished via LINK-11 under DOS.

5.7 SAMPLE SYSTEM GENERATION

System Features:

The following example illustrates the generation of the basic RSX11A system. Features of this system are:

a) The operator's console task.

b) Two extra task slots for loading new tasks into the system.

c) Multi-terminal driver (one terminal used).

d) Paper tape reader driver.

e) Paper tape punch driver.

f) Four partitions--one each for the operator's console and null tasks and two for loading new tasks into the system.

Generation Procedure:

STEP 1 = Partition table definition

The following partition definitions are inserted in the file TABLS. Two additional partitions are defined (partitions for the operator's console and null task are already defined). One partition is 512 words in length and starts at the first available memory location above the operator's console task. The other partition starts 512 words thereafter and occupies the remainder of core memory.

```
PART    0,INITL$
PART    0,INITL$+2000
```

STEP 2 = Task table definition

No tasks are to be linked into the system load module, thus the two extra task slots can be generated via the system generation parameter T$$SLT (see dialog below).

STEP 3 = Parameter file definition

The parameter file definition program GENFIL is run under DOS. The following dialog is carried out with this program. All programmer responses are underscored; if no response is indicated a carriage return was typed.

GENFIL Dialog:

```
RSX11A PARAMETER
FILE GENERATOR
V001A JAN. 12, 1973
```

All questions are to be answered with a number, 'Y(ES)', or carriage return. A number or 'Y' implies an affirmative answer and a carriage return implies a negative answer. Decimal numbers may be preceded with a plus or minus (+ is assumed), and octal numbers are designated by a preceding '#' sign. If an incorrect reply is received, the question will be repeated.

The following questions pertain to executive requests. If a particular request is desired, answer with a 'Y'; otherwise, answer with a carriage return.

```
ACTIVATE-SUSPEND (Y OR CR)
*

REQ-RLS BUFFER BLOCK (Y OR CR)
*Y

CANCEL TIMED INT (Y OR CR)
*

DELETE TASK (Y OR CR)
*Y

DEFINE LIST (Y OR CR)
*

END ACTION RETURN (Y OR CR)
*

END ACTION WAIT (Y OR CR)
*

TASK GROUP LOCK REQS (Y OR CR)
*

LIST MANIPULATION (Y OR CR)
*

RECEIVE MESSAGE (Y OR CR)
*Y
```

REQUEST DATE (Y OR CR)
*

REQ TIMED INT (Y OR CR)
*

REQ ASYNC EXEC (Y OR CR)
*

REQ TASK EXECUTION (Y OR CR)
*Y

REQ I-O OPERATION (Y OR CR)
*Y

REQ SYNC EX (Y OR CR)
*

REQ TOD EXEC (Y OR CR)
*

REQ TIME OF DAY (Y OR CR)
*

SET ALTERNATE STACK (Y OR CR)
*

SEND MESSAGE (Y OR CR)
*

SET ERROR TRAP ADDR (Y OR CR)
*Y

SET TRAP TRAP ADDR (Y OR CR)
*

REQ TIMED WAIT (Y OR CR)
*

The following questions pertain to I-O drivers. If a particular driver is desired, answer with a 'Y' or number (whichever is appropriate); otherwise, answer with a carriage return.

AD01-D A-D CONVT (Y OR CR)
*

AFC11 A-D CONVT (Y OR CR)
*

TERMINALS (NUM-1 OR CR)
*0

PC11 PAPER PUNCH (Y OR CR)
*Y

PC11 PAPER READER (Y OR CR)
*Y

RC11 DISK (Y OR CR)
*

```
RF11 DISK (Y OR CR)
*

RK11 DISK (Y OR CR)
*

UDC11 (NUM MODULES OR CR)
*

LP11 LINE PRINTER (Y OR CR)
*
```

The following parameters define table sizes or functional capabilities. If a particular feature is desired, answer with a 'Y' or number (whichever is appropriate); otherwise, answer with a carriage return.

```
PANIC DUMP (DEV CSR OR CR)
*

NUM 8 WD BLKS IN DYNAMIC POOL (NUM OR CR)
*15

DEV TIME OUT CYCLE (NUM OR CR)
*6

NUM QUEUE SLOTS (NUM OR CR)
*10

SZ RESOURCE ALLOC TBL (NUM OR CR)
*0

EXTRA TASK SLOTS (NUM OR CR)
*2

DISK RES TASK SUPPORT (Y OR CR)
*

EAE SUPPORT (Y OR CR)
*

CONSOLE TASK (Y OR CR)
*Y
```

The following questions pertain to console keyins. If a particular keyin is desired, answer with a 'Y'; otherwise, answer with a carriage return.

```
ACTIVATE TASK (Y OR CR)
*

BREAKPOINT TRAP (Y OR CR)
*

ENTER DATE (Y OR CR)
*

DEPOSIT MEMORY (Y OR CR)
*Y
```

```
DELETE TASK (Y OR CR)
*Y

EXAMINE MEMORY (Y OR CR)
*Y

LOAD TASK (Y OR CR)
*Y

REQ ASYNC TASK EXEC (Y OR CR)
*

REQ TASK EXECUTION (Y OR CR)
*Y

REQ SYNC TASK EXEC (Y OR CR)
*

REQ TOD TASK EXEC (Y OR CR)
*

SUSPEND TASK (Y OR CR)
*

ENTER TIME (Y OR CR)
*

THAT'S ALL FOLKS!
```

GENFIL output:

The output that results from the above dialog is a parameter file (named CONFIG) and directions for assembling and linking the system. The following parameter file and directions would result from the above dialog.

Parameter file:

```
;       COPYRIGHT 1973, DIGITAL EQUIPMENT
;       CORP., MAYNARD, MASS 01754
;
;       D.N.  CUTLER 1-12-73
;             Version V001A
;
;       RSX11A System Parameter File
;
B$$UFR=000000
D$$FLT=000000
R$$CDV=000000
R$$QEX=000000
R$$QIO=000000
S$$ERR=000000
C$$NSL=000000
P$$APP=000000
P$$APR=000000
C$$ORE=000017
D$$CNT=000006
Q$$SLT=000012
R$$SRC=000000
T$$SLT=000002
```

```
C$$TSK=000000
K$$DEP=000000
K$$DLT=000000
K$$EXM=000000
K$$LOD=000000
K$$QEX=000000
```

Assembly directions for executive requests:

The following executive request modules must be assembled

```
BUFR<RGDEF,BUFR.SRC
DELT<CONFIG,RGDEF,DELT.SRC
ER<RGDEF,ER.SRC
RCEV<RGDEF,RCEV.SRC
RQEX<CONFIG,RGDEF,RQEX.SRC
RQIO<RGDEF,RQIO.SRC
SETERR<RGDEF,SETERR.SRC
TERM<CONFIG,RGDEF,TERM.SRC
```

Assembly directions for executive routines:

The following executive routine modules must be assembled

```
ACHCK<RGDEF,ACHCK.SRC
ALSTK<RGDEF,ALSTK.SRC
ARITH<RGDEF,ARITH.SRC
BILDS<RGDEF,BILDS.SRC
ERGEN<CONFIG,ERGEN.SRC
ERINT<CONFIG,RGDEF,ERINT.SRC
ERRORS<CONFIG,RGDEF,ERRORS.SRC
EXCEL<EXCEL.SRC
GETID<RGDEF,GETID.SRC
INITL<CONFIG,RGDEF,INITL.SRC
IOSUB<RGDEF,IOSUB.SRC
LOWCOR<CONFIG,RGDEF,LOWCOR.SRC
QUEUE<RGDEF,QUEUE.SRC
REQSB<CONFIG,RGDEF,REQSB.SRC
RQLCB<RGDEF,RQLCB.SRC
SYSUB<CONFIG,RGDEF,SYSUB.SRC
TABLS<CONFIG,RGDEF,TABLS.SRC
TIMES<CONFIG,RGDEF,TIMES.SRC
```

Assembly directions for I/O drivers:

The following I-O driver modules must be assembled

```
PTPHND<RGDEF,PTPHND.SRC
PTRHND<RGDEF,PTRHND.SRC
TTYHND<CONFIG,RGDEF,TTYHND.SRC
```

The following system tasks must be assembled

```
CONTSK<MACRS,CONFIG,RGDEF,CONTSK.SRC
NULTSK<NULTSK.SRC
```

Linking directions for this system:

The following linker command strings must be used to link the system

```
RSX11A,RSX11A<LOWCOR,ERINT
ERGEN,EXCEL,TABLS,ER
BUFR,DELT,RCEV,RQEX
RQIO,SETERR,TERM,PTPHND
PTRHND,TTYHND,ARITH,ACHCK
ALSTK,BILDS,ERRORS,GETID
IOSUB,QUEUE,REQSB,RQLCB
SYSUB,TIMES,NULTSK,CONTSK
INITL/B:0/E
```

STEP 4 = Assembly

All modules are assembled according to the above definitions.

