TEXAS INSTRUMENTS

# 9900

# The Realtime Executive

MICROPROCESSOR SERIES™

## User's Guide

## IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time to improve design and to supply the best posible product for the spectrum of users.

Microprocessor Pascal System (TMSW753P or TMSW754P) is copyrighted by Texas Instruments Incorporated, and is sole property thereof. Use of this product is defined by the license agreement SC-1 between the customer and Texas Instruments. The software may not be reproduced in any form without written permission of Texas Instruments. However, application run-time packages generated from the Microprocessor Pascal System may be reproduced for resale exclusively by the customer purchasing the Microprocessor Pascal System.

All manuals associated with Microprocessor Pascal (MP351) are printed in the United States of America and are copyrighted by Texas Instruments Incorporated, all rights reserved. No part of these publications may be reproduced in any manner, including storage in a retrieval system or transmittal via electronic means, or other reproduction in any form or by any method (electronic, mechanical, photocopying, recording, or otherwise) without prior written permission of Texas Instruments Incorporated.

Information contained in these publications is believed to be accurate and reliable. However, responsibility is neither assumed for its use, nor for any infringement of patents or rights that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

COPYRIGHT, TEXAS INSTRUMENTS INCORPORATED, 1981

TABLE OF CONTENTS

SECTION 1: OVERVIEW



SECTION 2: Rx CONCEPTS



SECTION 3: Rx CONVENTIONS

## SECTION 4: Rx ROUTINES

## SECTION 5: CHANNEL ROUTINES



## SECTION 6: CONFIGURING TARGET SYSTEMS FOR OBJECT CODE EXECUTION



## SECTION 7: THE Rx STANDALONE DEBUGGER

SECTION 8: DEBUGGING THE TARGET APPLICATION WITH AMPL

## APPENDICES



## LIST OF ILLUSTRATIONS



## LIST OF TABLES

# SECTION I

# OVERVIEW

## 1.1 GENERAL

The Realtime Executive (Rx) is a standalone executive designed to support software applications executing on the 9900 family of microprocessors. The information in this manual is specifically oriented toward the assembly language user; the Microprocessor Pascal System User's Guide describes the MPP interface to Rx.)

A software environment denotes software constructs, routines, structures and all associated data and related software that permit system operation. The environment normally familiar to the assembly language programmer supports a single program executing sequentially. This program has the undivided attention of the central processor; it runs from beginning to end without interruption. Rx, however, supports multiprogramming. Independent sites of execution (processes) exist within a single environment and share a single processor. Execution of one process may be interrupted by the executive when another, more urgent, process is ready to execute. There are various constructs within Rx which support multiprogramming and automatically participate in helping the user application to run.

Rx can be thought of as a "configurable" executive; a "do it yourself" kit that allows the user to build an executive to fit user application needs by allowing selection of only those modules needed to execute the task at hand. The user can also take advantage of a variety of data structures and routines supplied by the executive which are usually not available to the low-level language application programmer. These routines support concurrency and reentrancy in code.

This section introduces the user to the Realtime Executive, its features and benefits. Section 2 discusses the Rx concepts that are necessary for the user to understand in order to use the product, while Section 3 builds on that information to enable the user to write an Rx routine. In addition, an example, "DEMOPGM" is offered to give the Rx user some early "hands-on" experience. Standard Rx routines are listed and explained in Section 4, with pertinent examples, while Rx Channel routines are detailed in Section 5. The configuration of the target system and creation of an object code load module are covered in Section 6. Target debugging explanations are provided for both the Rx Standalone Debugger (Section 7), and the AMPL Debugger (Section 8). The codes and remedial actions, routine templates, size breakdown, and assembler listing and link map for the Demo Program.

## 1.2 PRODUCT FEATURES

Short descriptions of the features available in Rx are presented below:

- o Processor Management (including concurrent process execution with preemptive priority scheduling)
- o Interrupt processing and control
- o Memory Management (including Stack and Heap management)
- o Inter-process communication capability
- o Real-time clock servicing
- o Semaphore creation/management
- o Dynamic process creation
- o Operator communications input/output
- o Debugger support
- o Configurability (to conserve memory requirements)
- o Reentrancy in code
- o Compatibility with Texas Instruments' device independent I/O Subsystems

## 1.3 Rx AS THE DRIVER OF COMPONENT SOFTWARE

Rx is associated with the Texas Instruments' 9900 family of component software. This family consists of a variety of individual software products that can be separately purchased and linked to the user's application.

Compatible with the spectrum of Texas Instruments' component software, Rx acts like a software bus, "driving" the application. Various component pieces such as Texas Instruments' Math Package, File Manager, and Data Communications packages can be added and used in the application. (Because of the modularity of the routines comprising these products, the load module produced to run on a target will include only those "pieces" of the component that are required by the application.)

## 1.4 AN Rx APPLICATION

Although several concepts will be new to some users, a simple example can be used to illustrate the benefits of Rx. Consider a

factory that needs a system to control both a manufacturing process and an accounting department. This factory has a TM990/101M CPU with a TM990/206 memory board and a TM990/305 I/O board. Within the factory, many machines must be controlled simultaneously, some interrelated and some not (see Figure 1-1). A standard operating system approach to control these many machines would be hard to implement and inefficient to use. A better way to approach the problem of control involves writing a separate piece of code for every machine process, each executing as required. Communication between these processes is necessary to ensure that each separate step associated with the manufacture of a product and the keeping of factory books is carried out within the correct time frame. A modular executive that supports multiprogramming will ease such software design. Rx supplies all of these executive features.

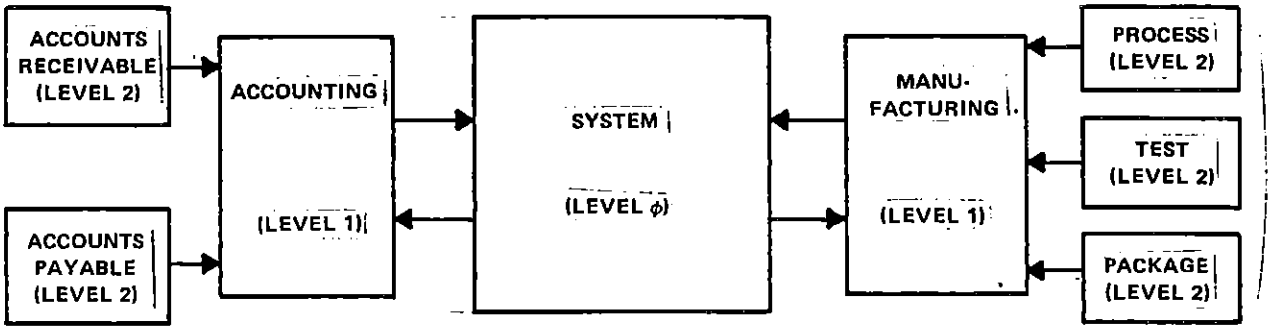



FIGURE 1-1. EXAMPLE OF Rx CONTROL OF A FACTORY SYSTEM.

## 1.5 Rx I/O SUBSYSTEMS

Rx supports Texas Instruments' File I/O Decoder which provides the means for the user to initiate and execute device-independent (file) I/O.

Use of the I/O subsystem enables communication with and manipulation of data to and from various locations regardless of the media on which the data resides.

For more detailed information regarding file I/O, refer to the Device Independent File I/O User's Manual, MP386.

## 1.6 REFERENCE MATERIALS

the following Texas Instruments publications were used in the development of this manual, and provide additional information relative to Rx and to related TI software.

- THE SOFTWARE DEVELOPMENT HANDBOOK, MPA29
- THE FILE MANAGER USER'S MANUAL, MMP355
- MODEL 990 COMPUTER TMS9900 MICROPROCESSOR ASSEMBLY LANGUAGE PROGRAMMER'S GUIDE, 943441-9701
- AMPL MICROPROCESSOR PROTOTYPING LABORATORY OPERATION GUIDE, 946275-9701*A
- 9900 FMILY SYSTEMS DESIGN AND DATA BOOK, LCC4400, 97049-118-NI
- THE MICROPROCESSOR PASCAL SYSTEM USER'S MANUAL, MP351
- DEVICE INDEPENDENT FILE I/O USER'S MANUAL, MP386

# SECTION 2

# Rx CONCEPTS

## 2.1 GENERAL

Understanding the concepts used and supported by Rx is a necessary prerequisite for understanding the product itself. Rx software organization is discussed in subsection 2.2 and descriptions of procedures, functions, processes and systems in the Rx environment are given. Subsection 2.3 will deal with the concept of concurrency, priority scheduling, semaphores and interrupts. Memory organization using stacks and heaps and a discussion of system initialization. Understanding the Rx terms and concepts in this section is necessary in order to understand Rx module construction. Templates are provided in Appendix C.

## 2.2 SOFTWARE ORGANIZATION

In Rx, user applications are built using a collection of processes nested at different levels in a system. These levels are referred to as "lexical levels" and indicate to Rx the level at which a process is embedded within the system. Figure 2-1 depicts this concept:

As the figure indicates, the system is found at lexical level 0, with those processes that the system starts residing at level 1. Any process called in the course of these (Level 1) processes' execution will be assigned level 2 or lower. A process that starts any other process is referred to as that process' lexical "parent", and the called process is referred to as the "spawn". These processes are composed of functional units referred to as procedures and functions.

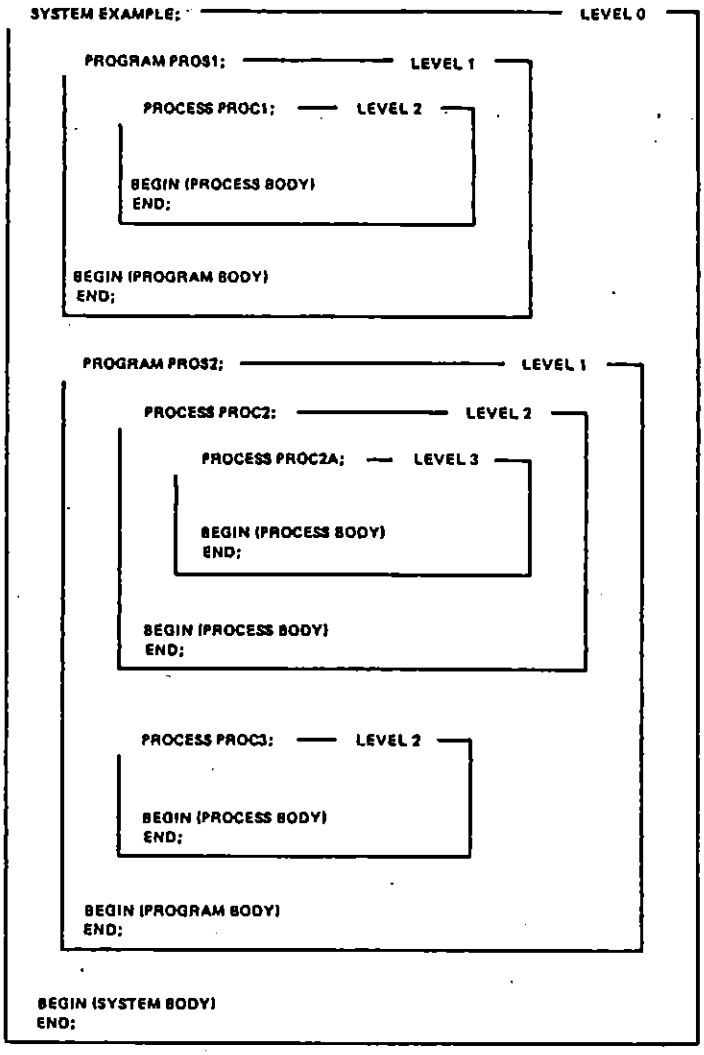



FIGURE 2-1. THE NESTING CONCEPT.

### 2.2.1 System

The System, designated by the SYSTM$ label, is a special case of a process within which all other modules are contained; it is the outermost level of executable statements in the Rx environment. The System usually starts the other processes contained within it. Only one System can exist in the Rx environment at any one time, and all global data is defined within it. Occupying lexical level 0, those processes nested in the System reside at levels 1, 2, 3, and so on. When started, the system is allocated all available memory (or less if it asks for less in the heapsize parameter to S$PRCS) which becomes the system heap (subsection 2.4.2). It is from this parent heap that subsequent allocations of stack and heap, when needed, are made to starting processes. The System, then, may be thought of as the process in which execution begins.

### 2.2.2 Program Process

A special case of process, an Rx program resides only at lexical level 1, and therefore cannot call another program. A program's lexical parent is always the System.

### 2.2.3 Process

Residing at lexical level 1 or lower, a process is a collection of procedures, functions, and data which perform an independent operation concurrently with the scheduling and execution of other processes. Analogous to a "task" in IBM's OS370, it is independently scheduled and may interact with other processes or with the executive as necessary during execution. Note that processes may be created by other processes.

An Rx process is defined in terms of its three component parts: a) the process code, b) the process data, and c) the process record. Processes differ from procedures or functions in that they are independently scheduled by the executive, and are executed in a concurrent fashion. A process will execute until it reaches a point where either required data is needed, a higher priority process becomes ready for execution, or the process simply completes. When control is switched from a process, the current state of the process is "saved" in the associated process record.

The process record itself is an area of memory dynamically allocated from the parent heap (subsection 2.4.1) when the process is started (i.e., performs a call to S$PRCS), and will be located wherever memory is available when it is started. In addition to the status of the current process, the record also contains the process priority, a pointer to the stack and other data required to schedule and execute the process. (Reference Appendix A for the structure of a process record.)

The executive uses the information contained in the process record to schedule. For 'ready' processes (subsection 2.3), a queue of pointers to process records is maintained. The priority of a process determines its place in this queue. (Subsection 2.3.1 discusses priorities and priority scheduling of processes.)

### 2.2.4 Procedure

Functional units of code can be isolated into separate modules called procedures. Similar to a subroutine in FORTRAN, PASCAL or BASIC, procedures are computer software design constructs used to associate sequences of low-level processing steps by the higher-level function they perform. A procedure is included in a process code section by virtue of being invoked (called) by a single instruction from either the process code section, or some other procedure which has been called by the process code section. When execution of the procedure

has completed, control returns to the calling routine at the instruction following the call.

When a procedure is called, an area of memory referred to as a "stack frame" is assigned for that procedure's workspace from the "stack" of the calling process. (See subsection 2.4 for more information on stacks and stack frames.) The size of the assigned memory is a function of the value specified by the stackframe size set in the beginning section of the procedure template (refer to Appendix C, subsection C.3).

### 2.2.5 Function

Functions are composed of a set of instructions that, like procedures, can be called by a single instruction in a process. Procedures and functions are similar with the exception that a function will generally return a result upon its completion. (An Rx function can be likened to the FORTRAN function.)

## 2.3 CONCURRENCY

The Rx environment supports the execution of several processes concurrently. Although it appears to the user that these processes are executing at the same time, execution is actually moving from one to another as CPU time is shared on a process priority basis. The execution of several processes in the same system is termed "multiprogramming". Each process in the Rx environment is in one of three states:

1) Active

2) Ready to execute

3) Suspended (blocked) and waiting for a condition in the system to change (an event to occur) before it can become ready to execute.

The executing process resides in the active queue. Processes that are ready to execute reside in a ready (or scheduling) queue. Processes that are suspended and waiting for an event to occur are placed in another queue associated with the event until notified that an event has taken place. This notification will come via information supplied by a "semaphore" (simply thought of as an event flag). Semaphores are signaled of events by other processes or interrupts. These three features (prioritized scheduling, semaphores, and interrupts) are tools used by Rx to support concurrency. They are described in more detail in the following subsections.

### 2.3.1 Priority Scheduling

The Rx scheduling policy determines the assignment of the processor to one of the ready processes. Ready processes are inserted into the ready queue and scheduled for execution according to priority.

A process' priority is represented by a user-assigned numeric value. The greatest urgency is represented by 0; the least by 32767, which is reserved for the IDLE process. (IDLE is active only when all other processes in the system are blocked.) Priority values 0 to 15 indicate device processes associated with interrupts. Interrupts occur due to a change in some "real world" condition or because they are programmed to occur. Priority values 16 TO 32766 represent non-device processes.

A scheduling decision is made by Rx each time a suspended process becomes ready or the currently executing process terminates or becomes suspended. (An explanation of process readiness follows this discussion of scheduling.) When the active process terminates execution (or becomes suspended) the first process in the ready queue becomes the active process. Because the ready queue is ordered by priority, the most urgent process that is ready is given the processor. When a suspended process becomes ready, it is inserted in the ready queue based on its priority. The newly-ready process preempts the currently active process (i.e., is placed in it's place in the active queue) if it is more urgent. Non-device processes that become ready are placed in the queue behind processes of equal priority. This ensures that when two processes have equal priority, the process that has been ready the longest executes first. Device processes are placed in front of other processes of equal priority including the active process. Figure 2-2 illustrates the working of this scheduling policy.

The first column contains the active process. The ready queue is represented as a horizontal series of boxes behind the active process. Each process (box) is labeled with a letter and a priority number. The first box in the ready queue is the active process. Time moves vertically from top to bottom. Comments to the right of each queue describe the action performed.

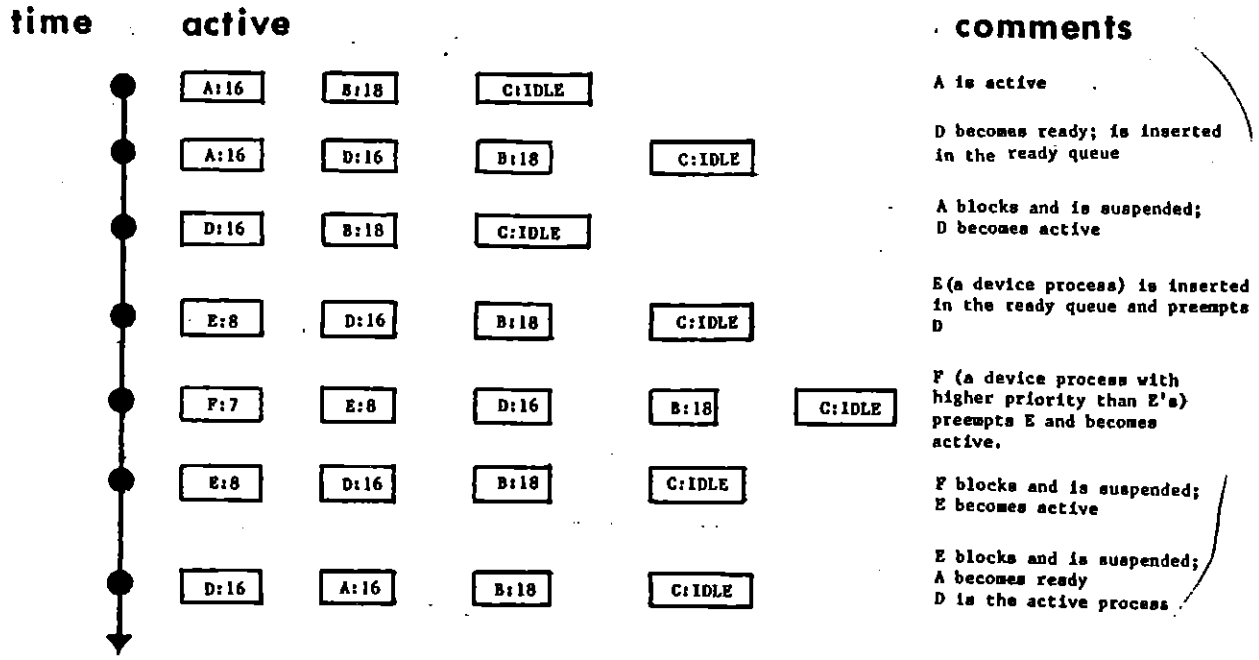



FIGURE 2-2. SCHEDULING POLICY.

The execution of RX scheduling policy displayed in Figure 2-2 results in process "B" never becoming active. In fact, B will never become active unless all other processes in queue with greater urgency become blocked or terminate execution. A process of higher urgency that becomes ready will always interrupt the (currently) active process. Once the more urgent process terminates (or becomes blocked) the previously active process will resume execution (unless another higher priority process become ready). This "preemptive scheduling with resumption" is designed for event-driven systems in which the event is some real-world occurrence that demands the immediate attention of the processor.

Up to this point, the discussion has been concerned with the management of processes that are ready for execution. However, processes may become suspended or blocked because of a condition in the system. When another process signals that the condition has changed, the waiting process can become ready. The mechanics of this process synchronization are described in the next section.

### 2.3.2 SEMAPHORES

The semaphore is the fundamental mechanism for synchronization of processes via Rx, and can be thought of as representing some event on which processes synchronize. A process which is dependent on the occurence of an event can perform a WAIT operation to ensure that the event has occurred before continuing execution. If the event has already occurred, the process executes; if not, the process is suspended in that semaphore's queue until the event does occur. A SIGNAL operation performed on the associated semaphore allows a process to signal the occurence of an event. If some process is waiting for the event, it is made ready for execution by removing it from that semaphore's queue and inserted into the ready (or 'scheduling') queue. If no process is waiting, the occurrence of the event is recorded in the semaphore until a WAIT operation occurs for that event.

The semaphores of Rx can be thought of as "counting" semaphores in that an occurence of an event is never lost, even if no process is waiting when the event occurs; a count is kept in the semaphore of all events that occurred (by SIGNAL) but were not received (by WAIT). (Reference Section 4, 4.2.2 and 4.2.3 for further informatio on SIGNAL and WAIT.)

Rx defines semaphores as structures composed of three elements:

1) A non-negative counter of unserviced events

2) A queue (possibly empty) of suspended processes. In this queue, processes are made ready on a first-in first-out (FIFO) basis

3) A level specifying the interrupt levels that this semaphore may be associated with.

A semaphore is operated on by several procedures, the most important of which are WAIT and SIGNAL. These operations are implemented as routines, but are executed as though they were single machine instructions. Until these operations have completed, nothing can access the semaphore, the queues, or the operations themselves. This is assured when the interrupt mask is set to zero upon entry to the routines, and reset to its previous state upon exit.

WAIT decrements the counter; if the counter is zero, the currently active process is suspended (the process is moved from the active queue to the semaphore queue).

SIGNAL increments the counter if the semaphore queue is not empty, the first process in the queue (which will always be the process that has been in the queue the longest) is activated by

moving the first process from the semaphore queue to the ready queue.

When semaphores are used to ensure exclusive access to two or more resources, extreme caution must be exercised to prevent a condition known as "deadlock". This takes place when a situation is created in which two or more processes are suspended, awaiting a condition that cannot happen because there is no active process to cause the needed event to occur.

For example, if two simultaneously executing processes (A and B) both require exclusive access to resources (X and Y), the following sequence can result:

| Incorrect (deadlock) | Correct |
|---|---|
| A gets X, A requests Y | A requests X, then Y |
| B gets Y, B requests X | B requests X, then Y |

In the above incorrect example, neither A nor B will ever resume execution, as A will be waiting for Y (which B has) and B will be waiting for X (which A has). The safest way to prevent such a situation is for all processes to request resources in the same order. In the above correct example, the X resource is used as a "lock"; if a process can allocate X it is guaranteed to find the rest of the resources available.

### 2.3.3 Interrupts

Interrupts are hardware-signalled events, usually associated with system peripheral or process monitoring devices. With Rx, processes may be specified to service interrupts. These processes must have a priority greater than or equal to the interrupt level which they will service. Level 0 denotes the highest priority: the RESET interrupt; all other interrupt level or device process priorities must be between 1 and 15.

There are several Rx routines that are used to associate semaphores with hardware interrupts. The primary two being Procedure EXTERNalevent and Procedure ALTEXTernalevent. (Section 4, 4.3 lists and describes these procedures.)

The CPU has a priority ranking system to resolve conflicts between simultaneous interrupts, and a level mask to disable lower priority interrupts. A process waits on an interrupt by waiting on a semaphore associated with the interrupt. (The process will have a priority level number less than or equal to the interrupt to be serviced.) The process is suspended until an interrupt at the appropriate level occurs. When the interrupt occurs, the process is scheduled to service the interrupt as the

CPU performs a context switch to the interrupt service routine. The interrupt mask of the process prohibits further interrupts from this or any lower priority device while the interrupt is being serviced.

There are three methods by which the user can write an interrupt handler. The first uses the Rx procedure EXTERNalevent to assign a semaphore to an interrupt. The second uses Procedure ASSEMBlyevent (See Section 4) to assign a dedicated assembly language interrupt handler. The last method entails writing a routine external to the Rx environment.

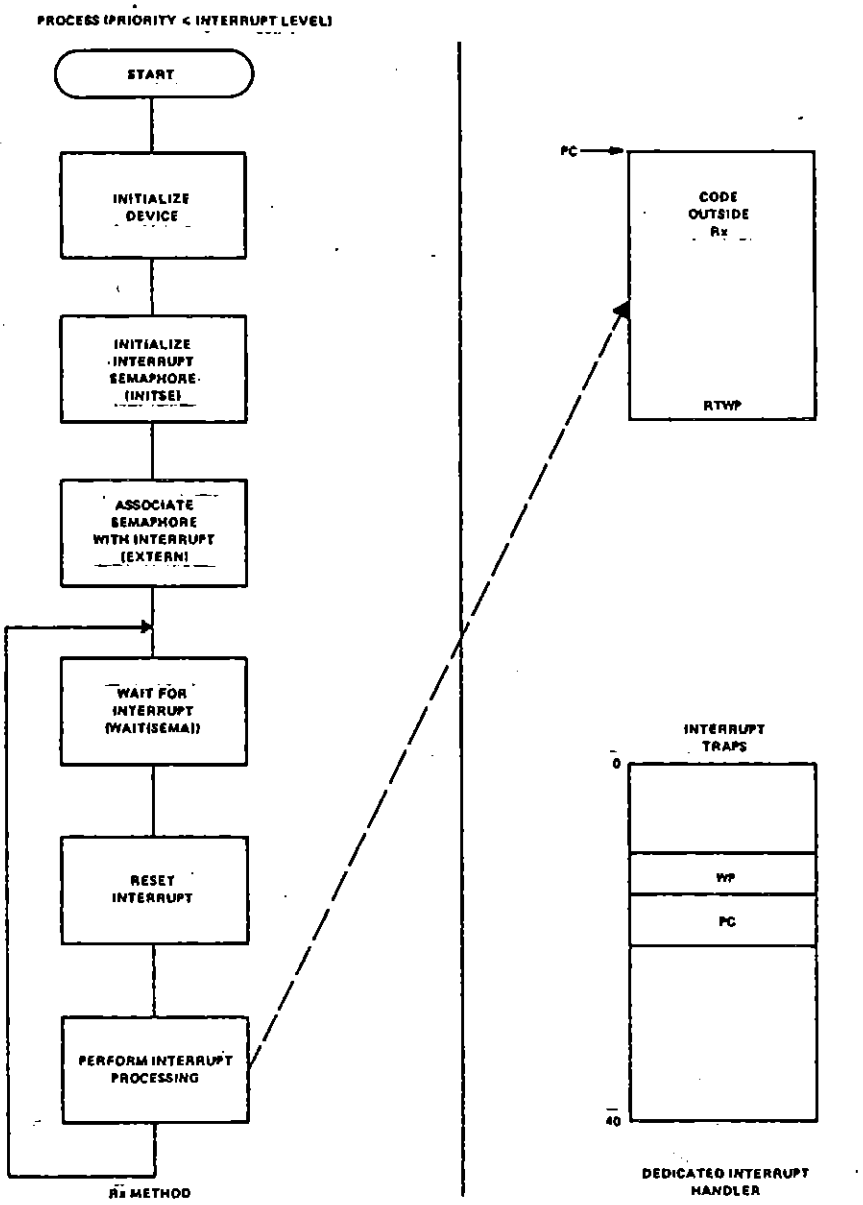



FIGURE 2-3. INTERRUPT HANDLERS

The Rx method shown in Figure 2-3 is useful to the user because it is flexible: the interrupt is reset and can be associated with another semaphore to perform its function over and over again.

In the interest of speed, however, the second method illustrated in 2-3 using a dedicated interrupt handler can be chosen. This involves the user writing a piece of code completely outside the Rx environment that will process the interrupt.

In the 9900 Family CPU, when the interrupt occurs, a context switch is performed. The contents of the current workspace (WP) and program counter (PC) registers are saved and then loaded with the address of the new WP and PC which are fetched from the appropriate interrupt vector, and stored. (This identifies the location of the workspace assigned to the interrupt service routine.) When the context switch is completed, processing resumes with the first instruction of the interrupt service routine (the user's interrupt handler code). Processing will continue in this mode until an RTWP instruction is encountered, and a reverse context switch returns control to the previous program in Rx.

Although the dedicated interrupt method is faster than the Rx method, there is a drawback: the workspace areas created for the context must reside in memory space (RAM) not known to Rx, and therefore cannot be reclaimed for use by Rx. In addition, interrupts must be masked during execution of the interrupt handler. LIMI 0 as the first instruction in the interupt handler results in the required masking of interrupts.

## 2.4 MEMORY ORGANIZATION

The following concepts deal with the utilization of memory in the Rx environment. Memory is allocated and used to hold variables and workspaces for each process, procedure and function through the use of the "stack" and "heap" concepts which are described in the following subsections. It may be said that Rx "owns" the memory allocated to it and "loans" it to processes as they need it, for as long as they need it.

### 2.4.1 Heap

The system heap is all available RAM memory allocated to the system when it is started. It is from this heap that all future memory allocations to starting processes will come. Each process that calls another process, procedure or function will be the "parent" that allocates memory for use by the routines it "spawns" or calls. A process, then, is allocated heap and stack from its lexical parent's heap. (Note that heap should be allocated to a process if that process will be starting any other to ensure there will be enough memory for the called process'

stack.)