STEP 5 = Linking

The resultant object modules are linked according to the above directions and a load module is punched out on a paper tape.

STEP 6 = Loading

The load module is loaded via the absolute loader. As soon as the system is loaded, RSX11A will type out its identification and start running the null task.

APPENDIX A

EXECUTIVE REQUEST SUMMARY

Task Termination:

a) Terminate Task Execution

Function: Terminate the execution of the requesting task.

Parameters:

None.

Macro calling sequence:

```
T$ERM
```

Assembly language generated from the macro expansion:

```
EMT     0
```

b) Delete Task

Function: Delete any other task in the same task group from the system.

Parameters:

TNAME = The address of the 4-character task name.

BUSY = The nonskip "place to go" address if the specified task is currently executing.

Macro calling sequence:

```
D$ELT   TNAME,BUSY
```

Assembly language generated from the macro expansion:

```
MOV     #TNAME,-(SP)
EMT     2
.WORD   BUSY
```

Timer:

a) Request Timed Wait

Function: Delay the execution of the requesting task for a specified interval.

Parameters:

INT = The address of a two-word time delay expressed in line frequency units.

NORM = The nonskip "place to go" address if the request cannot be accepted.

Macro calling sequence:

```
W$AIT   INT,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #INT,-(SP)
EMT     3
.WORD   NORM
```

b) Request Timed Interrupt

Function: Interrupt the execution of a task and trap to an end action address after a specified interval of time has elapsed.

Parameters:

ENDA = The end action "place to go" address that is to be trapped to.

INT = The address of a two-word time interval expressed in line frequency units.

NORM = The nonskip "place to go" address if the request cannot be accepted.

Macro calling sequence:

```
R$TINT  INT,ENDA,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #ENDA,-(SP)
MOV     #INT,-(SP)
EMT     4
.WORD   NORM
```

c) Cancel Timed Interrupt Request

Function: Cancel all timed interrupt requests that are outstanding.

Parameters:

None.

Macro calling sequence:

```
C$TINT
```

Assembly language generated from the macro expansion:

```
EMT     5
```

d) Request Time of Day

Function: Obtain the current time of day expressed as clock ticks past midnight.

Parameters:

None.

Macro calling sequence:

```
R$TOD
```

Assembly language generated from the macro expansion:

```
CMP     -(SP),-(SP)
EMT     6
```

e) Request Date

Function: Obtain the current date expressed in Julian relative to the year 1972.

Parameters:

None.

Macro calling sequence:

```
R$DATE
```

Assembly language generated from the macro expansion:

```
CMP     -(SP),-(SP)
EMT     7
```

Task Initiation:

a) Request Task Execution

Function: Request the execution of any task in the same task group.

Parameters:

TNAME = The address of the 4-character task name.

NORM = The nonskip "place to go" address if the request cannot be accepted.

Macro calling sequence:

```
R$QEX   TNAME,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #TNAME,-(SP)
EMT     10
.WORD   NORM
```

b) Request Synchronous Periodic Task Execution

Function: Request the synchronous periodic execution of any task in the same task group.

Parameters:

TNAME = The address of the 4-character task name.

INT = The address of a two-word time interval expressed in line frequency units.

NORM = The nonskip "place to go" address if the request cannot be accepted.

Macro calling sequence:

```
R$QSX   TNAME,INT,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #INT,-(SP)
MOV     #TNAME,-(SP)
EMT     11
.WORD   NORM
```

c) Request Asynchronous Periodic Task Execution

Function: Request the asynchronous periodic execution of any task in the same task group.

Parameters:

TNAME = The address of the 4-character task name.

INT = The address of a two-word time interval expressed in line frequency units.

NORM = The nonskip "place to go" address if the request cannot be accepted.

Macro calling sequence:

```
R$QAX   TNAME,INT,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #INT,-(SP)
MOV     #TNAME,-(SP)
EMT     12
.WORD   NORM
```

d) Request Task Execution at Time of Day

Function: Request the execution of any task in the same task group to begin at a specific time of day.

Parameters:

TNAME = The address of the 4-character task name.

TOD = The address of a two-word time of day expressed in line frequency units.

NORM = The nonskip "plan to go" address if the requested cannot be accepted.

Macro calling sequence:

```
R$QTX    TNAME,TOD,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #TOD,-(SP)
MOV     #TNAME,-(SP)
EMT     13
.WORD   NORM
```

Task Synchronization:

a) Suspend Task

Function: Suspend the execution (via blocking) of any task in the same task group.

Parameters:

TNAME = The address of the 4-character task name.

Macro calling sequence:

```
S$PND   TNAME
```

Assembly language generated from the macro expansion:

```
MOV     #TNAME,-(SP)
EMT     14
```

b) Activate Task

Function: Activate the execution (via unblocking) of any other task belonging to the same task group.

Parameters:

TNAME = The address of the 4-character task name.

Macro calling sequence:

```
A$CTV   TNAME
```

Assembly language generated from the macro expansion:

```
MOV     #TNAME,-(SP)
EMT     15
```

c) Test and Set Task Group Lock Return Immediate

Function: Test and set a task group lock and return control immediately regardless of whether the operation was successful.

Parameters:

KEY = An eight-bit lock identifier.

BUSY = The nonskip "place to go" address if the specified task group lock was already set when the request was executed.

Macro calling sequence:

```
T$SETI  KEY,BUSY
```

Assembly language generated from the macro expansion:

```
MOV     #KEY,-(SP)
EMT     20
.WORD   BUSY
```

d) Test and Set Task Group Lock Wait

Function: Test and set a task group lock and wait until the requesting task can set the lock.

Parameters :

KEY = An eight-bit lock identifier.

BUSY = The nonskip "place to go" address if the specified task group lock was already set and no queuing space is available.

Macro calling sequence:

```
T$SETW  KEY,BUSY
```

Assembly language generated from the macro expansion:

```
MOV     #KEY,-(SP)
EMT     21
.WORD   BUSY
```

e) Reset Task Group Lock

Function: Reset a task group lock.

Parameters:

KEY = An eight-bit lock identifier.

Macro calling sequence:

```
R$SET   KEY
```

Assembly language generated from the macro expansion:

```
MOV     #KEY,-(SP)
EMT     22
```

Intertask Communication:

a) Send Message to Task

Function: Send a two-word message to any other task in the same task group.

Parameters:

TNAME = The address of the 4-character task name.

MPKT = The address of the 2-word message packet.

NORM = The nonskip "place to go" address if the request cannot be accepted because no queuing space is available.

Macro calling sequence:

```
S$END   TNAME,MPKT,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #MPKT,-(SP)
MOV     #TNAME,-(SP)
EMT     23
.WORD   NORM
```

b) Receive Message From Task

Function: Get two-word message from requesting task's message queue.

Parameters:

NONE = The nonskip "place to go" address if there are no messages in the task's message queue.

Macro calling sequence:

```
R$CEIV  NONE
```

Assembly language generated from the macro expansion:

```
CMP     -(SP),-(SP)
EMT     24
.WORD   NONE
```

Input/Output:

a) Request I/O Operation

Function: Perform control functions and I/O transfers on peripheral devices.

Parameters:

PKT = The address of the I/O specification packet.

NORM = The nonskip "place to go" address if the request cannot be accepted because no queuing space is available.

Macro calling sequence:

```
R$QIO   PKT,NORM
```

Assembly language generated from the macro expansion:

```
MOV     #PKT,-(SP)
EMT     30
.WORD   NORM
```

End Action Control:

a) End Action Wait

Function: Wait (via blocking) until any of the requesting task's end action requests is satisfied.

Parameters:

NONE = The nonskip "place to go" address if there are no outstanding end action requests.

Macro calling sequence:

```
E$ACTW   NONE
```

Assembly language generated from the macro expansion:

```
EMT     34
.WORD   NONE
```

b) End Action Return

Function: Return control from end action coding to the interrupted place in the requesting task.

Parameters:

None.

Macro calling sequence:

```
E$ACTR
```

Assembly language generated from the macro expansion:

```
EMT     35
```

List Manipulation:

a) Define List

Function: Define a list and link all entries in the list together. Can also be used to define a buffer pool.

Parameters:

DESC = The address of the list descriptor.

Macro calling sequence:

```
D$LIST  DESC
```

Assembly language generated from the macro expansion:

```
MOV     #DESC,-(SP)
EMT     36
```

B Remove Entry From List

Function: Remove an entry from a first-in-first-out list.

Parameters:

DESC = The address of the list descriptor.

NONE = The nonskip "place to go" address if the list is empty.

Macro calling sequence:

```
R$MOV   DESC,NONE
```

Assembly language generated from the macro expansion:

```
MOV     #DESC,-(SP)
EMT     37
.WORD   NONE
```

c) Make Entry In List

Function: Make an entry in a first-in-first-out list.