A process' heap is an area of memory allocated in packets which may be disposed of and used again. These packets are used as storage for dynamically allocated variables. When required, a call to NEW$ by the executing process requests a packet of memory of a certain size and sets a pointer to this packet. When the process no longer needs this space, a call to FREE$ returns the packet to the "pool" where it can be used again.

### 2.4.2 Stack

This is a region of memory associated with a process. A separate stack region is allocated to each instance of a process when that process is started (via S$PRCS). Although the "top" and "bottom" of a stack are fixed, the use of the area inside is dynamic. Space is allocated to procedures and functions by Rx from both ends, working toward the middle, since both local variables and workspace areas are required.

Every invocation of a procedure or function requires a workspace for the called routine. Space in which to store local variables (i.e., variables that will be used by the called routine), as well as other data is also needed. This space is called the stack frame. Memory is allocated from the top down for workspaces, and from the bottom up for stack frames; a stack overflow would indicate that the two have met in the middle. (Reference Appendix A for further information on stacks and stack frames.)

In Figure 2-4, the concepts of stack and heap are depicted, beginning with the initial allocation of system heap and progressing through the starting of the lexical level 1 process which, in turn, calls a subordinate process. (Note the allocations of stack and heap by the lexical parents.)

Note that processes must have a stack region but are not required to have a heap region unless they start another process or use dynamically allocated heap packets.

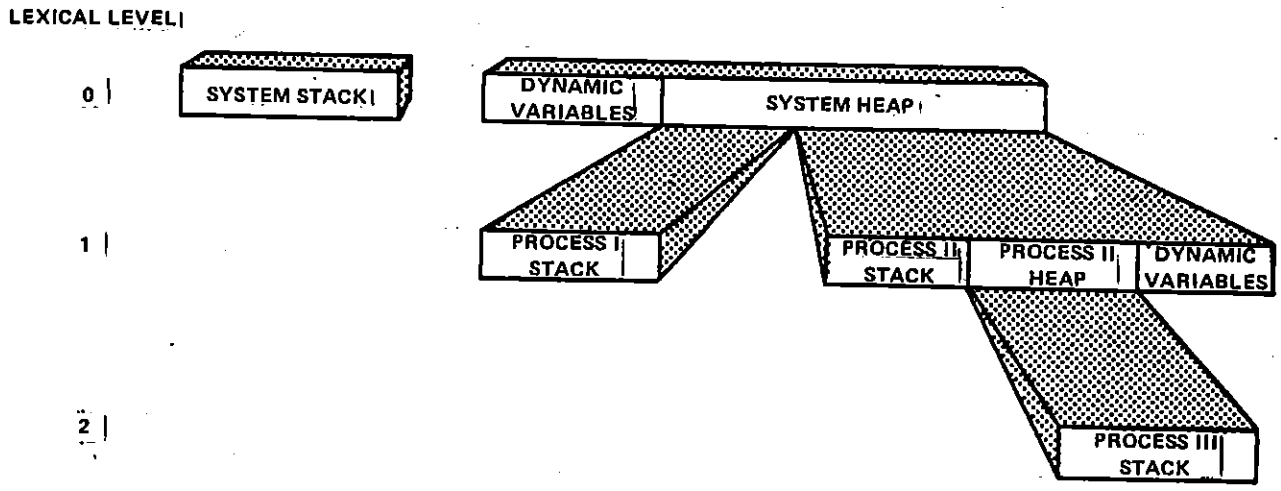



FIGURE 2-4. STACK AND HEAP ALLOCATIONS IN Rx.

## 2.5 SYSTEM INITIALIZATION

System initialization is accomplished by a hardware interrupt which is reserved for the processor hardware RESET. The reset vector contained in low memory points to the RXINIT routine. RXINIT declares the default system crash routine (which consists of an IDLE instruction), and starts the BOOT$ program. BOOT$ initializes system data structures, starts the IDLE program, and then calls the GHOST procedure. GHOST$, in turn, starts the user's system module at the SYSTM$ designation. (Note that GHOST$ may be customized to perform application-independent initialization (refer to 6.3 for further information).

The IDLE process runs when all other processes are "asleep" (i.e., suspended or waiting). With a priority of 32767 (the lowest possible priority), IDLE is actuated when all other processes of priority 32766 or less are suspended or when there are no other processes.

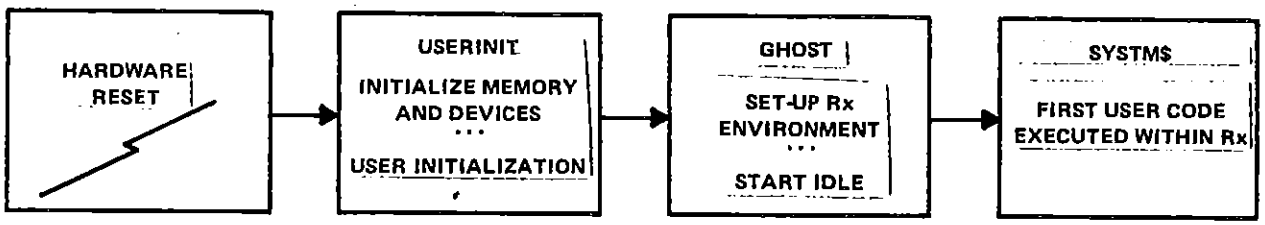



FIGURE 2-5. Rx SYSTEM INITIALIZATION.

## 2.6 SUMMARY

The software tools discussed in this section provide interprocess scheduling and coordination of system resources in Rx. While simple in terms of construct, these tools provide a sophistication in capability that enables a higher programmer productivity using a realtime programming environment. Detailed information on how to write Rx applications using the concepts described follows in Section III.

# SECTION 3

# Rx CONVENTIONS

## 3.1 GENERAL

When writing software to be used with Rx, certain conventions must be followed when interaction is required between the application software and the executive environment. These conventions apply to the way in which the application code is structured, how routines are called, and which registers may be used. The following sections detail these conventions as they apply to Rx procedures, functions, processes, and systems.

When using the Rx routine linkage mechanisms described in 3.2, the routines must be structured according to the proper module format (i.e., procedure, function, process). These linkage and module format conventions give the code certain properties which increase the reliability and flexibility of the software. Linkage conventions produce code which is reentrant. Reentrant procedures may be executing within more than one process at a time without erroneous results. By using the same portion of code to do two or more concurrent tasks, memory space is conserved. The standard linkage conventions also produce code which is recursive, allowing the procedure to call itself. This property can be very useful when solving certain types of complex problems.

## 3.2 LINKAGE CONVENTIONS

There are two types of linkage supported within Rx: standard and optimized procedure/function linkage. The standard linkage provides a modular approach to writing these routines. It allows the calling procedure to know nothing about the called procedure except the arguments passed between them (no registers must be saved, etc.). The optimized linkage provides a faster linkage mechanism for routines which will not call any other routines or need any local storage.

### 3.2.1 Standard Procedure/Function Linkage

The standard procedure/function linkage supports parameter passing, local storage, reentrancy, and recursion. It achieves these by using the stack data structure illustrated in Figure 3-1. In this stack, stack frames grow from the bottom toward high memory while workspaces grow from the top toward low memory. The stack region is allocated when the process is created. A stack overflow error occurs when there is not enough stack for another procedure call (the stack frames and

workspaces overlap). The first workspace is used by the process. Its stack pointer (R10)

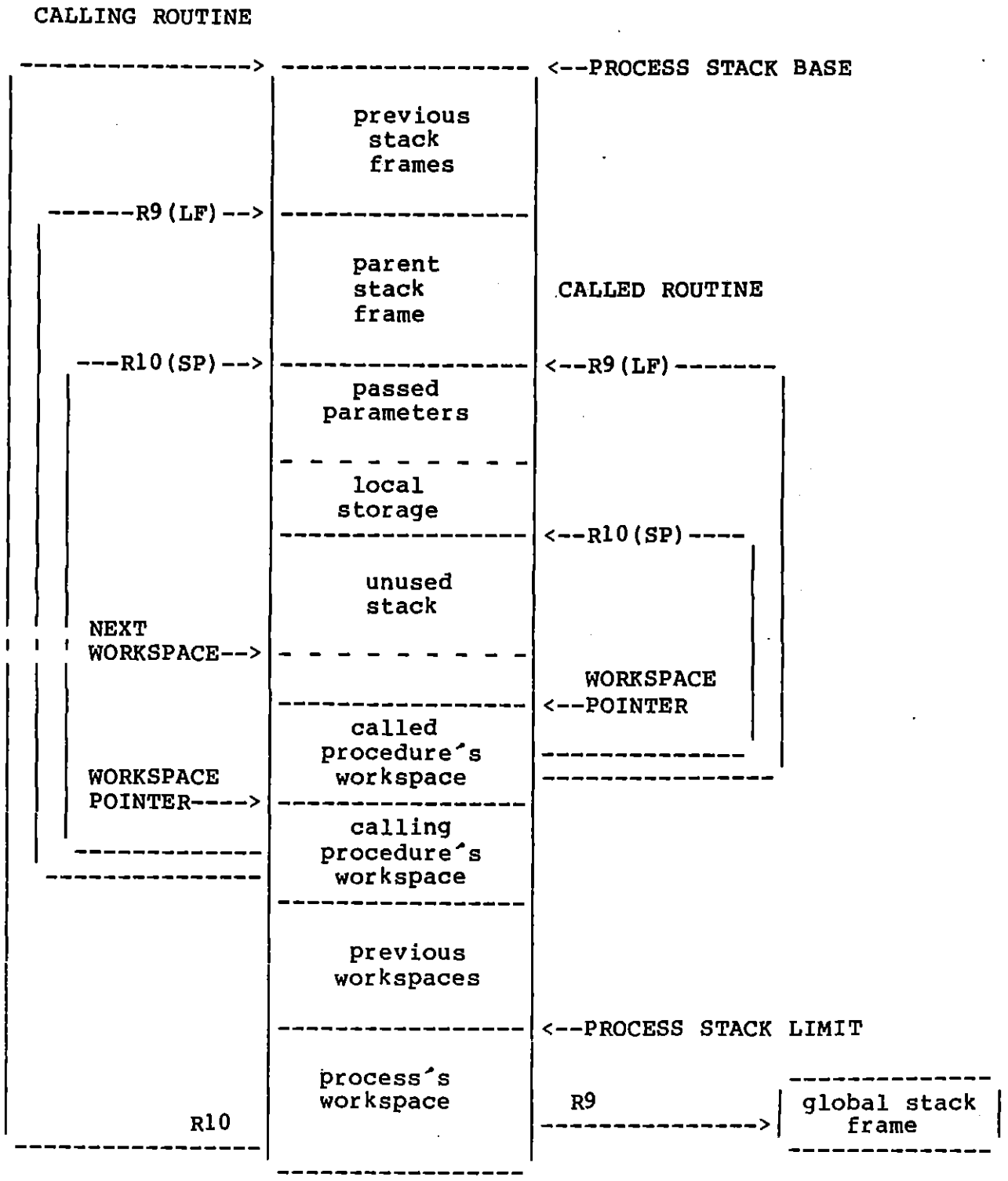



FIGURE 3-1. STANDARD STACK NESTING.

points to the base of the process stack, while the local frame pointer (R9) points to the global stack frame which is a separate memory packet. In the example, a routine has been called from another routine nested within a process. The calling routines stack pointer (R10) and local frame pointer (R9) are shown on the left. These pointers, other system pointers, and the routine's general registers are contained in the calling routine's workspace. The called routine has a new workspace allocated to it which is pointed to by the workpsace pointer shown on the right. Registers R9 and R10 in this workspace point to its local frame and current top-of-stack. Both the parameters and local storage are referenced using R9 (LF) as the base register. This type of routine nesting repeats for as many routines as are called. As routines return to their caller, their stack frame and workspace are returned to the unused portion of the stack.

The stack frame for a standard procedure/function is determined by the routine prologue described in Sections 3.3.1 and 3.3.2. Basically, the routine prologue specifies how many parameters the called routine expects the calling routine to have pushed and how much local storage the called routine needs. Both the parameters and local storage are referenced using R9(LF) as the base register.

A routine with standard linkage has two data areas that it may access during execution. It may use any of the general registers (described in Section 3.4) and the local storage space of its stack frame. The general registers should be used for frequently accessed data or if only a few words of storage are needed. If the general registers do not provide enough data space, then either local storage or the stack must be used. Local storage is an area reserved in the stack immediately above the passed parameters that remains allocated until the routine returns. Therefore, this space should be used for data which will be accessed during nested routine calls. When a routine is called, there must be enough unused stack to allow for the standard linkage memory requirements. These requirements include the new workspace, the passed parameters, and any local storage. When enough stack does not exist, a stack overflow error occurs. The process stack size is contained in the literals field of the process' code.

When a process is invoked, its stack is allocated with the desired size, and its workspace is defined and initialized. Its process record is also created, the first field of which is contains a pointer to the first 32 bytes of unused workspace stack which resides just before this new process's initialized workspace. This pointer is called the "next workspace" pointer and will be used as the workspace pointer of any routine called from the process module. Whenever a routine is called, the linkage handler decrements "next workspace" by 32 bytes to be prepared for the next call. As a result of this algorithm, the unused stack region will always be at least 32 bytes long, and it will always be possible to use this amount of stack without overflow occurring. For example, 16 ne-word parameters could be pushed onto the stack without danger of over-writing the current workspace. If the parameters intrude into what will become the called routine's

workspace, they may be modified as linkage handler executes out of the new workspace; however stack overflow will be detected before the called routine can make use of the erroneous parameters. CAUTION: if more parameters are pushed and there is no additional unused stack other than the next workspace, the contents of the allocated workspaces of previously called routines may be destroyed, and catastrophic errors may occur which are impossible to recover from and difficult to debug. If a routine will pass more than 32 bytes of parameters, the following code should inserted into the prologue of that routine to make a pre-emptive stack overflow check:

```
BL   @STK$CK
DATA "maximum number of bytes to be pushed"
```

The "next workspace" field of the process record in incremented by 32 at exit from a routine to reclaim the space used for its workspace.

To make use of the standard linkage, a routine must be called in the proper manner. Parameters can be passed in two different ways: by value and by reference. A value parameter contains the actual value being passed, while a reference parameter is the address of the variable. The user should be sure that a parameter is referenced in a consistent manner. An example of a call using the standard routine linkage is as follows:

Code in the calling routine:

```
  ...
MOV @PARM1,*R10+        PUSH FIRST ARGUMENT
MOV @PARM2,*R10+        PUSH SECOND ARGUMENT
  ...
MOV @PARMn,*R10+        PUSH (n)th ARGUMENT
DATA CALL$,ROUTIN       CALL ROUTINE 'ROUTIN'
  ...
```

Code in the called routine:

```
  ...
MOV *R9,@ARG1           SAVE ARGUMENT ONE
MOV @2(R9),@ARG2        SAVE ARGUMENT TWO
  ...
MOV @2*n-2(R9),@PARMn   SAVE ARGUMENT n
  ...
```

If any local storage is specified, this storage begins at an address pointed to by the displacement (2*n) off of R9 and extends for as many bytes as specified.