Parameters:

DESC = The address of the list descriptor.

ENT = The address of the entry that is to be made in the first-in-first-out list.

Macro calling sequence:

```
M$ENT   DESC,ENT
```

Assembly language generated from the macro expansion:

```
MOV     #ENT,-(SP)
MOV     #DESC,-(SP)
EMT     40
```

Dynamic Storage:

a) Request Buffer Block

Function: Request a variable size buffer block from the system dynamic buffer pool. Only priviledged tasks may execute this ER.

Parameters:

SIZE = The size of the buffer block expressed in 8-word units.

NOBF = The nonskip "place to go" address if no buffers are available.

Macro calling sequence:

```
G$BUF   SIZE,NOBF
```

Assembly language generated from the macro expansion:

```
MOV     #SIZE,-(SP)
EMT     41
.WORD   NOBF
```

b) Release Buffer Block

Function: Release a variable size buffer block to the system dynamic buffer pool. Only priviledged tasks may execute this ER.

Parameters:

SIZE = The size of the buffer block expressed in 8-word units.

BUF = The address of the buffer that is to be released.

Macro calling sequence:

```
R$BUF   SIZE,BUF
```

Assembly language generated from the macro expansion:

```
MOV     #BUF,-(SP)
MOV     #SIZE,-(SP)
EMT     42
```

Miscellaneous:

a) Set Error Trap Address

Function: Set the address to be trapped to for ER, processor, and stack overflow errors.

Parameters:

ERRA = The address to be trapped to if an ER, processor, or stack overflow error occurs.

Macro calling sequence:

```
E$RSET  ERRA
```

Assembly language generated from the macro expansion:

```
MOV     #ERRA,-(SP)
EMT     51
```

b) Set Alternate Stack Address

Function: Set the address of an alternate stack to be used in case stack overflow occurs.

Parameters:

ASTK = The starting address of the alternate stack.

Macro calling sequence:

```
A$LSET  ASTK
```

Assembly language generated from the macro expansion:

```
MOV     #ASTK,-(SP)
EMT     52
```

c) Set TRAP Trap Address

Function: Set the address to be trapped to if a TRAP trap instruction is executed from the task.

Parameters:

TRPA = The address to trap to if a TRAP instruction is executed.

Macro calling sequence:

```
S$TRAP  TRPA
```

Assembly language generated from the macro expansion:

```
MOV     #TRPA,-(SP)
EMT     53
```

APPENDIX B

DEVICE DEPENDENT INFORMATION

All I/O packets are address checked assuming a length of 9 words. However, unless the I/O packet lies at the extreme bottom or top of the requesting task's partition, only the packet words that are required need be specified.

a) Teletypes and all Teletype Compatible Devices Interfaced Via a KL11-"N"

Physical device type:

The physical device type for Teletypes and Teletype compatible devices is 0 for one Teletype, and n-1 for more than one Teletype.

Core requirements:

I/O handler: 213 words.
Device table: 10 words per Teletype.
Queue header: 2 words per Teletype.
Device dependent: 8 words per Teletype.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. There is no device dependent packet information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is one byte and no special format is assumed.

Legal functions:

Immediate mode is legal for all functions:

0 = Read ($RD)

1 = Write ($WR)

Device dependent storage:

Each Teletype requires 8 contiguous words of device dependent storage. These words have the following format:

Word 1 - (Status)

Bit 15 - Transfer direction.

0 = Direction is output.

1 = Direction is input.

Bit 14 - Input request.

0 = No input request was made during an output operation.

1 = An input request (key was struck) was made during an output operation.

Bit 13 - End-of-line.

0 = An end-of-line character has not been typed yet during an input operation.

1 = An end-of-line character has been typed during an input operation.

Bit 12 - Console definition.

0 = Teletype is not considered a console Teletype.

1 = Teletype is considered a console Teletype.

Bits 11 to 8 - Empty and not used but must be zero.

Bits 7 to 0 - The last character input during an input operation.

Word 2 - (Item count)

Bits 15 to 0 - The number of remaining bytes for an input operation. This word is not used for output operations.

Word 3 - (Item address)

Bits 15 to 0 - The address where the next byte is to be stored for an input operation. This word is not used for output operations.

Word 4 - (Printer CSR)

Bits 15 to 0 - The address of the teleprinter control status register.

Word 5 - (Printer DBR)

Bits 15 to 0 - The address of the teleprinter data buffer register.

Word 6 - (Keyboard CSR)

Bits 15 to 0 - The address of the keyboard control status register.

Word 7 - (Keyboard DBR)

Bits 15 to 0 - The address of the keyboard data buffer register.

Word 8 - (Expansion)

Bits 15 to 0 - Empty and not used but must be zero.

Special I/O handler characteristics:

The handling of Teletypes is via half duplex software; however, the Teletypes themselves must be full duplex. If a Teletype is in output mode when a key is struck on the corresponding input keyboard, then the output is allowed to finish before the keyin is started. There is no type-ahead capability.

Carriage return is the only line terminating character that is recognized. Upon encountering a carriage return during an input operation, both a carriage return and line feed are echoed and inserted in the input buffer.

Rubout and line delete are the only implemented control functions for input operations. Rubout causes a backslash to be echoed and the last character in the input buffer to be deleted. Control U is interpreted as the line delete control character. A line delete causes no echo; whatever has been previously typed is deleted.

The position of the carriage is not checked, and therefore the horizontal tab function is not implemented. Form feed and vertical tab are also not implemented. An input buffer may be of any size; however, after 72 echoing characters have been typed, the carriage will not be returned. When a carriage return is typed the carriage is returned and the input line is terminated. If more characters are typed than will fit in the input buffer, the last character in the buffer is continuously overlayed until a carriage return is typed. When this situation occurs, the last character to appear in the input buffer is a line feed.

NOTE

KL11 interrupt vectors must be assigned starting at #300 and the device registers starting at #176500. Multiple KW11s are assigned consecutive locations thereafter.

b) Paper Tape Reader

Physical device type:

The physical device type for the paper tape reader is 1.

Core requirements:

I/O handler: 53 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. There is no device dependent packet information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is one byte and no special format is assumed.

Legal functions:

Immediate mode is legal for all functions.

0 = Read ($RD)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

c) Paper Tape Punch

Physical device type:

The physical device type for the paper tape punch is 2.

Core requirements:

I/O handler: 54 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet must be specified. There is no device dependent packet information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is one byte and no special format is assumed.

Legal functions:

Immediate mode is legal for all functions.

1 = Write ($WR)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

d) LP11 Line Printer

Physical device type:

The physical device type for the line printer is 3.

Core Requirements

I/O handler: 116 Words
Device table: 10 Words
Queue header: 2 Words

I/O packet format:

Words 0 to 5, as described for the general I/O packet must be specified. There is no device dependent packet information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is one byte and no special format is assumed.

Legal functions:

Immediate mode is legal for all functions.

1 = Write ($WR)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

e) AD01-D A/D Convertor

Physical Device Type:

The physical device type for the AD01-D A/D convertor is 4.

Core requirements:

I/O handler: 52 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. There is no device dependent information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is two consecutive words. These words have the following format:

Word 0 - (Terminal Connection)

Bit 15 - Empty and not used but must be zero.

Bits 14 to 12 - The hardware gain code of the gain to be applied to the conversion.

Bit 11 - Empty and not used but must be zero.

Bits 10 to 0 - The multiplexer channel that is to be converted.

Word 1 - (Data)

Bits 15 to 0 - Receive the converted value.

Legal functions:

Immediate mode is legal for all functions.

0 = Read ($RD)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

f) AFC11 A/D Convertor

Physical device type:

The physical device type for the AFC11 A/D Convertor is 5.

Core requirements:

I/O handler: 45 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O, must be specified. There is no device dependent information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is two consecutive words. These words have the following format:

Word 0 - (Terminal connection)

Bit 15 - Empty and not used but must be zero.

Bits 14 to 12 - The hardware gain code of the gain to be applied to the conversion.

Bit 11 - Empty and not used but must be zero.

Bits 10 to 0 - The multiplexer channel that is to be converted.

Word 1 - (Data)

Bits 15 to 0 - Receive the converted value.

Legal functions:

Immediate mode is legal for all functions.

0 = Read ($RD)

Device dependent storage:

None.

Special I/O handler characteristics:

The maximum multiplexing rate of the AFC11 is 200 channels per second; however, a single channel may only be sampled at a maximum rate of 20 points per second. Failure to observe this restriction will result in inaccurate data. The AFC11 I/O handler makes no attempt to guard against improper sampling rates. It is left to the user to ensure that this does not happen.

g) UDC11 Universal Digital Control Unit

Physical device type:

The physical device type for the UDC11 universal Digital Control unit is 6.

Core requirememts:

I/O handler: 97 words.
Device table: 10 words.
Queue header: 2 words.
Device dependent: 1 word per UDC module.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. There is no device dependent information and therefore words 6, 7, and 8 need not be specified.

Item size and format:

The item size is two consecutive words. These words have the following format:

Word 0 - (Terminal connection)

Bits 15 to 12 - The field size minus 1 of the field to be read or written if a special function is specified. If the function is a normal read or write, this field is ignored.

Bits 11 to 8 - The right most bit of the field to be read or written if a special function is specified. If the function is a normal read or write, this field is ignored.

Bits 7 to 0 - The module number of the module to be read of written.

Word 1 - (Data)

Bits 15 to 0 - This word receives the module data, right justified and zero filled, on read functions. For write functions this word contains the data, right justified, to be written.

Legal functions:

Immediate mode is legal for all functions.

0 = Full word read ($RD)

1 = Full word write ($WR)

2 = Field read ($SPRD)

3 = Field write ($SPWR)

Device dependent storage:

One word of storage is required for each module. This word is used to hold the current state of the module in core.

Special I/O handler characteristics:

The UDC11 I/O handler does not support the interrupt capability of the UDC11. If this capability is desired, the user must write his own interrupt handler. It is impossible to distinguish between input and output modules in the I/O handler. The user, therefore, must ensure that the proper functions are performed on a particular module. An attempt to write an input module results in no operation. Reading an output module results in zeros being read.

h) RK11 Cartridge Disk Control

Physical device type:

The physical device type for the RK11 cartridge disk control is 7.

Core requirememts:

I/O handler: 139 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. Words 6, 7, and 8 must also be specified and have the following format:

Word 6 - (Disk address)

Bits 15 to 0 - The starting logical sector address for I/O transfer, drive reset, write lock, and seek functions. This word need not be specified for the control reset function. RK11 disk addresses are specified as logical sector addresses starting with sector zero of drive zero and continuing to the last logical sector of drive 8. Logical sectors are always 256 words in length.

Word 7 - (Final error status)

Bits 15 to 0 - Receive the final contents of the error status register.

Word 8 - (Final control status)

Bits 15 to 0 - Receive the final contents of the control status register.

Item size and format:

The item size is one word and no special format is assumed.

Legal functions:

Immediate mode is illegal for all functions.

0 = Read ($RD)

1 = Write ($WR)

2 = Read check ($SPRD)

3 = Write check ($SPWD)

#14 = Seek

#15 = Drive reset

#16 = Write lock

#17 = Control reset

Device dependent storage:

None.

Special I/O handler characteristics:

None.

i) RF11 Fixed Head Disk Control

Physical device type:

The physical device type of the RF11 fixed head disk control is 8.

Core requirements:

I/O handler: 80 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. Words 6, 7, and 8 must also be specified and have the following format:

Word 6 - (Disk address)

Bits 15 to 0 - The starting logical disk address. RF11 disk addresses are specified as logical sector addresses starting with sector zero of platter zero

and continuing to the last logical sector on platter 8. Logical sectors are always 256 words in length.

Word 7 - (Final error status)

Bits 15 to 0 - Receive the final contents of the error status register.

Word 8 - (Final control status)

Bits 15 to 0 - Receive the final contents of the control status register.

Item size and format:

The item size is one word and no special format is assumed.

Legal functions:

Immediate mode is illegal for all functions.

0 = Read ($RD)

1 = Write ($WR)

3 = Write check ($SPWR)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

j) RC11 Fixed Head Disk Control

Physical device type:

The physical device type of the RC11 fixed head disk control is 9.

Core requirememts:

I/O handler: 80 words.
Device table: 10 words.
Queue header: 2 words.

I/O packet format:

Words 0 to 5, as described for the general I/O packet, must be specified. Word 6, 7, and 8 must also be specified and have the following format:

Word 6 - (Disk address)

Bits 15 to 0 - The starting logical disk address. RC11 disk addresses are specified as logical sector addresses

starting with sector zero of platter zero and continuing to the last logical sector of platter 8. Logical sectors are always 256 words in length.

Word 7 - (Final error status)

Bits 15 to 0 - Receive the final contents of the error status register.

Word 8 - (Final control status)

Bits 15 to 0 - Receive the final contents of the control status register.

Item size and format:

The item size is one word and no special format is assumed.

Legal functions:

Immediate mode is illegal for all functions.

0 = Read ($RD)

1 = Write ($WR)

3 = Write check ($SPWR)

Device dependent storage:

None.

Special I/O handler characteristics:

None.

APPENDIX C

ERROR MESSAGES AND MEANING

The general form of all error messages is as follows:

ERR XX SEV X TNAM

Where:

ERR XX = The error number.

SEV X = The severity of the error.

TNAM = The 4-character name of the task that was running when the error occurred.

The following error designations are defined.

Error	Severity	Meaning
#1	1	An illegal ER was executed from a task. TNAM is the name of the task.
#1	2	An ER was executed from the executive. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#2	2	A call to the set schedule request (SETRQ$) subroutine was executed which specified an illegal task index. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#3	2	A call to the execution request (TSKRQ$) subroutine was executed which specified an illegal task index. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#4	2	A request was made to delete a task whose status word claimed that this task was in the sleep queue; however, the task was not in the sleep queue. TNAM is the name of the task that made the request. The system is stopped after the error message has been printed.
#5	1	An illegal task name was specified in an ER that uses GETID to obtain the task index of the requested task. TNAM is the name of the requesting task.

#6	1	An address check violation has occurred. This violation is either due to an out-of-range address or a byte address when a word address is expected. TNAM is the name of the requesting task.
#7	1	An illegal device number was specified in a request I/O operation ER. TNAM is the name of the requesting task.
#10	1	An illegal I/O function was specified in a request I/O operation ER. TNAM is the name of the requesting task.
#11	2	A call to one of the queuing subroutines (QINSP$, QINSF$, QOUTP$, QOUTF$) was executed which specified an illegal queue number. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#12	2	A complete scan of the task list has occurred without finding a runnable task. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#13	0	An unsolicited interrupt has occurred. TNAM is the name of the task that was running when the interrupt was received.
#14	1	An illegal release of a buffer block was attempted. This may have been caused by a nonprivileged task attempting to release a buffer or a bad buffer address. TNAM is the name of requesting task.
#16	2	The system stack has underflowed. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#17	1	An illegal instruction was executed from the task TNAM.
#20	2	An illegal instruction was executed from the executive. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error. The system is stopped after the error message has been printed.
#21	1	An illegal immediate mode designation was specified in a request I/O operation ER. TNAM is the name of the requesting task.

#24	2	The UDC11 timed out. This is an impossible happening; occurrence is due to a hardware failure and the system is stopped after the error message has been printed.
#26	1	A terminate task execution ER was executed while the requesting task still had outstanding I/O operations pending. TNAM is the name of the requesting task.
#30	0	Task stack underflow has occurred due to an end action request or a processor error trap. TNAM is the name of the task that was running when the error occurred, but it is not necessarily responsible for the error.

The following error messages are typed from the console task.

INSF PRM	Insufficient parameters.
ILL PRM XX	Illegal Parameter xx.
LATER	Partition busy.
LOAD ERR	Task name already defined; task table entry occupied; undefined priority; attempt to load in wrong partition.

APPENDIX D

GENERAL QUEUING SPACE

The various queues in the system share one generalized queuing space and a corresponding set of reentrant queue manipulation routines. Queues may be ordered by priority or simply first-in-first-out. In either case, they are constructed as single linked lists. The queuing space is divided into queue headers and queuing slots. The queue headers define the individual queues, whereas the queue slots are used to form the queues themselves. A list of available queue slots is maintained as a last-in-first-out list. The head of this list is at symbolic location QAVL$. The queueing space consists of three lists. Queue headers only require entries in two of the lists, thus no space is allocated in the third list for this purpose. Queue slots, however, require entries in all three lists. The three lists have the following format:

Queue Headers

List 1 - QSPC1$ (First)

Bits 15 to 0 - The index, relative to the beginning of the lists, of the first entry in the queue. If the queue is empty, then this word is zero.

List 2 - QSPC2$ (Last)

Bits 15 to 0 - The index of the last entry in the queue. If the queue is empty, then this word contains the index of the queue header itself.

Queue Entries

List 1 - QSPC1$ (Next)

Bits 15 to 0 - The index of the next entry in the queue. If this is the last entry in the queue, then this word is zero.

List 2 - QSPC2$ (Priority or Data)

Bits 15 to 0 - If the queue is a priority ordered queue then this word contains the priority key; otherwise, this word contains 16 bits of data.

List 3 - QSPC3$ (Data)

Bits 15 to 0 - 16 bits of data.

APPENDIX E

DEVICE TABLE

The device table is used to control the transfer of information between a device and core memory. The table consists of 10 lists each of which has a one-word entry per device. Each list is addressed by device index (device number times two) to locate information pertaining to a specific device. The device dependent, as well as the device independent, routines access these lists to store temporary information and to retrieve device parameters. The 10 lists have the following format:

List 1 - DT1$ (Device Status and Functions)

Bits 15 to 8 - The current device status

-1 = Device is performing an I/O operation.