Function linkages are similar, and arguments are passed in exactly the same way. The function result is returned at the end of the stack; i.e. upon return from the function, the stack pointer R10 points to the first word of the result, which may be 1 byte, 2 bytes, or 4 bytes long.

For detailed examples of subroutine linkage see the routine templates in Appendix C, and the demonstration program in Appendix E.

The actual linkage functions are performed by the standard RX routine CALL$$. This external symbol CALL$ is resolved to be the instruction BLWP *PR which performs a "branch and link workspaces" using the transfer vector contained in the first two words of the process record of the active process. As mentioned previously, the first word in the process record is a pointer to the next workspace. The second word contains the address of the entry handler, CALL$$. The resulting action is a branch to CALL$$ and begin executing in the "next workspace." CALL$$ initializes registers in the new workspace, resets the next workspace pointer in the process record, and branches to the code of the called routine.

When the calling routine has completed, it returns via a branch or branch and link to the exit routine. This routine deallocates the stack frame and workspace allocated to the execution of the calling routine.

### 3.2.2 Optimized Linkage

An alternative to the standard routine linkage is the optimized linkage mechanism. This linkage executes faster than the standard linkage since it does not perform as many functions: a new workspace is allocated and initialized for the called routine; local storage, other than the space needed for passed parameters, is not allocated. Any routine which was called with the optimized linkage cannot call other routines.

Figure 3-2 illustrates a process stack after an optimized linkage. The calling routine's workspace pointer and registers are shown on the left, while those of the called routine are shown on the right.

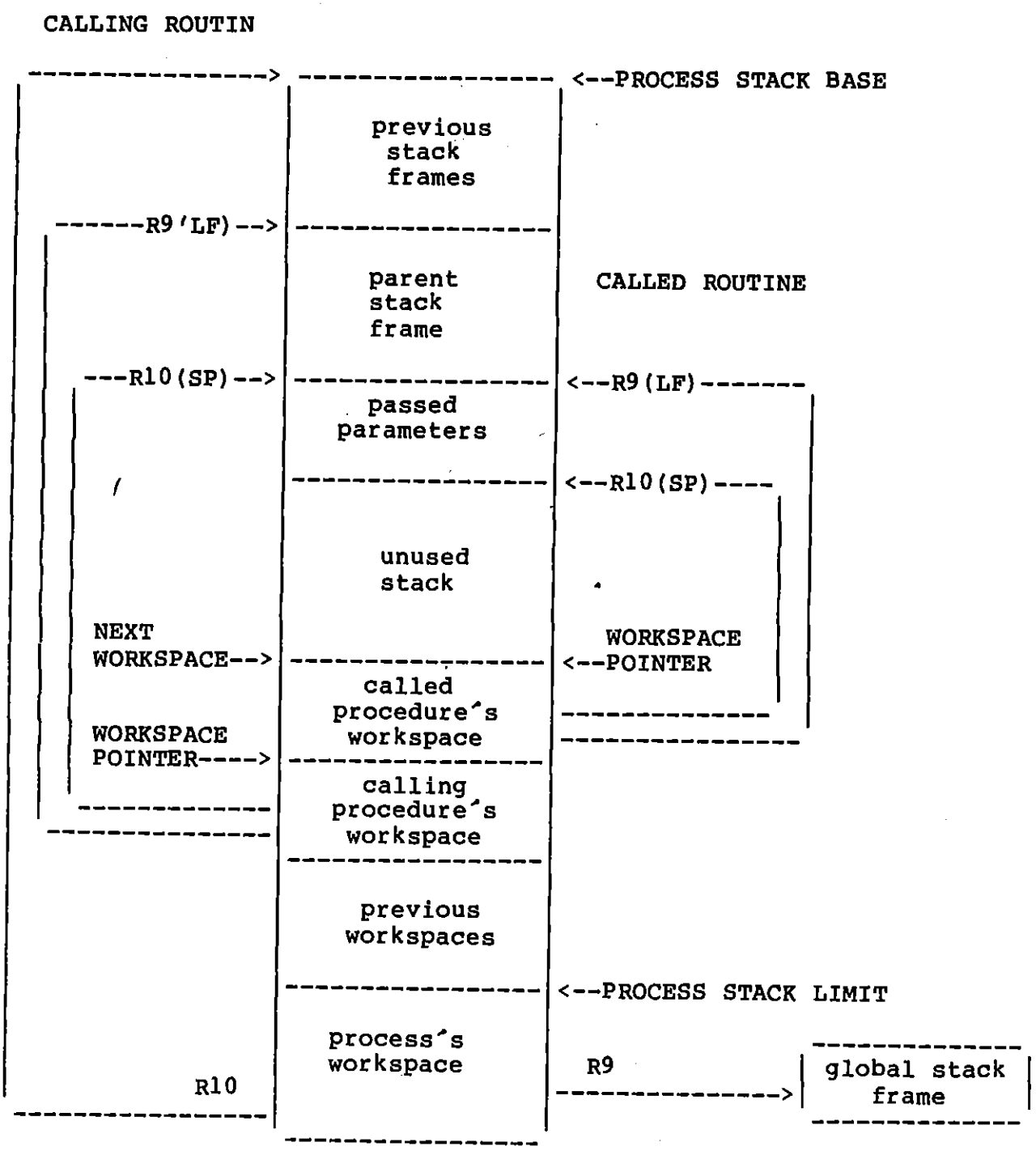



FIGURE 3-2. OPTIMIZED STACK NESTING.

The stack frame, eneral registers, and unused stack are available to an optimized routine. Since the optimized linkage produces a stack frame which is only used for passed parameters, local storage is not reserved.

The calling sequence to an optimized routine is identical to the calling sequence for a standard linkage routine. This similarity allows the calling routine to call other routines without knowing whether they are coded for either standard or optimized linkage. When the MPX linkage routine determines that the called routine is using the optimized linkage, it initializes workspace registers and branches directly to the new routine's code. Since a routine with optimized linkage is not permitted to call other routines, there is no need to update the "next workspace" pointer in the process record. The called routine with optimized linkage references the passed parameters in the same way as the standard linkage routine.

When the optimized routine has completed, it returns to the calling routine via a return with workspace pointer (RTWP) instruction. This takes the saved workspace pointer, program counter, and status from registers R13 through R15 and restores them.

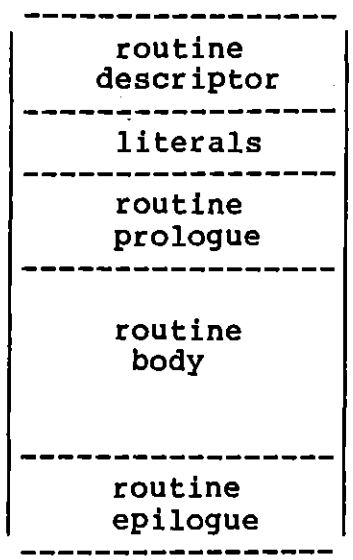
### 3.2.3 Process Linkage

The process linkage mechanism is very similar in use to the standard procedure/function linkage. The process is called in an identical manner; however parameters can only be passed by value. The called process gets the parameters in an identical manner. However, the effect on the stack is quite different. The initial code of a process contains calls to the executive which create a new stack region (along with a process record and other process data structures) for the new process to execute from. Once this initial code has executed the calling procedures stack returns to the state it was in prior to pushing parameters and calling the process. The new processes' stack and process record are allocated from the heap region of its calling process. If the calling process does not have a heap, the stack and heap are allocated from the system heap.

NOTE: for one process to start another, about 144 bytes of stack space are required.

## 3.3 SOURCE MODULE FORMAT

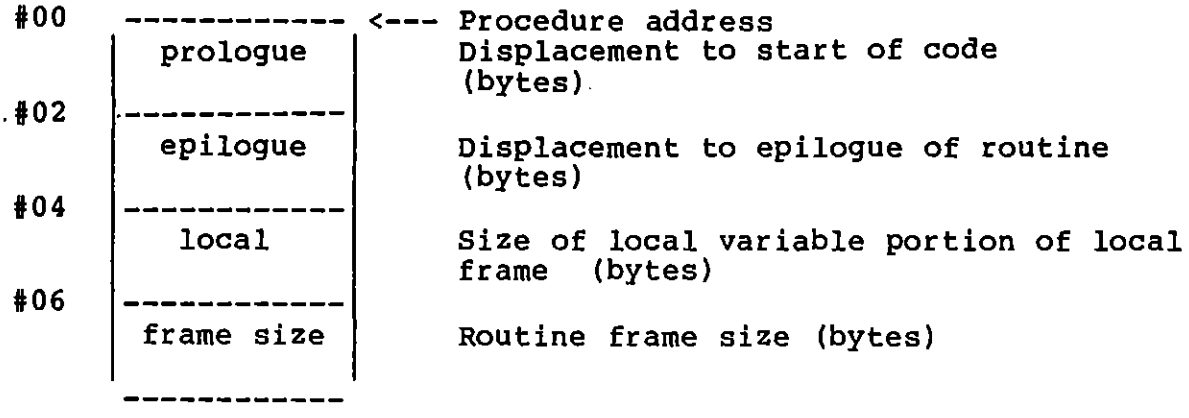
To make use of the Rx linkage mechanisms, routines must be formatted in a certain structure. It is this structure which allows the linkage mechanism to operate. The basic structure consists of the following segments within the routine:




The routine descriptor contains constants needed by the linkage routine upon routine entry. The routine prologue contains any code necessary to start the routine. The routine body is the code which actually performs the purpose of the routine. The routine epilogue is the code required to exit the routine. Some routine types do not require all of these code segments. The different routine formats are summarized in Appendix C.

### 3.3.1 Standard Procedure

A standard procedure requires a desriptor with the following information:




The start offset defines the offset to be added to the procedure address for the initial procedure program counter. The end offset defines the offset to be added to the procedure address in case the procedure is aborted. The local size specifies how many bytes should be allocated from the stack when the procedure is called for use as local storage. The frame size specifies the total stack frame size including passed parameters and local storage. Both the local size and

frame size should be even values.

The prologue of a procedure is usually just a label which is immediately followed by the procedure's main body.

The procedure body consists of the assembly language statements required to achieve the procedure's desired effect. (This will vary from procedure to procedure).

The procedure epilogue contains a branch to the Rx procedure exit routine EXIT$P. This routine returns execution to the calling routine.

### 3.3.2 Standard Function

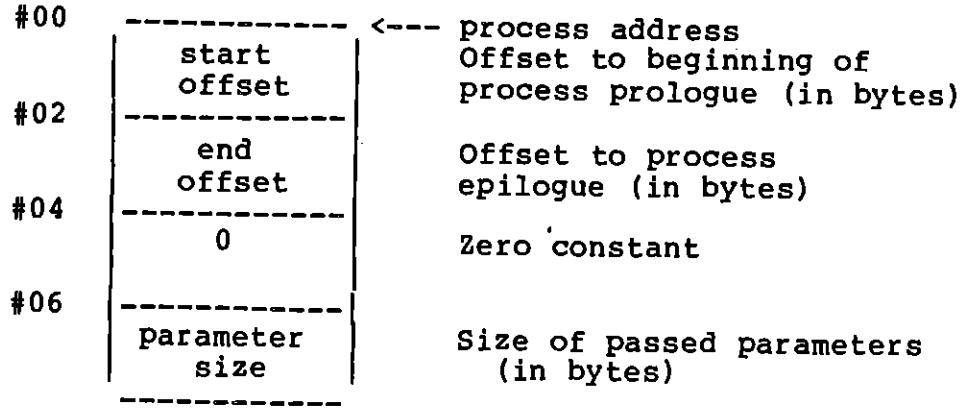
The standard function format is very similar to the standard procedure format, the only difference being that the epilogue section of the function must return the function result. A standard function epilogue consists of the following:

```
BL   @EXIT$n
DATA mmmm
```

In this example, "n" is the length of the result in words and "mmmm" is the displacement into the stack frame in bytes of the result. The EXIT$n routine returns the function result at the stack pointer of the calling routine and returns execution to the calling routine.

### 3.3.3 Process

The standard process format contains a descriptor with the following information:




The start offset and end offset have the same meaning as for a procedure or function. The parameter size specifies how many bytes of parameters that the starting routine has pushed onto the stack. The zero constant specifies how much local storage should be allocated. This is always zero because a process is invoked in two steps. The first phase invokes the process routine as a standard procedure with a

stack frame just large enough to contain to process's parameters; the "0" in the process descriptor suppresses allocation of local variables. When the process routine is entered, it calls procedure S$PRCS to perform the second phase of process invocation by using constants in the literals section to create the data structures that permit the new process to become a separate site of execution. In particular, one of these literals is the total frame size of the process module. (With this implementation the invoking process need not have enough stack space to hold the global frame of the new process.)

The literals segment contains the following:

```
(1)  routin  PSEG
(2)          EQU  $            Origin of process
                .
                .
(3)  frmsiz  DATA >nnnn        Total size of stack frame needed for
                                 process body (in bytes)
(4)  lexlvl  DATA >nnnn        Lexical nesting level.
(5)  priori                    Process Priority.
(6)  stksiz                    Size of stack region to be allocated for
                                 process (in words).
(7)  hpsize                    Size of heap required for process (in
                                 words).
```

The frame size specifies the process's global stack frame size. The lexical level specifies the number of levels that this process is nested within other processes. The lexical level of a system process is 0, a program process started from the system has a lexical level of 1, a process started from a program has a lexical level of 2, etc. These lexical levels are not enforced by the system, but are conventions which the user must follow to allow variable scoping and communication with other software using the RX Executive.

The process priority specifies the relative urgency of this process compared to other processes. The lower the numerical priority, the greater the urgency. The process stack size specifies how many words of stack will be required for the routines within the process. The process heap size specifies how many words of heap memory the process will need. Any user defined constants are stored after the heapsize, and before the first executable statement.