0 = Device is idle.

+1 = Device is down and out of service.

Bits 7 to 0 - Legal function mask. A bit is set for every power of 2 that is a legal function code. For example, if 0 and 3 are the only legal functions, then the function mask would be 11.

List 2 - DT2$ (Device Queue and Initial Timeout)

Bits 15 to 8 - The queue number of the primary device queue. If more than one queue is required for a particular device (i.e., Teletypes), the additional queues are addressed relative to the primary queue.

Bits 7 to 0 - Initial timeout value in device time-out units (units are system generatable).

List 3 - DT3$ (I/O Packet Address)

Bits 15 to 0 - The address of the I/O packet that defines the operation currently being performed.

List 4 - DT4$ (Device Type and Characteristics)

Bits 15 to 8 - The device type.

0 = One Teletype or Teletype compatible terminal (i.e., VT05 or VT06).

1 = Paper tape reader.

2 = Paper tape punch.

3 = LP11 Line Printer

4 = AD01-D high level A/D convertor.

5 = AFC11 A/D convertor.

6 = UDC-11 universal digital control.

7 = RK11 disk cartridge control.

8 = RF11 fixed head disk control.

9 = RC11 fixed head disk control.

Bit 7 - Queuing flag. This flag signifies whether I/O packets are to be queued before calling the device handler.

0 = Queue I/O packets before calling the device handler.

1 = Do not queue I/O packets before calling the device handler.

Bit 6 - Immediate mode flag. This flag signifies whether immediate mode I/O is legal for the device.

0 = Immediate mode I/O is legal.

1 = Immediate mode I/O is not legal.

Bits 5 to 2 - Empty and not used.

Bits 1 to 0 - The number of bytes contained in each item to be processed in an I/O operation.

0 = One byte per item (character oriented devices).

1 = Two bytes per item (word oriented devices).

2 = Four bytes per item (double word devices).

3 = Eight bytes per item (quadruple word devices).

List 5 - DT5$ (Task Index)

Bits 15 to 0 - The task index of the task for which the current I/O operation is being performed.

List 6 - DT6$ (Physical Index and Timeout)

Bits 15 to 8 - The physical index of the device. This index is used by the various device handlers to index information that is only pertinent to that type pf device.

Bits 7 to 0 - Current timeout value in device time out units. This value is decremented every device time out unit if the device is active. If the value is decremented to zero, the device is timed out.

List 7 - DT7$ (Item Count)

Bits 15 to 0 - The number of items remaining or the initial number of items in the current I/O operation.

List 8 - DT8$ (Item Address)

Bits 15 to 0 - The address of the next item or the initial item address of the current I/O operation.

List 9 - DT9$ (Handler Entry Point)

Bits 15 to 0 - The entry point address of the device handler.

List 10 - DT10$ (Timeout Entry Point)

Bits 15 to 0 - The timeout entry point address of the device handler.

APPENDIX F

TASK CONTROL TABLE

The task control table is used by the executive to control the execution of the various multiprogrammed tasks in the system. It is composed of 10 lists, each of which has a one-word entry per task. These lists hold all the necessary information about a task. Each list is addressed by the task index to locate information about a specific task. Tasks at the front of the lists have a higher priority than tasks at the end of the lists. Thus a task's index is also its priority. The 10 lists have the following format:

List 1 - TSK1$ (Task Status)

Bit 15 - Execution (Blocking)

0 = Task is not in execution.

1 = Task is in execution.

Bit 14 - Periodicity

0 = Task is not periodic.

1 = Task is periodic.

Bit 13 - I/O Wait (Blocking)

0 = Task is not in I/O wait.

1 = Task is in I/O wait.

Bit 12 - Sleep Queue (Blocking)

0 = Task is not in the sleep queue.

1 = Task is in the sleep queue.

Bit 11 - Suspend (Blocking)

0 = Task is not suspended.

1 = Task is suspended.

Bit 10 - Inhibit (Blocking)

0 = Task is not inhibited.

1 = Task is inhibited.

Bit 9 - Execution Request

0 = No outstanding execution request.

1 = Outstanding execution request.

Bit 8 - Resource Wait (Blocking)

0 = Task is not in resource wait.

1 = Task is in resource wait.

Bit 7 - Immediate Execution

0 = Task is not to be executed at startup.

1 = Task is to be executed at startup.

Bit 6 - Definition (Blocking)

0 = Task defined for this entry.

1 = Task not defined for this entry.

Bit 5 - In/Out (Blocking)

0 = Task is in core.

1 = Task is not in core.

Bit 4 - Residency

0 = Task is core resident.

1 = Task is disk resident.

Bit 3 - Mode

0 = Task is privileged.

1 = Task is not privileged.

Bit 2 - End Action Wait (Blocking)

0 = Task is not in end action wait.

1 = Task is in end action wait.

Bit 1 - The Periodic Type Bit. This bit only has meaning if bit 14 is a 1.

0 = Task is asynchronous periodic.

1 = Task is synchronous periodic.

Bit 0 - Stop Task Bit.

0 = Task is not in the process of being stopped.

1 = Task is in the process of being stopped.

List 2 - TSK2$ (Entry Address)

Bits 15 to 0 - The starting address of the task and the initial stack register contents.

List 3 - TSK5$ (Group and Message Queues)

Bits 15 to 8 - The queue number of the task's message queue.

Bits 7 to 0 - The group number of the group that the task is a member of.

List 4 - TSK6$ (Task Name)

Bits 15 to 0 - The first two ASCII characters of the task name.

List 5 - TSK7$ (Task Name)

Bits 15 to 0 - The second two ASCII characters of the task name.

List 6 - TSK8$ (Period)

Bits 15 to 0 - The low-order 16 bits of the task period in line frequency units. This word only has meaning if bit 14 of word one is a 1.

List 7 - TSK9$ (Period)

Bits 15 to 0 - The high-order 16 bits of the task period in line frequency units. This word only has meaning if bit 14 of word one is a 1.

List 8 - TSK11$ (Common and Task Partition)

Bits 15 to 8 - The partition number of the task's common data partition.

Bits 7 to 0 - The partition number of the task's execution partition.

List 9 - TSK12$ (Device)

Bits 15 to 8 - The device number of the disk that the task is stored on.

Bits 7 to 0 - Empty and not used.

List 10 - TSK13$ (Low Disk)

Bits 15 to 0 - The starting disk address of the task.

NOTE

TSK12$ and TSK13$ are only needed if nonresident tasks are being supported. If the system is assembled as a core-only system, then these lists are not allocated any space.

APPENDIX G

RESOURCE ALLOCATION TABLE

The resource allocation table is a dynamic table that is used to hold interlock keys for the various resources in the system. The size of the table is system generatable and therefore enough room may not be available to hold all resource interlock keys at any one time. Idle resources, however, require no room in the table. Only when a resource becomes active (someone is actively using it) does an entry appear in the table.

Three internal executive functions are provided to manipulate this table. These functions are carried to the user level via 6 ER's. The T$SETI, T$SETW, and R$SET ER's are provided for user specified resource interlock keys. Up to 256 interlock keys may be specified per task group. The I$INIT, I$INTW, and R$ELS ER's are provided for the system I/O devices.

The resource allocation table is composed of a list with 3-word entries. The head of the list is at symbolic location RATBL$ and each entry has the following format:

Word 1 (Type)

Bits 15 to 8 - The type of entry.

0 = Entry is empty.

1 = System device entry.

2 = Task specified entry.

Bits 7 to 0 - The format of this byte is dependent upon the value of bits 15 to 0.

If equal to zero, then this byte is also zero.

If equal to one, then this byte contains the device number of the device.

If equal to two, then this byte is the group of the task that specified the interlock key.

Word 2 (Key and Queue)

Bits 15 to 8 - The queue number of the queue assigned to the entry.

Bits 7 to 0 - The format of this byte is dependent upon the value in bits 15 to 8 of RATBl$.

If equal to zero, then this byte does not contain meaningful information.

If equal to one, then this byte is zero.

If equal to two, then this byte is a task supplied 8-bit interlock key.

Word 3 (Owner)

Bits 15 to 0 - The task index of the current owner of the interlock key. If the entry is empty this word is not meaningful.

APPENDIX H

PARTITION TABLE

The partition table defines the fixed allocation of core memory. The table consists of 3 lists, each of which has a one-word entry per partition. Up to 256 divisions (partitions) of core memory may be defined. Each division can be of any size, however, they may not overlap each other. All core resident tasks require a partition exclusively to themselves. Nonresident tasks may share a partition with other nonresident tasks. The partition table must be ordered in ascending order of core address. The 3 lists have the following format:

List 1 - AREA1$ (Status and Area Queue)

Bit 15 - The busy flag.

0 = Partition is not busy and is free to use.

1 = Partition is busy and is currently occupied by a task.

Bits 14 to 8 - Empty and not used.

Bits 7 to 0 - The queue number of the partition wait queue. This queue defines the waiting list of tasks that are waiting to be loaded into the partition.

List 2 - AREA2$ (Partition Address)

Bits 15 to 0 - The starting core address of the partition.

List 3 - AREA3$ (Partition Status Table Address)

Bits 15 to 0 - The address of the partition status table currently assigned to the partition. If zero, the no partition status table is assigned.

APPENDIX I

SLEEP QUEUE

The sleep queue is a linked list of tasks ordered by the time of day that they are to become eligible to run on the processor. This list is used to implement periodic task execution, time of day task execution, timed interrupt, and timed wait. A pointer to the head of the list is contained at symbolic location SLQL$. Core space for the sleep queue is obtained dynamically from the system buffer pool. An 8-word buffer is used for each entry. Sleep queue entries have the following format:

Word 1

Bits 15 to 0 - The address of the next entry in the list. If this is the last entry in the list, this word is zero.

Word 2

Bits 15 to 0 - The task index of the pertinent task.

Word 3

Bits 15 to 0 - The type of entry

0 = Timed wait ER.

1 = Time interrupt ER.

2 = Asychronous periodic task request.

3 = Synchronous periodic task request.

4 = Time of day task request.

Word 4

Bits 15 to 0 - The low-order 16 bits of the time of day that the request is to occur.

Word 5

Bits 15 to 0 - The high-order 16 bits of the time of day that the request is to occur.

Word 6

Bits 15 to 0 - The end action "place to go" address if the entry is a type 1 entry. Otherwise this word does not contain meaningful information.

Word 7

Bits 15 to 0 - The address of the partition status table if the entry is a type 1 entry. Otherwise this word does not contain meaningful information.

APPENDIX J

PARTITION STATUS TABLE

The partition status table is a 16-word block (optional 24 if 16 I/O channels are selected) of contiguous memory that is used to store information about the task that is currently resident in a partition. A partition status table is allocated only for those partitions that currently have tasks resident in them. The partition tables for partitions that are utilized by nonresident tasks are allocated dynamically from the dynamic storage pool. The partition status table for a particular partition is pointed to by the corresponding partition entry in AREA3$ in the partition table (see Appendix H). A partition status table has the following format:

Word 1 ($SSK)

Bits 15 to 0 - The suspended stack address of the task that is currently in the partition.

Word 2 ($TRP)

Bits 15 to 0 - The address to trap if a TRAP instruction is executed. This address is initially set to point to the illegal instruction exit and may be reset to a task address via the set trap trap ER.

Word 3 ($ASK)

Bits 15 to 0 - The alternate stack address to be used if stack underflow is detected. This address is initially set to the initial stack address of the task. The address may be set to point to an alternate stack area via the set alternate stack ER.

Word 4 ($IRQ)

Bits 15 to 0 - The total number of outstanding I/O requests that have been made by the task.

Word 5 ($ERQ)

Bits 15 to 0 - The total number of outstanding end action requests that have been made by the task.

Word 6 ($FAR)

Bits 15 to 0 - The address of the first argument on the user's stack. This word is used by the ER code to find the top of the user stack on errors.

Word 7 ($ERR)

Bits 15 to 0 - The address of the task's error handling routine for severity 1 errors. This address is initially set to point to the system error exit but may be set by a user task via the set error address ER.

Word 8 ($EAE)

Bits 15 to 0 - If an EAE is present, this word is used to store the shift count register when the task is not running. Otherwise, it is not used.

Word 9 ($EAE+2)

Bits 15 to 0 - If an EAE is present, this word is used to store the multiplier/quotient register when the task is not running. Otherwise, it is not used.

Word 10 ($EAE+4)

Bits 15 to 0 - If an EAE is present, this word is used to store the accumulator register when the task is not running. Otherwise, it is not used.

Word 11 ($RSQ)

Bits 15 to 8 - Empty and not used.

Bits 7 to 0 - The queue number of the resource wait queue the task is currently in. If the task is not in resource wait, this byte is empty and not used.

Words 12 to 15

Bits 15 to 0 - Empty and not used.

Word 16 ($LST)

Bits 15 to 8 - If only 8 I/O channels are selected, this byte is empty and not used. Otherwise, this byte contains the status of I/O channels 8 thru 15. A one bit signifies an assigned channel. Bits are set only for those channels that have been assigned.

Bits 7 to 0 - The status of I/O channels through 7. A one bit signifies an assigned channel. Bits are set only for those channels that have been assigned.

Words 17 to 24 (or 32) ($CHN)

Bits 15 to 8 - The number of outstanding I/O requests for the channel.

Bits 7 to 0 - The device number of the device assigned to the I/O channel.

NOTE

I/O channel capability is not implemented and therefore words 17 to 24 are neither allocated nor used.

APPENDIX K

OPERATOR'S CONSOLE COMMAND DATA BLOCK

Commands that are recognized by the operator's console task are each described by a 6-word data block. These data blocks are generated via the keyin definition macro KEY (see operator's console task). Each command data block has the following format:

Word 1 (Name)

Bits 15 to 0 - The first two ASCII characters of the command mnemonic.

Word 2 (Name)

Bits 15 to 0 - The second two ASCII characters of the command mnemonic.

Word 3 (Mask)

Bits 15 to 0 - The parameter type mask. Each bit from right to left corresponds to one parameter. A one bit indicates a numeric parameter and a zero bit an alphanumeric parameter.

Word 4 (Max,Min)

Bits 15 to 8 - The minimum number of parameters that are required by the command.

Bits 7 to 0 - The maximum number of parameters that are allowed by the command.

NOTE

The absolute maximum number of parameters for all keyins is 16.

Word 5 (Code,Flag)

Bits 15 to 8 - If the keyin translates directly into an ER, this byte contains a flag to indicate whether or not the ER has a skip return. Zero indicates no skip return whereas nonzero indicates a skip return. If the keyin does not translate directly into an ER, this byte may be used to contain auxiliary data. The data will appear in register R1 when the command execution routine is entered.

Bits 7 to 0 - If the keyin translates directly into an ER, this byte contains the ER function code. Otherwise, this byte may contain auxiliary data. The data will appear in register R2 when the command execution routine is entered.

Word 6 (Address)

Bits 15 to 0 - The starting address of the command execution routine.

APPENDIX L

TASK STACK FRAMES

Each task under RSX11A has it's own stack area. This is accomplished via software multiplexing of the stack register (R6). When the processor is switched from one task to another, the stack register is loaded with the stack address of the new task. Control is then given to the task with the stack register pointing to the top of its stack.

The task stack is used to save the context of an active task when it is not in control of the central processor. This context consists of the general registers R0 thru R5 and the task's PS and PC words. The stored context has the following format:

SP+16	The processor status word (PS).
SP+14	The program counter (PC).
SP+12	Register R0.
SP+10	Register R1.
SP+8	Register R2.
SP+6	Register R3.
SP+4	Register R4.
SP+2	Register R5.
SP+0	A zero word.

A task must ensure that 9 stack words are always available to store its context (i.e., a higher priority task could become runnable at any time). The actual value of the stack register is stored in the task's partition status table.

Executive requests also cause information to be stored on the task stack. A minimum of 19 stack words must be reserved by the task for this purpose (see Executive Requests). These stack words are used to store the task's and the executive's registers while the ER is executing. These stack words have the following format:

SP+40	Second ER parameter if required.
SP+38	First ER parameter if required.
SP+36	The processor status word of the task (PS).
SP+34	The program counter of the task (PC). (address of ER plus 2).
SP+32	Task register R0.

SP+30	Task register R1.
SP+28	Task register R2.
SP+26	Task register R3.
SP+24	Task register R4.
SP+22	Task register R5.
SP+20	A zero word (used to remove arguments on exit to task).
SP+18	A pointer to the previous first argument if nested ER (normally zero).
SP+16	The processor status word of the executive (PS).
SP+14	The program counter of the executive (PC).
SP+12	Executive register R0.
SP+10	Executive register R1.
SP+8	Executive register R2.
SP+6	Executive register R3.
SP+4	Executive register R4.
SP+2	Executive register R5.
SP+0	A zero word.

NOTE

The 9 stack words needed to store the task's context and the 19 stack words needed to execute ER's overlap (i.e., 9+19 words are not required).

APPENDIX M

SOFTWARE CONFIGURATION PARAMETERS

Software configuration parameters allow for generation of many different software/hardware configurations. These parameters can be classified into four groups; 1) Those that select executive requests, 2) Those that define table sizes and additional functional capabilities, 3) Those that select I/O drivers, 4) Those that select operator's console keyins.