The prologue of a process body is required to initialize the process data structures and schedule the process according to its priority. The prologue contains the following start up code:

```
prolog  EQU
        MOV   @frmsiz-routin(R8),*R10+   Push frame size (in bytes)
        MOV   @LEXLVL-routin(R8),*R10+   Push lexical level
        MOV   @PRIORI-routin(R8),*R10+   Push process priority
        MOV   @STKSIZ-routin(R8),*R10+   Push stack size (in words)
        MOV   @HPSIZE-routin(R8),*R10+   Push heap size (in words)
        DATA  CALL$,S$PRCS               Call start process
```

This code passes the necessary parameters from the literals segment to the process start procedure S$PRCS. All of the parameters do not necessarily have to be distinct entries in the literals section. The code can be optimized so that the parameters are shared or omitted altogether. For example, if a system process with a zero frame size, priority, stack, and heap size were started, the literals section could be empty and the prologue optimized.

```
CLR   *R10+          PASS FRAME SIZE
CLR   *R10+          PASS LEX LEVEL
CLR   *R10+          PASS PRIORITY
CLR   *R10+          PASS STACK SIZE
CLR   *R10+          PASS HEAP SIZE
DATA  CALL$          CALL START PROCESS
DATA  S$PRCS
```

The S$PRCS routine concludes the second step of process initialization. Not only does it allocate and initialize the new process's data structures, it also copies the parameters contained in the temporary stack frame into the new global stack frame and sets the contents of the new process workspace to begin execution at the first instruction following the call to S$PRCS. The temporary workspace and stack frame are then returned to the unused stack of the starting process.

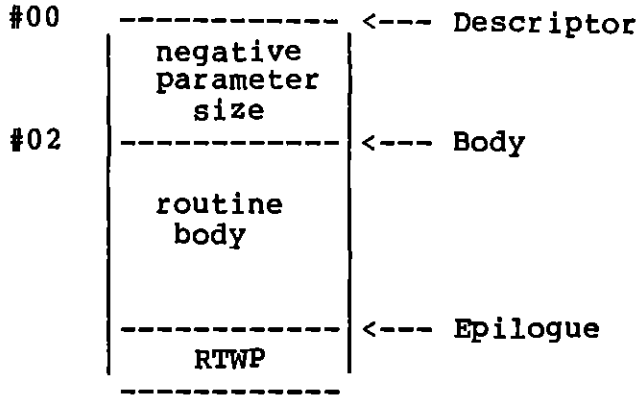
The epilogue of a process terminates execution of the process by calling the MPX routine E$PRCS to deallocate its resources. Its process record and stack region are deallocated immediately; the global stack frame is deallocated after all offspring processes have terminated. The epilogue contains the following termination code:

```
MOV   @LEXLVL-routin(R8),*R10+   PUSH LEX LEVEL
DATA  CALL$,E$PRCS               CALL PROCESS TERMINATION ROUTINE
B     @EXIT$P                    COMPLETE TERMINATION PROCESSING
```

This code passes the lexical level to the process termination routine and then branches to the procedure exit routine.

### 3.3.4 Optimized Procedure

The optimized routine format is fairly simple. The format of an optimized routine is as follows:




The descriptor contains only the negative value of the parameter size (or zero if the routine has no parameters). This non-positive value indicates to the linkage routine CALL$$ that this routine is in an optimized format. The actual parameter size can then be easily computed by negating the value. The routine body contains no prologue. The epilogue is only a return with workspace pointer (RTWP) instruction which causes execution to return to the calling routine when this routine is finished.

### 3.3.5 Optimized Function

An optimized function format is very similar to an optimized procedure format. The difference is the epilogue which must contain the following:

```
REF   EXIT$O          OH, NOT ZERO
...
MOV   @RESULT,*LF     SAVE RESULT ON THE STACK
BL    @EXIT$O         CALL EXIT ROUTINE
```

The function result is stored in the location pointed to by R9 (LF), and the EXIT$O optimized exit routine is called. This routine saves the status so that it is available to the caller. The calling routine retrieves the result from the location pointed to by its own stack pointer (R10). Note that EXIT$O only allows one word function results.

## 3.4 REGISTER USAGE

Rx uses certain registers within procedure, function, and process workspaces to maintain system level pointers. These registers must not be changed by the application software or erroneous results may occur. The following registers may NOT be changed when using optimized linkages:

R13 - Old Workspace Pointer. This register maintains a link to the previous routine's workspace.

R14 - Old Program Counter. This register maintains a link to the previous routine's program counter.

When using the standard linkage mechanism, the following registers are assigned special purposes and may not be altered by the application software:

R7 - Process Record Pointer. This register contains the address of the process record for this process.

R8 - Code Base. This register contains the address of this routine and may be used as a base register.

R9 - Local Frame. This register contains the address of this routine's stack frame which contains passed parameters and local storage.

R10 - Stack Pointer. This register contains this routine's stack pointer.

R15 - Status Register. This register stores the summary of the results of the processor operation.

The user may, for convenience, include the following equates in his programs:

```
PR    EQU  7
CODE  EQU  8
LF    EQU  9
SP    EQU  10
```

The remaining registers R0, R1, R2, R3, R4, R5, R6, R11, and R12 may be used by the application software. R11 is used by the BL instruction to hold the return address, and may be used as such for subroutine linkage outside of the Rx environment. R12 is reserved as the CRU base if any CRU operations are to be performed.

## 3.5 EXAMPLE PROGRAM

A portion of the Rx demonstration program is included at the end of this section as an example of an Rx program. A complete system is included in Appendix E.

This routine takes three arguments: (1) the CRU address of a terminal, (2) a baud rate flag (which is returned by a routine which sets the baud rate of the terminal), and (3) a pointer to a message which is terminated by a null character (>00).

This program loops through the string and calls another routine TI$COT to print the character on the terminal. TI$COT also takes three arguments: (1) the CRU address of a terminal, (2) a baud rate flag (which is returned by a routine which sets the baud rate of the terminal), and (3) the character to be printed.

```
0380            ****************************************************************
0381            *                                                              *
0382            *      TI$MSG:  OUTPUT A STRING, DELIMITED BY A NULL           *
0383            *                                                              *
0384            *                                                              *
0385            *   PURPOSE:  OUTPUT A STRING TO A 9902 PORT. THE STRING       *
0386            *             IS COMPOSED OF CONSECUTIVE BYTES POINTED         *
0387            *             TO BY THE THIRD ARGUMENT, AND DELIMITED BY       *
0388            *             A ZERO BYTE.                                     *
0389            *                                                              *
0390            *   CALLING_SEQUENCE:                                          *
0391            *             PUSH CRU ADDRESS OF PORT                         *
0392            *             PUSH BAUD RATE FLAG FROM TI$SET                  *
0393            *             PUSH POINTER TO MESSAGE STRING                   *
0394            *                                                              *
0395            *             MOV @<CRU ADDRESS>,*SP+                          *
0396            *             MOV @<BAUD RATE FLAG>,*SP+                       *
0397            *             MOV @<PTR TO MSG>,*SP+                           *
0398            *             DATA CALL$,TI$MSG                                *
0399            *                                                              *
0400            *   INPUTS:   PORT: CRU BASE OF OUTPUT PORT                    *
0401            *             BAUD: BAUD RATE FLAG FROM TI$SET                 *
0402            *             MSGP: POINTER TO MESSAGE STRING                  *
0403            *                                                              *
0404            *   OUTPUTS:  OUTPUT IS SENT TO PORT.                          *
0405            *                                                              *
0406            *   EXCEPTIONS: NONE.                                          *
0407            *                                                              *
0408            *   CALLS:    TI$COT                                           *
0409            *                                                              *
0410            ****************************************************************
0411            *
0412            * REFERENCES
0413            *
0414                  REF    CALL$,EXIT$P
0415                  REF    TI$COT
0416            *
0417            * EQUATES
0418            *
0419     0000   PORTOF EQU    >0000          ADDRESS 2
0420     0002   BAUDOF EQU    >0002          INTEGER 2
0421     0004   MSGP   EQU    >0004          POINTER 2
0422            *
0423     0001   MSGPTR EQU    1
0424     0002   WORD   EQU    2
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                    PAGE 0013
 0426 0126                PSEG
 0427       0126' TI$MSG  EQU $
 0428 0126 0008           DATA   MSGENT-TI$MSG        OFFSET TO EXECUTABLE CODE
 0429 0128 0024           DATA   MSGEXI-TI$MSG        OFFSET TO TERMINATION CODE
 0430 012A 0000           DATA   0                    LOCAL VARIABLE SIZE
 0431 012C 0006           DATA   6+0                  LOCAL FRAME SIZE
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                    PAGE 0014
 0433       012E' MSGENT EQU    $                    MSGENT POINT OF PROCESS
 0434             ************************************************************
 0435             *                  ---- MAIN BODY OF CODE ----              *
 0436             ************************************************************
 0437 012E C069           MOV    @MSGP(LF),MSGPTR     GET POINTER TO MESSAGE
      0130 0004
 0438 0132 04C2 CHARLP    CLR    WORD                 ZERO BOTH BYTES
 0439 0134 D0B1           MOVB   *MSGPTR+,WORD        CHR IN HIGH BYTE OF WORD
 0440 0136 1309           JEQ    MSGEXI               YES: QUIT
 0441 0138 CEA9           MOV    @PORTOF(LF),*SP+     PUSH PORTOF
      013A 0000
 0442 013C CEA9           MOV    @BAUDOF(LF),*SP+     PUSH BAUDOF RATE FLAG
      013E 0002
 0443 0140 06C2           SWPB   WORD                 CHR IN LOW BYTE OF WORD
 0444 0142 CE82           MOV    WORD,*SP+            PUSH WORD ON STACK
 0445 0144 0000           DATA   CALL$,TI$COT         SEND CHR
      0146 00A0'
 0446 0148 10F4           JMP    CHARLP
 0447             ************************************************************
 0448             *                      ---- END OF CODE ----                *
 0449             ************************************************************
 0450       014A' MSGEXI EQU    $                    EXIT CODE
 0451 014A 0460           B      @EXIT$P
      014C 0000
 0452                      END
NO ERRORS,     NO WARNINGS
```

# SECTION 4

# Rx ROUTINES

## 4.1 GENERAL

This section describes user-callable routines within the Rx executive package. These routines are intended to perform commonly needed functions within a software system. Use of these routines will considerably reduce application software complexity while increasing its reliability and understandability.

The routines are grouped according to function, and the purpose of each routine described in detail. The required parameters and calling sequence are listed along with any possible side effects or errors.

## 4.2 LINKAGE ROUTINES

The linkage routines are used to call and return from procedures, functions, and processes.

### 4.2.1 Procedure CALL$$

This nonstandard procedure performs the necessary linkage for standard procedures, standard functions, standard processes, optimized procedures, and optimized function calls. A BLWP vector contained at the beginning of each process record points to this routine and the next available workspace. The routine is entered by performing a BLWP *R7 (R7 points to the current process record). This BLWP instruction has been equated to the symbol CALL$.

The first word of the routine descriptor is compared to zero. If it is zero, an optimized linkage is assumed and the routine is immediately entered. If the first word of the desriptor is nonzero, a standard linkage is performed. The standard linkage includes allocating a workspace for nested routines, initializing the stack pointer (R10), allocating local storage, and initializing the local frame pointer (R9).

EXAMPLE:

```
        REF routine
              .
              .
        DATA CALL$
        DATA routine
```

EXCEPTIONS AND CONDITIONS: A stack overflow error occurs during a

standard linkage if there is not enough stack remaining to allocate the new workspace.

### 4.2.2 Procedure EXIT$P

This nonstandard procedure performs the return from a standard procedure. The next workspace (for nested routines) is deallocated and a RTWP is performed to return to the caller's context.

EXAMPLE:

```
        REF  EXIT$P
               .
               .
        B    @EXIT$P
```

EXCEPTIONS AND CONDITIONS: None.

### 4.3.3 Procedure EXIT$n

This nonstandard procedure returns 'n' words from a standard function. The next workspace (for nested routines) is deallocated. Then 'n' words at 'OFFSET' bytes into the function local frame (R9) are returned at the caller's stack pointer (R10), 'n' must be either 1, 2, or 4. Upon return from this routine, the caller's stack pointer (R10) points to the first word of the function result.

EXAMPLE:

```
        REF  EXIT$n
               .
               .
        BL   EXIT$n
        DATA nnnn          'OFFSET' INTO LOCAL FRAME
```

EXCEPTIONS AND CONDITIONS: The caller's condition code is set based on the returned result.

### 4.2.4 Procedure EXIT$O

This nonstandard procedure returns 1 word from an optimized function. The next workspace (for nested routines) is deallocated. Then 1 words at 'OFFSET' bytes into the function local frame (R9) are returned at the caller's stack pointer (R10); upon return from this routine, the caller's stack pointer (R10) points to the function result.

EXAMPLE:

```
        REF   EXIT$O
                .
                .
        BL    EXIT$O
        MOV   result,*LF
```

EXCEPTIONS AND CONDITIONS: The caller's condition code is set based on the returned result.

### 4.2.5 Procedure S$PRCS

This procedure is called by the prologue of a process to initialize its data structures and start itself. It allocates a process record, process stack, and routine stack from its parent's heap and then initializes all the necessary fields in these structures. The new process is then inserted into the ready queue according to its priority.

EXAMPLE:

```
        REF   S$PRCS
                .
                .
        MOV   @<ga>,*R10+          PUSH FRAME SIZE IN BYTES
        MOV   @<ga>,*R10+          PUSH LEXICAL LEVEL
        MOV   @<ga>,*R10+          PUSH PROCESS PRIORITY
        MOV   @<ga>,*R10+          PUSH STACK SIZE IN WORDS
        MOV   @<ga>,*R10+          PUSH HEAP SIZE IN WORDS
        DATA  CALL$
        DATA  S$PRCS
```

EXCEPTIONS AND CONDITIONS: Errors will occur if there is not enough parent heap to allocate data structures, if the called process does not have enough stack, or if the lexical level is invalid.

### 4.2.6 Procedure E$PRCS

This procedure is called by the epilogue of a process to terminate it.

EXAMPLE:

```
        REF   E$PRCS
                .
                .
        MOV   @<ga>,*R10+          PUSH LEXICAL LEVEL
        DATA  CALL$
        DATA  E$PRCS
```

EXCEPTIONS AND CONDITIONS: None.

## 4.3 SEMAPHORE ROUTINES

These procedures synchronize processes on the basis of events.

### 4.3.1 Procedure INITSEmaphore

This procedure initializes the semaphore SEMA. It allocates three words from the system heap for the semaphore record and puts the address of this record at the specified location. Recall that a semaphore is defined as the address of the semaphore record. It uses the parameter COUNT as the initial value of the number of unprocessed (unreceived) SIGNALs to the semaphore. It initializes the semaphore queue to be empty by setting the queue pointer to zero. It initializes the semaphore level field to be 32767, the lowest urgency level, so that while at this level, any process may wait on this semaphore.

EXAMPLE:

```
        REF   INITSE
              .
              .
        MOV   @<ga>,*R10+       PUSH ADDRESS WHERE SEMAPHORE 'SEMA'
                                WILL BE PLACED
        MOV   @<ga>,*R10+       PUSH INITIALIZATION 'COUNT'
        DATA  CALL$
        DATA  INITSE
```

EXCEPTIONS AND CONDITIONS: This procedure may fail if there is not enough system heap available. It is illegal to initialize the semaphore with a negative number.