All configuration parameters are denoted by symbols whose second and third characters are both dollar signs. The following describes the classes of parameters and the parameters within each class.

1) Parameters That Select Executive Requests

Parameters in this class need only be defined to select the required service. Each parameter is listed below along with the subsection(s) in which the request(s) is defined.

A$$CSP - Activate and suspend task execution (2.2.4.1 and 2.2.4.2).

B$$UFR - Dynamic storage allocation (2.2.9.1 and 2.2.9.2).

C$$INT - Cancel timed interrupt (2.2.2.3).

D$$ELT - Delete task from system (2.2.1.2).

D$$LST - Define list structure (2.2.8.1).

E$$ACR - End action return (2.2.7.2).

E$$ACW - End action wait (2.2.7.1).

G$$LCK - Task group lock manipulation (2.2.4.5).

L$$IST - Make/remove entry from list (2.2.8.2 and 2.2.8.3).

R$$CEV - Receive message from task (2.2.5.2).

R$$DAT - Request current date (2.2.2.5).

R$$INT - Request timed interrupt (2.2.2.2).

R$$QAX - Request asynchronous periodic task execution (2.2.3.3).

R$$QEX - Request task execution (2.2.3.1).

R$$QIO - Request I/O operation (2.2.6.1).

R$$QSX - Request synchronous periodic task execution (2.2.3.2).

R$$QTX - Request task execution at time of day (2.2.3.4).

R$$TOD - Request time of day (2.2.2.4).

S$$ASK - Set alternate stack address (2.2.10.2).

S$$END - Send message to task (2.2.5.1).

S$$ERR - Set error trap address (2.2.10.1).

S$$TRP - Set TRAP trap address (2.2.10.3).

W$$AIT - Request timed wait (2.2.2.1).

2) Parameters That Select I/O Drivers

Specific I/O drivers are selected by the definition of the corresponding parameter.

A$$D01 - AD01-D A/D convertor driver.

A$$FC1 - AFC11 A/D convertor driver

C$$NSL - Multi-terminal driver. This parameter must be defined to the number of terminals minus one.

L$$P11 - LP11 line printer driver

P$$APP - PC11 paper tape punch driver.

P$$APR - PC11 paper tape reader driver.

R$$CDS - RC11 disk controller driver.

R$$FDS - RF11 disk controller driver.

R$$KDS - RK11 disk controller driver.

U$$DC1 - UDC11 Universal digital control unit. This parameter must be defined to the number of UDC11 functional modules.

3) Parameters That Define Table Sizes And Additional Functional Capabilities

In general these parameters must be defined to a specific value. However, D$$ISC and E$$EAE need only be defined to select the desired capability.

D$$BG1 - The panic dump routine. This parameter must be defined to the CSR of the device that is to receive panic dump output (i.e., #177564 for console terminal, #177554 for PC11 paper tape punch, #177514 for the LP11 line printer, etc).

NOTE

Dumps may not be output to block oriented devices.

C$$ORE - The number of 8-word blocks that are to be allocated for dynamic storage allocation. The space for these blocks is allocated from the top of memory down by the

initialization code and is physically taken out of the last partition that has been defined in the partition table. The default value is 32 blocks (256 words).

D$$CNT - The device time out cycle count. This parameter must be defined to the number of line frequency units between device time out periods. The default value is 6 (100 milliseconds).

Q$$SLT - The number of general queuing space storage slots. The default value is 10 slots.

R$$SRC - The number of entries in this resource allocation table. This parameter need not be defined if the task group lock manipulation requests have not been selected.

T$$SLT - The number of extra task slots that are to be allocated in the task table so new tasks may be loaded into the system on-line.

D$$ISC - Selects a core-disk task system.

E$$EAE - Selects support of the extended arithmetic unit.

C$$TSK - Selects console task.

4) Parameters That Select Operator's Console Keyins

Parameters in this class need only be defined to select the desired keyin. Each parameter is listed below along with the subsection in which the keyin is defined

K$$ACT - Activate task execution (3.2.9).

K$$BRK - Execute breakpoint trap (3.2.13).

K$$DAT - Enter date (3.2.3).

K$$DEP - Deposit memory (3.2.2).

K$$DLT - Delete task (3.2.11).

K$$EXM - Examine memory (3.2.1).

K$$LOD - Load task (3.2.12).

K$$QAX - Request asynchronous periodic task execution (3.2.6).

K$$QEX - Request task execution (3.2.5).

K$$QSX - Request synchronous periodic task execution (3.2.7).

K$$QTX - Request task execution at time of day (3.2.8).

K$$SPN - Suspend task execution (3.2.10).

K$$TIM - Enter time of day (3.2.4).

APPENDIX N

PANIC DUMP ROUTINE

The panic dump routine (PANIC) is selected by the configuration parameter D$$BG1 and resides in lower memory just above the system stack and interrupt vectors. Its purpose is to provide formatted octal dumps on the device specified by D$$BG1.

The panic dump routine is called in two ways: 1) detection of a fatal system error, 2) a forced entry via the console switches (at symbolic address PAN$). On entry the current processor status and general registers R0 through R6 are saved in an internal save area. Register R6 is then loaded with the address of a temporary stack and a RESET followed by a HALT instruction is executed.

The panic dump routine receives its dump limits via the switch register. After the initial halt, the low dump address is entered in the switch register, CONT is depressed, the high dump address is entered in the switch register and CONT is again depressed. The desired dump will commence on the specified device (D$$BG1). When the dump is finished, the panic dump routine again executes a halt in anticipation of another set of dump limits.

The first line of output is always the contents of the processor status word and general registers R0 through R6 at the time the panic dump routine was called. The first word on this line is the address of the internal save area and should be ignored.

All dump lines have the same format including the line with the processor status and general registers. An output line is identified by its starting address followed by the contents of eight consecutive memory locations. Each location is edited as a word and as two bytes.

The following example illustrates the use of the panic dump routinte.

Dump memory from location 0 to location 2000:

Step 1 - Halt the processor.

Step 2 - Enter the address of PAN$ in the switch register and depress LOAD ADDRESS and then START. The processor will immediately halt.

Step 3 - Enter 0 in the switch register and depress CONT. The processor will again halt.

Step 4 - Enter 2000 in the switch register and depress CONT. The dump will commence on the specified device.

APPENDIX O

SPECIFICATION FOR 8K SYSTEM

An 8K core-only RSX11A(DEC-11-IRSAA-A-LA, -PA)System is available from the Software Distribution Center. This system includes all the executive routines previously described. I/O handlers are:

Device	Number
1 Teletype -	0
Paper Tape Reader -	1
Paper Tape Punch -	2
LP11 Line Printer -	3
AD01-D A/D Convertor -	4
AFC11 A/D Convertor -	5
1 UDC11 Universal Digital Control Unit -	6

The Panic Dump is handled by the line printer. All of the functions of the console Teletype are enabled.

In addition to the console Teletype task, four tasks may be installed on-line and four paritions are defined with the following starting addresses. (Partition number one is reserved for the console task.)

Partition 2: 21744
Partition 3: 25744
Partition 4: 31744
Partition 5: 35744

Other table sizes and capabilities are:

Number of 8-word blocks in dynamic pool: 50
Device time out cycle: 100 milliseconds
Number of queue slots: 20
Size of resource allocation table: 2

APPENDIX P

I/O HANDLER INTERFACE

The device handlers and the RSX11A system interface with one another through the device tables (in module TABLES). The device tables consist of ten lists with one entry in each list for each device. Information for the device table is supplied at assembly time through the DEVICE macro instruction and at run time from the parameters for the I/O request, (e.g., the I/O packets; see section 2.2.6).

The macro DEVICE will generate entries into the device table. DEVICE is of the form:

DEVICE HENT,IENT,TENT,DLBL,VECT,FMSK,TIME,FLGS,TYPE,INDX

where

HENT is the I/O handler entry point.

IENT is the I/O handler interrupt point.

TENT is the timeout entry point.

DBL is the global device index or label name for the word index into the device table.

VECT is the interrupt vector address.

FMSK is the legal function mask. A bit is set for each power of 2 that is a legal function. For example, if 0 and 3 are the only legal functions, the mask would be 11.

TIME is the initial timeout value in number of units (units specified at system generation time).

FLGS is a numeric device type (not currently used).

INDX is the physical index of the device. This index is used by various device handlers to index information pertinent to that type of device.

Device handlers are called from module RQIO, request I/O, with R0 containing the word index for that device into the 10 device table lists. A detailed descritpion of the device table lists is given in Appendix E.

At the initial entry to the device handler a JSR PC,GTPK1$ is performed. Global routine GTPK1$ transfers information from the I/O packet to the device tables. Specifically, GTPK1$ performs the following functions:

1. Checks for device busy. If busy, exits calling location+2.

2. Checks queue for I/O request for this device. If none, exits calling location+2.

3. Saves packet address, task index, and sets initial timeout value.

4. Moves the immediate item count and item address from the I/O packet to the device table lists.

5. Sets device index and exits calling location+4.

The I/O handler performs the I/O operation. After I/O has been completed (or attempted, but in error), a call is made to global subroutine IOEND via the PC. The entry conditions are: R0 contains the final packet status and R1 contains the device index.

R0 = 0	The I/O operation was completed without error.
R0 = 2	I/O operation ended in error.
R0 = 3	I/O operation timed out before being completed.
R0 = 4	Specified device is down.

IOEND performs the following functions:

1. Sets the device to idle in the device table.

2. Sets the packet status from R0.

3. Clears I/O hold or end action wait bits and reschedules tasks.

INDEX

Activate task, 2-18
  keyin, 3-8
Address for alternate stack, 2-42
Alphanumeric parameters, 3-2
Alternate stack address, 2-42
Array addressing, 1-9
Assembling system, 5-7

Blocking bit, 2-17
Breakpoint trap (keyin), 3-11
Buffer blocks, 2-38, 2-39
Buffer pool, 2-34

Calling sequence for Executive request, 2-3
Cancel timed interrupt request, 2-9
Carriage return, B-3
Command parameters, 3-2
Commands at console, 3-1
Commands recognized by operator's console Task, K-1
Command syntax, 3-2
Common data partition (definition), 1-3
CONFIG file, 5-5
Console keyin parameter, M-3
Console Task, operator's, 3-1
Context, 2-33
Control functions, B-3
Conventions in manual, 1-9
Core memory, H-1
Core residency, 1-7, 1-9
  definition of , 1-2

Data block, K-1
Date keyin command, 3-4
Decimal fields, 3-2
Decimal numbers, 1-9
Define list, 2-33
Definitions of terms, 1-2
Delayed execution (definition), 1-2
Delete task (keyin), 3-9
Deposit memory (keyin), 3-4
Device dependent information, B-1
Device index, E-1
Device table, E-1
Disk address (definition), 1-3
Disk device (definition), 1-3
Disk residency, 1-9
  definition of, 1-2
Dollar sign ($) (writing convention), 1-9
Dynamic storage pool, 2-38, 2-39, 5-6

8K system specification, O-1
End action control, 2-32
End action purpose, 2-26
End action return, 2-33
End action wait, 2-32
End action wait ER, 2-28
Enter date (keyin), 3-4
Enter time (keyin), 3-5
Entry address (definition), 1-3
ER Descriptions, 2-4
  delete Task from system, 2-5
  dynamic storage, 2-38
  end action control, 2-32
  Input/Output, 2-25
  intertask communication, 2-23
  list manipulation, 2-33
  miscellaneous, 2-40
  Task initiation, 2-11
  Task synchronization, 2-17
  Task termination, 2-4
  timer, 2-6
  also see Executive requests
Error handling, 1-8
  for keyin, 3-2
Error messages, C-1
  typed from console Task, C-3
Errors that cause traps, 2-40
  trap address, 2-40
Examine memory (keyin), 3-3
Executive partition (definition), 1-3
Executive request, 2-11
  flag, 2-4
Executive communication, 2-1
Executive request parameters, M-1
Executive request summary, A-1
Executive requests (ER's), 2-1
  calling sequence, 2-3
  parameters, 2-3

Features of RSX-11A, 1-1, 1-6
Fixed allocation of core memory, H-1
Fixed priority of scheduling, 1-6
Four steps to system generation, 5-1
Functional capabilities parameter, M-2
Functions provided by operator's console task, 3-1

GENFIL program, 5-1, 5-6

Hardware configurations, 1-1
  minimum, 1-2
Hardware/software configuration, 5-1
"Hold" I/O operations, 2-25

Idle resources, G-1
Immediate execution (definition), 1-2
Index (definition), 1-9
Index of Task, F-1
Initiation request, 2-11
Input/Output, 2-25
Internal executive functions, G-1

Interruptible services, 2-1
Intertask communication, 2-23
I/O driver request parameters, M-2
I/O facility, 1-8
I/O handler interface, P-1
I/O hold, 2-26
I/O packet, 2-25

Keyin commands
  Activate Task, 3-8
  Breakpoint trap, 3-11
  Delete Task, 3-9
  Deposit memory, 3-3
  Enter date, 3-4
  Enter time, 3-5
  Examine memory, 3-3
  Load Task, 3-9
  Request asynchronous periodic Task execution, 3-6
  Request synchronous periodic Task execution, 3-7
  Request Task execution, 3-6
  Request Task execution at time of day, 3-7
  Suspend Task, 3-8
Keyins at console, 3-1
Key to resource, 2-19

Last-in-first-out, 2-26
Line terminating character, B-3
Linking system, 5-8
List descriptor, 2-34
List manipulation, 2-33
Lists, E-1, F-1
Load Task (keyin), 3-9
Lock ER's, 2-19

Macro definition, 2-4
Make entry in list, 2-37
Message queue (definition), 1-3
Modularity, 1-7
Multiple level buffering, 2-27
Multiprogramming, 1-6
Multi-user environment, 1-7

Noninterruptible services, 2-1
Nonperiodic (definition), 1-2
Nonprivileged tasks (definition), 1-3
Nonskip return, 2-3
Numbers, writing convention for, 1-9
Numeric parameters, 3-2

Octal fields, 3-2
Octal numbers, 1-9
Operator communication, 3-1
Operator's console command data block, K-1
Overhead, minimum, 1-7

Panic dump routine, 5-6, N-1
Parameter fields, 3-2
Parameter file generation, 5-5
Parameters
  for ER's, 2-3
  that define table sizes and additional functional capabilities, M-2
  that select executive requests, M-1
  that select I/O drivers, M-2
  that select operator's console keyins, M-3
Partitions, 1-4
  definition of, 1-3
Partition status table, J-1
Partition table definition, 5-1, 5-4
Partition table, H-1
Periodic (definition), 1-2
Periodic task execution request
  asynchronous, 2-14
  synchronous, 2-13
Peripherals, 1-1
"Place to go" address, 2-3, 2-25
Pound sign (#), 1-9
Priority of Task, F-1
Privileged tasks (definition), 1-3

Queue headers, D-1
Queue slots, 5-7, D-1
Queuing space, D-1

Receive message from Task, 2-24
Reentrancy, 2-28
Reentrant services, 2-1
Release buffer block, 2-39
Remove entry from list, 2-36
Request asynchronous periodic Task execution, 2-14
  keyin, 3-6
Request buffer block, 2-38
Request date, 2-11
Request I/O operation, 2-25
Request synchronous periodic task execution, 2-13
  keyin, 3-7
Request task execution, 2-11
  keyin, 3-6
Request Task execution at time of day (keyin), 3-7
Request timed interrupt, 2-8
Request timed wait, 2-6
Request time of day, 2-10
Reset Task group lock, 2-22
Residency, core and disk, 1-9
Resource allocation table, 5-7, G-1
Resources, idle, G-1
Responses to GENFIL, 5-5

Sample Tasks, 4-1
Send message to Task, 2-23
Serially reusable services, 2-1
Set alternate stack address, 2-42
Set error trap address, 2-40
Set TRAP address, 2-43

Skip return, 2-3
Sleep queue, I-1
Software configuration parameters, M-1
Special I/O handler characteristics, B-3
Stack area, L-1
  overflow, 2-40
  underflow, 2-26
Status table, position, J-1
Summary of Executive requests, 2-1, A-1
Suspend Task, 2-17
  keyin, 3-8
Synchronous periodic task execution, 2-13
System error, fatal, N-1
System generation, 5-1
  example, 5-8
System specification, 8K, O-1

Table size parameters, M-2
Task control table, F-1
Task (definition), 1-2
Task errors, 1-8
Task execution request,
  asynchronous, 2-14
  synchronous, 2-13
Task groups, 1-4
  definition, 1-3
Task initiation, 2-11
Task name (definition), 1-3
Task slots, 5-7
Task stack frames, L-1
Task synchronization, 2-17
Task table definitions, 5-2, 5-5
Task termination, 2-4
Terminate Task execution, 2-4
Terminology (definition), 1-2
Test and set Task group lock return, immediate, 2-19
Test and set Task group lock wait, 2-21
Timed wait ER, 2-27
Time keyin command, 3-5
Time out cycle, 5-7
Timer, 2-6
TRAP trap address, 2-43
TXTOUT file, 5-5

Underflow stack, 2-26
Unrunnable task, 2-17

DEC-11-IRSAA-A-D
RSX-11A Programmers
Reference Manual

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Repcrt (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

Did you find errors in this manual? If so, specify by page.

____
____
____
____
____
____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____
____
____
____
____
____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State ____________ Zip Code or Country ____________

If you do not require a written reply, please check here. ☐

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Services
Programming Department
Maynard, Massachusetts 01754

digital

DIGITAL EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS 01754