### 4.3.2 Procedure SIGNAL

This procedure performs a SIGNAL operation on the semaphore named SEMA. The semaphore count field of the semaphore record is incremented by one, indicating that another event which requires processing has occurred. If the semaphore count field is still less than or equal to zero after the increment, there are process(es) WAITing on the semaphore, and a scheduling procedure is called. The rescheduled WAITing process either preempts the calling process or is placed in the ready queue, according to its priority. If the semaphore count field is greater than zero after being incremented, the procedure simply returns to its caller, leaving the semaphore with its one extra unprocessed (or unreceived) event. If the semaphore count overflows, the run-time support exception routine is called.

EXAMPLE:

```
        REF  SIGNAL
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE
        DATA CALL$
        DATA SIGNAL
```

EXCEPTIONS AND CONDITIONS: An exception occurs if the semaphore counter overflows or if an illegal semaphore counter is passed.

### 4.3.3 Procedure WAIT

This procedure causes a WAIT operation to be performed on the semaphore SEMA. The procedure decrements the semaphore count field of the semaphore record. If there are unprocessed (unreceived) SIGNALs to the semaphore, the procedure simply returns to the calling process. If, however, there are no unprocessed SIGNALs, the process becomes suspended on the semaphore SEMA and is placed in the semaphore queue behind any other WAITing processes. The procedure also checks that the priority (contained in the priority field of the process record) is numerically less than the semaphore priority level (contained in the semaphore level field of the semaphore record). This ensures that if the semaphore is ever associated with an interrupt level, any processes which are suspended on it are of sufficient urgency to handle the interrupt immediately. (If the semaphore was initialized by INITSE, its priority field contains 32767 so that any process may wait on it.)

EXAMPLE:

```
        REF  WAIT
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE
        DATA CALL$
        DATA WAIT
```

EXCEPTIONS AND CONDITIONS: An exception occurs if a process attempts to become suspended on a semaphore when the process priority is numerically greater than the semaphore level (i.e., it is not urgent enough to WAIT on that semaphore). An exception will also occur if an illegal semaphore is passed.

### 4.3.4 Procedure TERMSEmaphore

This procedure is used to terminate a semaphore when it is no longer to be used. IF there are no WAITing processes (i.e., the semaphore count field is not a negative integer) the procedure passes the address of the semaphore to the routine HP$FRE which reclaims the memory allocated to the semaphore into the system heap.

EXAMPLE:

```
        REF  TERMSE
             .
             .
        MOV  @<ga>,*R10+        PUSH ADDRESS OF SEMAPHORE 'SEMA'
        DATA CALL$
        DATA TERMSE
```

EXCEPTIONS AND CONDITIONS: An exception will occur if there are WAITing processes on the semaphore that is being terminated or if the semaphore is illegal.

### 4.3.5 Procedure CSIGNAl

This procedure performs a conditional SIGNAL operation on the semaphore SEMA. It first checks the validity of the semaphore, then sets the value of WAITER to false, then masks all interrupts and checks to see if any processes are WAITing on the semaphore SEMA. It does this by looking at the semaphore count field . If the semaphore count is less than zero (there are WAITing processes), the parameter WAITER is set true and the procedure branches to the SIGNAL procedure. If there are no WAITing processes, the procedure returns with WAITER set to false.

EXAMPLE:

```
        REF  CSIGNA
             .
             .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SEMA'
        MOV  @<ga>,*R10+        PUSH ADDRESS OF 'WAITER'
        DATA CALL$
        DATA CSIGNA
```

EXCEPTIONS AND CONDITIONS: An exception occurs when the semaphore is illegal.

### 4.3.6 Procedure CWAIT

This procedure performs a conditional WAIT operation on the semaphore SEMA. The procedure first checks the validity of the semaphore, then masks interrupts, and then tests the semaphore counter. If the semaphore counter is less than or equal to zero (i.e., there are no unprocessed or unreceived SIGNALs), control simply returns to the caller. If unprocessed SIGNALs exist, the semaphore counter is decremented by one, just as if a WAIT operation had been performed under similar circumstances. The parameter WAITER is set true if there was at least one unprocessed SIGNAL on the semaphore, and false if there were no unprocessed SIGNALs.

EXAMPLE:

```
        REF  CWAIT
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SEMA'
        MOV  @<ga>,*R10+        PUSH ADDRESS OF 'WAITER'
        DATA CALL$
        DATA CWAIT
```

EXCEPTIONS AND CONDITIONS: Because this procedure never results in suspension of the calling process on the semaphore, the procedure does not check the process priority; however, an invalid semaphore will be detected.

### 4.3.7 Procedure WAITSIgnal

This procedure performs a WAIT operation on the semaphore WAITFOR and a SIGNAL operation on the semaphore SIGNALTHE, in a single indivisible step. This procedure ensures that both operations are performed at once, which cannot be done by performing a SIGNAL followed by a WAIT or vice versa. In the first case, the SIGNAL might cause another process to preempt the current process before it does the WAIT; in the second, the process might become suspended when it does the WAIT before it can do the SIGNAL.

WAITSIgnal first checks the validity of the semaphore, then masks all interrupts. Next, WAITSIgnal decrements the semaphore counter of the WAITFOR semaphore. This is the essential part of the WAIT operation. If this action leaves the semaphore count greater than or equal to zero, (i.e., will not cause the process to become suspended), a signal operation is performed on the SIGNALTHE semaphore (without using another workspace). Control returns to the calling process if no process is waiting on the semaphore.

If a decrement to the semaphore count will cause suspension of the calling process, the routine $WAIT is executed. $WAIT performs a variety of functions. It verifies that the process has sufficient urgency to wait on the semaphore (i.e., that the process priority is numerically less than the semaphore level). It sets the semaphore pointer field in the process record to point to the semaphore on which the process is suspended. It places the process on the semaphore queue, stores the workspace pointer and becomes ready to perform the context switch.

Having completed the WAIT operation, but before the context switch is performed, WAITSIGNAL performs an operation similar to SIGNAL on the SIGNALTHE semaphore; it increments the semaphore counter and checks to see that it does not overflow. If there are suspended processes on the semaphore, the oldest is taken off the semaphore queue and rescheduled (according to its priority) in the same fashion as the procedure SIGNAL. If the process is an interrupt handler, it will be placed in the active queue as long as it is of equal or numerically lower

priority (i.e., equal or higher urgency) to the present head of the queue. If the rescheduled process is not put in the active queue, it is put on the queue before the last process of the same priority.

Finally, the context switch is performed and the process on the head of the ready queue (which may be the rescheduled process) becomes the currently executing process.

EXAMPLE:

```
        REF  WAITSI
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'WAITFOR'
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SIGNALTHE'
        DATA CALL$
        DATA WAITSI
```

EXCEPTION AND CONDITIONS: All the exceptions that occur under SIGNAL and WAIT; incorrect priority of a process attempting to WAIT on the semaphore WAITFOR, overflow of the semaphore counter SEMOVR in the semaphore SIGNALTHE, and an illegal semaphore.

### 4.3.8 Function SEMASTate

This function returns the state of the semaphore, which can be 0(AWAITED), 1(EMPTY), or 2(SIGNALED). It first initializes the return value to 0(AWAITED). It then inspects the semaphore counter field of the semaphore record. If this is found to be less than zero, 0(AWAITED) is the correct value to be returned. If the semaphore count field equals zero, then the returned value is 1(EMPTY). If the semaphore count is greater than zero, then the returned value is 2(SIGNALED).

NOTE: The value returned accurately reflects the state of the semaphore at the time the function was called, but the state could change immediately thereafter.

EXAMPLE:

```
        REF  SEMAST
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SEMA'
        DATA CALL$
        DATA SEMAST
        MOV  *R10,@<ga>         POP STATE
```

EXCEPTIONS AND CONDITIONS: None.

### 4.3.9 Function SEMAVAlue

This function simply returns the value in the semaphore count field of the semaphore record. A positive integer value indicates the number of unprocessed (unreceived) SIGNALs to the semaphore. A negative integer value indicates the number of processes WAITing on the semaphore. A zero value indicates that there are neither unreceived SIGNALs nor WAITing processes.

EXAMPLE:

```
        REF   SEMAVA
                .
                .
        MOV   @<ga>,*R10+       PUSH SEMAPHORE 'SEMA'
        DATA  CALL$
        DATA  SEMAVA
        MOV   *R10,@<ga>        POP SEMAPHORE VALUE
```

EXCEPTIONS AND CONDITIONS: None.

## 4.4 INTERRUPT ROUTINES

The routines listed in this subsection (with the possible exception of Procedure ASSEMBLYEVENT and Procedure NOASSEMBLYEVENT), are used to associate semaphores with hardware interrupts and perform necessary functions within the interrupt service code.

When an interrupt occurs, the interrupt handling procedure INT$PC searches for processes to handle the interrupt. INT$PC first looks for an assembly language event then for any process WAITing on the semaphore that has been designated as the primary receiver of interrupts at that level by use of the EXTERNalevent procedure. If either:

1) There is no semaphore assigned to that interrupt level as a primary receiver, or

2) There are no processes WAITing on the semaphore which has been assigned as the primary receiver of interrupts at that level,

INT$PC looks for a secondary receiver of interrupts by inspecting the semaphore associated with that level by use of the ALTEXTernalevent procedure. If no processes are found waiting, the system will crash.

Only one primary receiver and one secondary receiver are allowed to exist at a time, at any particular interrupt level.

### 4.4.1 Procedure EXTERNalevent

This procedure designates the semaphore SEMA to be the primary interrupt handler at a particular level of interrupts specified by the parameter LEVEL. $LVLCK is called to ensure that the interrupt level is in the range 1 to 15. LOWER$ is called to determine if the semaphore is associated with an external event and if the process(es) WAITing on the semaphore are of sufficiently high urgency to handle an interrupt at the level specified by LEVEL. LOWER$ also sets the level of the semaphore to that of the interrupt if it is associated with an interrupt handler.

LOWERLevel is called a second time to adjust the semaphore level field of the semaphore record of any semaphore that was previously allocated as the primary receiver of interrupts at the level specified by LEVEL. It lowers the urgency level to either the level at which this semaphore is attached, or to 32767 (the lowest possible level).

Because only one semaphore may be attached to a given interrupt level as the primary receiver at any one time, this procedure effectively does a NOEXTErnalevent at the level specified by LEVEL and then attaches a new primary interrupt handler at this level.

EXAMPLE:

```
        REF  EXTERN
               .
               .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SEMA'
        MOV  @<ga>,*R10+        PUSH LEVEL  (INTEGER)
        DATA CALL$
        DATA EXTERN
```

EXCEPTIONS AND CONDITIONS:

1) The interrupt level is invalid, i.e., outside the range 0-15.

2) Processes which are already Waiting on the semaphore are unable to handle that interrupt level.

3) The interrupt level is not allocated a dedicated workspace.

4) An illegal semaphore is detected.

### 4.4.2 Procedure NOEXTErnalevent

This procedure detaches the semaphore which has been designated as the primary receiver of interrupts at the level specified in the parameter LEVEL. In its place, the "No-event" semaphore, NOEVT is attached at this level. If no primary receiver semaphore is attached to this level, the procedure has no effect. The detached semaphore (if

any) has its level adjusted as required by the procedure LOWER$.

EXAMPLE:

```
        REF  NOEXTE
              .
              .
        MOV  @<ga>,*R10+        PUSH LEVEL  (INTEGER)
        DATA CALL$
        DATA NOEXTE
```

EXCEPTIONS AND CONDITIONS: An exception will occur if the interrupt level is outside the range 1-15.

### 4.4.3 Procedure ALTEXTernalevent

This procedure designates the semaphore SEMA to be the secondary receiver of interrupts at the level specified by the parameter LEVEL. The interrupt level must be in the range 1-15.

This procedure checks for the exception conditions (see below) and calls $LVLCK and then LOWER$ twice. LOWER$ first checks to see that the semaphore level field of the semaphore record indicates that any processes WAITing on the semaphore are of sufficiently high urgency to handle an interrupt at the level specified by LEVEL. LOWER$ then adjusts the semaphore level field of the semaphore that was previously attached as the primary receiver at this level of interrupts, as necessary.

EXAMPLE:

```
        REF  ALTEXT
              .
              .
        MOV  @<ga>,*R10+        PUSH SEMAPHORE 'SEMA'
        MOV  @<ga>,*R10+        PUSH LEVEL  (INTEGER)
        DATA CALL$
        DATA ALTEXT
```

EXCEPTIONS AND CONDITIONS: An exception occurs if:

1) The interrupt level is invalid, i.e., outside the range 1-15.

2) Processes which are already WAITing on the semaphore SEMA are unable to handle that interrupt level.

3) The interrupt level is not allocated a dedicated workspace.

4) Illegal semaphore is detected.

### 4.4.4 Procedure NOALTExternalevent

### 4.4.4 Procedure NOALTExternalevent

This procedure detaches a semaphore which is the secondary receiver of interrupts at the level specified in the parameter LEVEL. If no semaphore has been previously allocated (by ALTEXTernalevent) as the secondary receiver, this procedure has no effect.

The no event semaphore NOEVT is re-attached to this level. The level of interrupt specified in the parameter LEVEL must be in the range 1-15. This procedure calls LOWER$, which adjusts the semaphore level as necessary.

EXAMPLE:

```
        REF  NOALTE
              .
              .
        MOV  @<ga>,*R10+        PUSH LEVEL  (INTEGER)
        DATA CALL$
        DATA NOALTE
```

EXCEPTIONS AND CONDITIONS: An exception occurs if the interrupt level LEVEL is outside the range 1-15.

### 4.4.5 Function INTLEVel

This function returns a number which indicates the type of SIGNAL which activated the process. If the returned value is in the range 1-15 then an interrupt of that level activated this process. If the value returned is "-1", then activation was by another process SIGNALing the semaphore on which this process had been WAITing. If the returned value is "0", the process has not been suspended and reactivated since it was started. INTLEVel can be used by a reactivated process in order to find which interrupt level (if any) that activated it.

EXAMPLE:

```
        REF  INTLEV
              .
              .
        DATA CALL$
        DATA INTLEV
        MOV  *R10,@<ga>         POP INTERRUPT LEVEL
```

EXCEPTIONS AND CONDITIONS: None.

### 4.4.6 Procedure MASK

This procedure is called to disable interrupts. Interrupts remain masked until the procedure UNMASK is called. A return from the routine which called MASK does NOT remove the mask. The interrupt mask, bits 12 to 15 of the status register are set to zero.

EXAMPLE:

```
        REF  MASK
             .
             .
        DATA CALL$
        DATA MASK
```

EXCEPTIONS AND CONDITIONS: None.

### 4.4.7 Procedure SETMASK

This procedure sets the interrupt mask to disable all interrupts equal to or less urgent than the interrupt level passed (as parameter NEWMASK) to SETMASK in the calling sequence. The value of the previous mask (i.e., the value at which the interrupt mask was set prior to calling SETMASK) is returned to the user as an output parameter. Specifying 0 as the NEWMASK value, will cause all interrupts to be masked Calling SETMASK, specifying the previous mask as the new mask will restore the old interrupt setting. NEWMASK must be between 0 and 15 (inclusive).

EXAMPLE:

```
        REF SETMASK
            .
            .
        MOV  @<ga>,*R10+          PUSH NEW MASK
        MOV  @<ga>,*R10+          PUSH PTR TO WORD
        DATA CALL$                  WHICH WILL BE SET
        DATA SETMASK                TO OLD MASK
```

EXCEPTIONS AND CONDITIONS: None.

### 4.4.8 Procedure UNMASK

This procedure enables interrupts. It reverses the effect of the MASK procedure. The procedure first checks the priority of the current process. If that process is not an interrupt handler (i.e., it has a numerical priority of greater than 15), all interrupts are enabled by setting the interrupt mask (bits 12 to 15 of the status register) to "1's". If the process is an interrupt handler, those interrupt levels which are equally or less urgent remain inhibited. Level zero interrupts always remain enabled.

EXAMPLE:

```
        REF   UNMASK
              .
              .
        DATA CALL$
        DATA UNMASK
```

EXCEPTIONS AND CONDITIONS: None.

### 4.4.9 Procedure INT$PC

This procedure controls interrupt handling. When an interrupt occurs, control passes to this procedure, which immediately masks further interrupts.

The procedure first checks for ASSEMBlyevent and then determines if a process is suspended on the semaphore that has been designated as the primary receiver of interrupts (by EXTERNalevent). If no primary receiver of interrupts has been specified, the semaphore designated as the secondary receiver of interrupts is checked (by ALTEXTernalevent). If no processes are found suspended on an interrupt handling semaphore, the system crashes.

If a process capable of handling an interrupt is found, the oldest process is taken off the semaphore queue, and the pointer to the next process is moved to the head of the that queue. The level of interrupt is placed in the current interrupt field of the process record for debug information. The context of the current process is stored and the interrupt handler process is moved to the active queue. The context of the interrupt handler process is loaded and the interrupt handler becomes the current active process.

NOTE: The user will never need to call this procedure; therefore, no example is given.

EXCEPTIONS AND CONDITIONS: If there is no designated interrupt handler at a given interrupt level, the system crashes.

### 4.4.10 Procedure ASSEMBlyevent

When an interrupt occurs, and further interrupts are masked, internal data structures are examined to determine whether ASSEMBlyevent has been called to associate an assembly language handler with the current interrupt. If so, the workspace pointer and entry point address passed to ASSEMBlyevent in the calling sequence are used as a transfer vector to branch to the handler. The interrupt mask is zero when the handler is given control (i.e., all interrupts are masked) and must not be modified at any

time within that routine. When the assembly language handling of interrupts is complete, the user has a choice of action:

1) If no further processing of this interrupt is required, a return is made directly to the interrupt workspace by returning control to the interrupted routine. This may be done using the following sequence:

```
        ...
        LI    R14,R
R       RTWP
```

This code causes two "RTWP" instructions to be executed in a row. Note that this code is not position independent. If the assembly event handler is to be position independent, another method of setting R14 to the address of an RTWP instruction is to get the routine entry point from the interrupt trap address and add the offset of the RTWP from the routine entry point:

```
        PSEG
ENTRY   EQU   $
        ...
        MOV   @<level>*4+2,R14    GET ADDRESS OF ENTRY
        AI    R14,R-ENTRY           FROM INT TRAP AREA
R       RTWP                      DOUBLE RETURN
```

2) If the interrupt should also be processed by the Rx interrupt environment, simply execute a single "RTWP" instruction. This causes a return to the point at which the Rx transfer code would have branched had there been no assembly handler. Thus, internal data structures are examined to determine if an event semaphore has been associated with this interrupt level by a call to either EXTERNALEVENT or ALTEXTERNALEVENT, and Rx handles the interrupt.

EXAMPLE:

```
MOV  @<ga>,*R10+      ADDRESS OF INTERRUPT WORKSPACE
MOV  @<ga>,*R10+      ADDRESS OF INTERRUPT ROUTINE
MOV  @<ga>,*R10+      INTERRUPT LEVEL
DATA CALL$
DATA ASSEMB
```

EXCEPTIONS AND CONDITIONS: An exception occurs when a bad LEVEL parameter is passed.

### 4.4.11 Procedure NOASSEmblyevent

This procedure is called to disassociate an interrupt level with an assembly language handler.

EXAMPLE:

```
MOV @<ga>,*R10+          LEVEL OF HANDLER TO BE REMOVED
DATA CALL$
DATA NOASSE
```

EXCEPTIONS AND CONDITIONS: An exception occurs when a bad LEVEL parameter is passed.

## 4.5 PROCESSOR MANAGEMENT ROUTINES

These routines are used to reschedule the execution of multiple process systems and locate the current process record.

### 4.5.1 Procedure SETPRIority

This procedure modifies the priority of the most urgent, non-interrupt process. It ensures that NEWVALUE is in the range 16-32767 and masks all interrupts, scanning the ready queue and active queue for the first non-interrupt handler (i.e., a process with a numerical priority greater than 15). If one is not found, the parameter OLDVALUE is returned as zero, otherwise the old process priority is returned. The new process priority is then loaded from NEWVALUE and compared with the old value. If the new value is numerically greater than the old, the procedure SWAP is called to re-schedule the process. This routine is used to force rescheduling of the most urgent non-interrupt process.

EXAMPLE:

```
REF  SETPRI
       .
       .
MOV  @<ga>,*R10+         PUSH ADDRESS OF 'OLDVALUE'
MOV  @<ga>,*R10+         PUSH ADDRESS OF 'NEWVALUE'
DATA CALL$
DATA SETPRI
```

EXCEPTIONS AND CONDITIONS: An exception will occur if the NEWVALUE is outside the range 16-32767.

### 4.5.2 Procedure SWAP

This procedure reschedules the current non-interrupt process (i.e., the process nearest the head of the ready queue or the active process with a priority numerically greater than 15). The

process that is being SWAPped is placed in the ready queue behind the last process of the same priority. This means that if there is more than one process with the same priority as the currently active one (and it is a noninterrupt process), a SWAP operation will cause a new process to become the currently active one.

The following example illustrates a swap operation when the current process is resheduled. In this example each process is represented by its priority and a letter indicating initial sequence.

```
                                  Current process
                                        :
  Ready queue (before SWAP):       20a 20b 20c  23d 23e 25f
                                        :
  Ready queue (after SWAP):        20b 20c 20a  23d 23e 25f
```

The SWAP operation may be used to allocate execution time slices to different processes. This time slicing is implemented by the CLKINT process described later.

EXAMPLE:

```
        REF  SWAP
               .
               .
        DATA CALL$
        DATA SWAP
```

EXCEPTIONS AND CONDITIONS: None.

## 4.6 MEMORY MANAGEMENT PROCEDURES

These routines are used to perform dynamic management of the heap packets of memory.

### 4.6.1 Procedure NEW$

This process allocates a contiguous area from the current process's heap of LENGTH or more words and returns a pointer to the area in pointer. This memory may then be used by the calling process until released by use of the FREE$ procedure.

EXAMPLE:

```
        REF   NEW$
              .
              .
        MOV   @<ga>,*R10+       PUSH ADDRESS OF POINTER
        MOV   @<ga>,*R10+       PUSH 'LENGTH' IN WORDS
        DATA  CALL$
        DATA  NEW$
```

EXCEPTIONS AND CONDITIONS: If the heap area cannot be allocated, a zero value pointer is returned.

### 4.6.2 Procedure FREE$

This procedure releases an area of heap allocated by the NEW$ procedure. The pointer to the heap packet is passed to this procedure which sets it equal to zero.

EXAMPLE:

```
        REF   FREE$
              .
              .
        MOV   @<ga>,*R10+       PUSH ADDRESS OF POINTER
        DATA  CALL$
        DATA  FREE$
```

EXCEPTIONS AND CONDITIONS: None.

## 4.7 CLOCK MANAGEMENT ROUTINES

The clock management routines time events or ensure specific time delays in the user's system. Whenever any real time operations are required the user must first start CLKINT process. (Note that the Rx clock is NOT a time of day clock, but rather an internal timer.)

### 4.7.1 Process CLKINT

This process performs the following three functions:

1) Initialize the system clock to 00000000 milliseconds,

2) Provide 'time out' signals that a specified time interval has occured, with a resolution of 'n' milliseconds per interrupt,

3) Implement time slicing between processes by calling SWAP every 'n' milliseconds. If 'n' is zero, no time slicing is performed.

This process must be started whenever a system clock is required. There is some processor overhead when using the clock routines, and the overhead will increase as the number of time elements waiting increases. The overhead is also inversely proportional to the number of milliseconds per timer interrupt: the lower the numbr of milliseconds per interrupt, the greater the overhead.

The system clock uses a double integer (32 bits) contained in the clock record. A pointer to the head of a queue of time element records is also maintained in the clock record. (Clock and clock service records are described in Appendix A). Note that a workspace for level three interrupts is provided in the CONFIG module by default.

Time elements are put on the queue by the TWAIT procedure, and may be signaled by other user processes.

EXAMPLE:

```
        REF  CLKINT
               .
               .
        MOV  @<ga>,*R10+      PUSH MILLISECONDS PER INTERRUPT
        MOV  @<ga>,*R10+      PUSH MILLISECONDS PER SWAP (IF 0, NO SWAPPING
        MOV  @<ga>,*R10+      PUSH 9901 BASE (USUALLY >100)
        DATA CALL$
        DATA CLKINT
```

EXCEPTIONS AND CONDITIONS: Must have workspace for level three interrupts.

### 4.7.2 Procedure TWAIT

This procedure is used to suspend a process for a specified time interval or until another process has signaled an event, whichever comes first. This feature is necessary when only a certain amount of time can pass before the event should have occured (such as I/O).

The time interval is a two word, signed positive integer value. The most significant word is pushed first, followed by the least significant word.

This procedure builds a time element from the parameters it is passed, and then proceeds to place it in the clock's time queue. It then performs a WAIT on the semaphore. If the semaphore is signaled by the clock process, it sets the status word to zero and returns. If the semaphore was signaled from another user process (i.e., before the time was up), then the procedure sets the status word to one.

Note that the resolution of the clock is user-specified in CLKINT and therefore a delay request will suspend a process for 'n' ms (where 'n' is a multiple of the user-specifed clock resolution).

RETURN CODES:

0 SIGNALLED
1 TIMED-OUT
2 THERE WAS ANOTHER WAITER
3 THE "TIME TO WAIT" IS INVALID, OR THE SEMAPHORE IS INVALID.

EXAMPLE:

```
REF  TWAIT
       .
       .
MOV  @<ga>,*R10+     PUSH THE ADDRESS OF THE SEMAPHORE
MOV  @<ga>,*R10+     PUSH FIRST WORD OF TIME PARAMETER
MOV  @<ga>,*R10+     PUSH SECOND WORD OF TIME PARAMETER
MOV  @<ga>,*R10+     PUSH THE ADDRESS OF STATUS
DATA CALL$
DATA TWAIT
```

EXCEPTIONS AND CONDITIONS: The process CLKINT must be started prior to the use of this procedure. There must be no other processes waiting on the semaphore.

### 4.7.3 Procedure DELAY

This procedure causes the user process to suspend execution for a specified number of milliseconds. This delay is achieved by calling TWAIT with a specially allocated DELAY semaphore. This semaphore is allocated once, and is thereafter re-used by DELAY, and is reclaimed by the CLK$TE routine when the process terminates.

EXAMPLE:

```
REF  DELAY
       .
       .
MOV  R0,*R10+     PUSH FIRST WORD OF TIME PARAMETER
MOV  R1,*R10+     PUSH SECOND WORD OF TIME PARAMETER
DATA CALL$
DATA DELAY
```

EXCEPTIONS AND CONDITIONS: The process CLKINT must be started prior to the use of this procedure.

## 4.8 ERROR REPORTING PROCEDURE EXCEPTion

This procedure performs an error trap from a process. The values

of CLASSCODE and REASONCODE are placed in the error fields of the caller's process record and the run-time support exception routine is called.

EXAMPLE:

```
        REF  EXCEPT
               .
               .
        MOV  @<ga>,*R10+          PUSH 'CLASSCODE' IN MSB
        MOV  @<ga>,*R10+          PUSH 'REASONCODE' IN MSB
        DATA CALL$
        DATA EXCEPT
```

EXCEPTIONS AND CONDITIONS:
None.

# SECTION V

# CHANNEL ROUTINES

## 5.1 GENERAL

Rx makes it possible for the user to create and pass messages between processes using the concept of channels. Channels can be thought of as data structures over which messages (data) can be sent and received by processes located at either end (see Figure 5-1). Initialization of the channel, construction of the message to be sent, and synchronization of the actual message transfer are performed using the Channel Routines described in the following subsections.

In the normal start-up sequence for transmission between producer and consumer processes, each process issues a C$INIT on the same channel name. (A process MUST initialize a channel in order to send messages over that channel.) A call is then made to C$ALLOC by the process sending the message to allocate a heap packet for passing the text data.

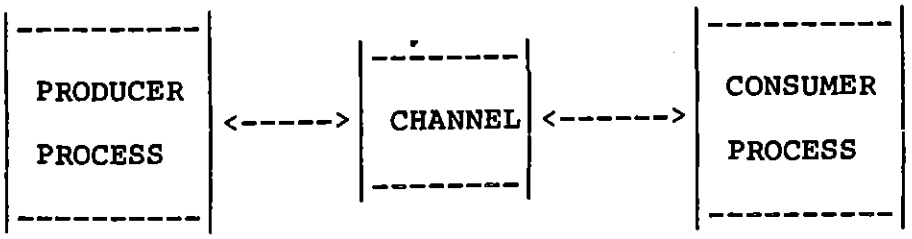



FIGURE 5-1. PROCESS COMMUNICATION VIA CHANNELS

The following is a skeletal outline for a typical message transmission between producer/consumer processes. For a complete example, see the demonstration program in Appendix E.

```
SYSTEM:

        REF   PRODUC
        REF   CONSUM
                .
                .
        DATA  CALL$,PRODUC
        DATA  CALL$,CONSUM
```

```
PRODUCER:

        REF   C$INIT
        REF   C$TERM
        REF   C$ALLO
        REF   C$SEND
        REF   C$WAIT
              .
              .
        <call C$INIT to connect to channel>
        <call C$ALLO to allocate message buffer>
        loop:
           <fill message buffer>
           <call C$SEND to send message>
           <call C$WAIT to wait for acknowledgement>
        loop exit:
        <call C$TERM to disconnect fom channel>

CONSUMER:

        REF   C$INIT
        REF   C$TERM
        REF   C$RECE
        REF   C$ACKN
              .
              .
        <call C$INIT to connect to channel>
        loop:
           <call C$RECE to send message>
           <process data in message buffer>
           <call C$ACKN to wait for acknowledgement>
        loop exit:
        <call C$TERM to disconnect fom channel>
```

In the previous example, the producer first initializes a channel that will be used to send the message to the consumer, then allocates memory for that message using the C$INIT and C$ALLO routines respectively. The C$SEND routine actually sends the message to the consumer, while C$WAIT will suspend the producer (i.e., the producer is placed in a WAIT queue) until an acknowledgement of the message is received. (Reference Section 2, 2.3.2 for detailed information on semaphore queues.)

The consumer initializes the same channel as the producer via the C$INIT routine and then calls C$RECE to suspend until a message is received over the intialized channel. When the message is received, the consumer performs whatever action is necessary to process the data. The consumer may send messages back to the producer by modifying the message that was sent. When the consumer has processed the message, it calls C$ACKN to notify the producer that the message has been received/processed, and the producer is released from suspension.

Finally, when each process is finished, it calls C$TERM to disconnect from the channel. When the last process disconnects from a particular channel, all message buffers and other data structures associated with that channel are freed.

When dealing with channels, it is important to remember that a channel will not exist unless it is designated (named) by a process as existing (via C$INIT), and that in order to speak to another process, the receiving process must designate (name) the same channel. Two terms are used: channel "name" and channel identifier or "ID". The name of a channel is a number from 1 to 32767 which is passed to C$INIT. C$INIT returns a channel ID, which is a pointer to the channel data structure.

## 5.2 CHANNEL ROUTINE DESCRIPTIONS

The following subsections will list and describe each channel routine, as well as provide example calling sequences for each.

### 5.2.1 Procedure C$ACKN

This routine acknowledges that a message has been received and/or processed, and notifies the sending process of that fact.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+       PUSH ADDRESS MESSAGE TEXT POINTER
DATA CALL$,C$ACKN
```

EXTERNAL ROUTINES: C$$HEADER, RT$ENTER, RT$EXIT, SIGNAL

### 5.2.2 Procedure C$ALLO

The C$ALLO routine allocates a heap packet for passing text data between processes. The heap packet has a header of fixed size which contains information used to synchronize interprocess communication. The header is followed by a text data field containing any message to be transmitted. The maximum number of characters allowed in the message is defined in the first parameter of the calling sequence.

CALLING SEQUENCE:

```
MOV @<ga>,*R10+        PUSH MESSAGE SIZE (INTEGER)
MOV @<ga>,*R10+        PUSH POINTER TO HEAP PACKET
DATA CALL$,C$ALLO
```

EXTERNAL ROUTINES: C$$MSG, RT$ENTER, RT$EXIT, CKSEMAPHORE, INITSEMAPHORE, TERMSEMAPHORE, HP$FREE, HP$NEW, HP$SYSTEM

### 5.2.3 Procedure C$CRECEive

This routine checks that a message has been sent to a channel. If one is present, the routine returns its address. If no message has been sent, a message pointer set to nil is returned.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+            PUSH CHANNEL ID
MOV  @<ga>,*R10+            PUSH MESSAGE TEXT POINTER
DATA CALL$,C$CRECE
```

EXTERNAL ROUTINES: SETMASK, C$$MSG, RT$ENTER, RT$EXIT, CWAIT

### 5.2.4 Procedure C$CWAI

This routine conditionally waits for a sent message to be acknowledged. If the message has been acknowledged, the status word is set to TRUE (all ones). If the receiving process does not acknowledge, the status word is set to FALSE (zero).

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+            PUSH MESSAGE TEXT POINTER
MOV  @<ga>,*R10+            PUSH PTR TO STATUS WORD
DATA CALL$,C$CWAI
```

### 5.2.5 Procedure C$DISPose

This procedure deallocates a channel message.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH MESSAGE TEXT POINTER
DATA CALL$,C$DISP
```

EXTERNAL ROUTINES C$$HEADER, RT$ENTER, RT$EXIT, TERMSEMAPHORE
HP$FREE, HP$SYS

### 5.2.6 Procedure C$INIT

C$INIT searches the channel directory for the channel identified by <name>. If the channel is found, C$INIT sets the ID to point to the channel. If the channel is not found, a channel is created, inserted into the channel directory, and the ID is set to point to it. If the passed value of <name> is "0", the channel is assumed to have been created either in COMMON, or in other memory not declared available to the system. Although the channel does not appear in the channel directory, the routines using ID as an identifier will operate correctly on such a channel.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH CHANNEL NAME
MOV  @<ga>,*R10+          PUSH POINTER TO CHANNEL ID
DATA CALL$,C$INIT
```

EXTERNAL ROUTINES: RT$ENTER, RT$EXIT, CKSEMAPHORE, INITSEMAPHORE, SIGNAL, WAIT, HP$FREE, HP$NEW, NP$SYSTEM, MY$MPX

### 5.2.7 Procedure C$NOTI

C$NOTIFY is used to set the channel "Notify" semaphore field to point to a user-supplied semaphore. This allows a consuming process waiting on a single semaphore associated with several conditions to receive notification that a message is present on the channel. The consuming process must perform a receive to get control of the message buffer. When several channels are used, the consumer can perform a conditional receive to determine the location of the message.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH CHANNEL ID
MOV  @<ga>,*R10+          PUSH ADDRESS OF SEMAPHORE
DATA CALL$,C$NOTI
```

EXTERNAL ROUTINES: None.

### 5.2.8 Procedure C$RECEive

This routine causes a process to suspend until a message is sent to the channel. It then takes the message from the channel and sets <msg> to point to that message's text field.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH CHANNEL ID
MOV  @<ga>,*R10+          PUSH MESSAGE TEXT POINTER
DATA CALL$,C$RECE
```

EXTERNAL ROUTINES: C$$MSG, SETMASK, RT$ENTER, RT$EXIT, WAIT

### 5.2.9 Procedure C$SEND

C$SEND sends a message to the channel and signals that a message is present for processing. The oldest pending C$RECEive on this channel will be activated.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH CHANNEL ID
```

```
MOV  @<ga>,*R10+          PUSH MESSAGE TEXT POINTER
DATA CALL$,C$SEND
```

EXTERNAL ROUTINES: C$$HEADER, RT$ENTER, RT$EXIT, SIGNAL

### 5.2.10 Procedure C$TERM

This procedure disconnects the calling process from the channel. When the last process is disconnected, the routine closes the data structures associated with the channel, terminates the channel pointed to by <c>, and updates the directory to reflect the termination.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH CHANNEL ID
DATA CALL$,C$TERM
```

EXTERNAL ROUTINES: RT$ENTER, RT$EXIT, SIGNAL, WAIT, TERMSEMAPHORE, HP$FREE, HP$SYSTEM, MY$MPX

### 5.2.11 Procedure C$WAIT

C$WAIT waits for a message to be acknowledged by a consuming process. No further use of the message is allowed until an acknowledgment is received from the consumer (via C$ACKNO).

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH MESSAGE TEXT POINTER
DATA CALL$,C$WAIT
```

EXTERNAL ROUTINES: C$$HEADER, RT$ENTER, RT$EXIT, WAIT

### 5.2.12 Function C$$HEA

C$$HEADE, given a pointer to a message text field, returns a pointer to that message's header. This function is normally used only by the other channel routines.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH MESSAGE HEADER POINTER
DATA CALL$,C$$HEA
```

EXTERNAL ROUTINES: LOCATION, SIZE

### 5.2.13 Procedure C$$MSG

C$$MSG, given a pointer to a message header, returns a pointer to the text field of that message. Upon return, the stack pointer (R10)

points to a word containing the address of the text field of the message.

CALLING SEQUENCE:

```
MOV  @<ga>,*R10+          PUSH MESSAGE HEADER POINTER
DATA CALL$,C$$MSG
```

EXTERNAL ROUTINES: LOCATION, SIZE

# SECTION 6

# CONFIGURING TARGET SYSTEMS FOR OBJECT CODE EXECUTION

## 6.1 GENERAL

The user has the capability to customize Rx, modeling the system to fit his or her application requirements by producing a load module that includes the application routines and those processes, procedures, and functions supplied in the Rx library that will enable the application to execute.

The link editor of the user's software development system is used to create the load module. The configuration process involves giving a simple description of the target machine to identify ROM/RAM addresses and the location of the target machine's restart vector. The user's own interrupt handlers and system crash handler may also be included. The result will be a 9900 load module which may be debugged using AMPL or the Rx Standalone Debugger.

The steps required to produce a customized load module will be fully explained in following sections. They are:

1) Assemble user source.

2) Customize and assemble CONFIG. (Section 6.2)

3) Create a link edit control file. (Section 6.5)

4) Execute linkage editor. (Section 6.5.2)

5) Test using the debugger of the user's choice. (Sections 7 and 8)

## 6.2 CUSTOMIZING THE CONFIG MODULE

Configuration of a target system requires that the user build a simple specification of the target machine into the Rx module called "CONFIG". CONFIG contains the specification of the system's RAM organization and the locations of the system RESTART and LREX vectors.

NOTE: Because of the symbols used to define the Ram Table, the warning message 'VALUE TRUNCATED' will be sent to the user's error file during assembly of CONFIG. The user can ignore this message.

Figure 6-1 is the version of CONFIG supplied the user. It specifies that RAM is located from >5000 to >A000. Note that this

version of CONFIG may not contain the correct specifications for the system being configured. Information on the appropriate modifications may be found in following subsections.

```
        IDT   'CONFIG'
        IDT   'CONFIG'              SPECIFY CONFIGURATION
* REVISION: 08/01/80  1.00         ORIGINAL FOR RX 2.0
* ROUTINE LIST: CONFIG, IWP$0 .. IWP$15, BAD$WP,
*                $RAMTB, $RESTA, $LREX, $SYSCR,
*                $DEFAU, $FILL, $STKSZ, $BOOTP,
*                $IODIR, DB$WP
* COPY MODULES:
*       NONE.
* MACRO DEFINITIONS:
*       NONE.
* EXTERNAL ROUTINES:
*       NONE.
* EXTERNAL DATA:
        PSEG
* MODULE CONSTANTS:
IWPSZ   EQU   24                    SIZE OF AN INTERRUPT
*                                     WORKSPACE (R4-R15)
LOWRAM  EQU   >5000                 LOW BOUNDARY OF RAM
* MODULE VARIABLES:
*
        DORG  LOWRAM
*
        DEF   IWP$0,IWP$1,IWP$2,IWP$3
        DEF   IWP$4,IWP$5,IWP$6,IWP$7
        DEF   IWP$8,IWP$9,IWP$10,IWP$11
        DEF   IWP$12,IWP$13,IWP$14,IWP$15
        DEF   BAD$WP,DB$WP
IWP$0   BSS   32
IWP$1   BSS   32
DB$WP   EQU   IWP$1
IWP$2   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$3   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$4   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$5   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$6   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$7   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$8   EQU   $-32+IWPSZ
        BSS   IWPSZ
```

FIGURE 6-1. CONFIG MODULE (Sheet 1 of 4).

```
IWP$9   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$10  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$11  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$12  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$13  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$14  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$15  EQU   $-32+IWPSZ
        BSS   IWPSZ
BAD$WP  BSS   32
*
LOWHP   EQU   $
*
        RORG
        TITL 'CONFIG:          SPECIFY CONFIGURATION'
        6-
* ABSTRACT:
*       SPECIFY CERTAIN SYSTEM PARAMETERS, THE RAM
*       CONFIGURATION, AND THE I/O SUBSYSTEM
*       DIRECTORY.
* CALLING SEQUENCE:
*       NONE.
* EXCEPTIONS AND CONDITIONS:
*       NONE.
* LOCAL DATA:
*       NONE.
* ENTRY POINT:
*       NONE.
*****************************************************************
* ADDRESS OF THE "BLWP" VECTOR FOR RESTARTS; USE "0" FOR
* LEVEL 0 INTERRUPT, ">FFFC" FOR THE "LREX" VECTOR, OR
* THE ADDRESS OF A USER-DEFINED VECTOR.
*****************************************************************
        DEF   $RESTA
$RESTA  DATA  0
```

FIGURE 6-1. CONFIG MODULE (Sheet 2 of 4).

```
*************************************************************
* ADDRESS OF THE "BLWP" VECTOR FOR THE "LREX" INSTRUCTION;
* USE "0" IF THERE IS TO BE NO "LREX" VECTOR OR IF HIGH
* MEMORY IS ROM.
*************************************************************
       DEF  $LREX
$LREX  DATA 0
*************************************************************
* ADDRESS OF THE USER-DEFINED ROUTINE TO BE INVOKED IN CASE
* OF A SYSTEM CRASH; USE "0" FOR THE SYSTEM DEFAULT WHICH
* IS TO MASK INTERRUPTS AND IDLE THE PROCESSOR.
*************************************************************
       DEF  $SYSCR
$SYSCR DATA 0
*************************************************************
* ADDRESS OF THE MPP ROUTINE TO BE INVOKED IF AN EXCEPTION
* OCCURS BUT NO EXCEPTION HANDLER HAS BEEN SPECIFIED; USE
* "0" FOR THE SYSTEM DEFAULT WHICH IS A "NO EXCEPTION
* HANDLER" SYSTEM CRASH.
*************************************************************
       DEF  $DEFAU
$DEFAU DATA 0
*************************************************************
* THIS IS THE VALUE WITH WHICH THE HEAP WILL BE
* INITIALIZED AT POWER-UP.
*************************************************************
       DEF  $FILL
$FILL  JMP  $
*************************************************************
* THIS IS THE DEFAULT STACK SIZE (IN WORDS) THAT IS USED
* IF A "STACKSIZE" CONCURRENT PARAMETER IS NOT SPECIFIED.
*************************************************************
       DEF  $STKSZ
$STKSZ DATA >100
*************************************************************
* THE PARAMETER LIST FOR THE CALL TO "S$PRCS" TO START THE
* "BOOT" PROGRAM.
*************************************************************
       DEF  $BOOTP
$BOOTP DATA >0000              FRAME SIZE
       DATA >0000              LEXICAL NESTING LEVEL
       DATA >0000              PRIORITY
       DATA >0100              STACK SIZE
       DATA >0000              HEAP SIZE
```

FIGURE 6-1. CONFIG MODULE (Sheet 3 of 4).

```
*****************************************************************
* ADDRESS OF THE "RAM TABLE," THE TABLE THAT DESCRIBES THE
* REGIONS OF READ-WRITE MEMORY TO BE COLLECTED INTO THE
* HEAP.
*****************************************************************
       DEF  $RAMTB
$RAMTB DATA RAMTB
*****************************************************************
* ADDRESS OF THE DIRECTORY OF I/O SUBSYSTEMS.
*****************************************************************
       DEF  $IODIR
$IODIR DATA IODIR
*****************************************************************
* THE FOLLOWING TABLE IS A LIST OF "LENGTH_IN_BYTES,
* STARTING_ADDRESS" PAIRS THAT DEFINE THE RAM TO BE USED
* BY THE EXECUTIVE; A WORD OF "0" TERMINATES THE LIST.
* THE RAM REGIONS MUST BE IN ASCENDING ORDER AND MUST NOT
* OVERLAP.
*****************************************************************
RAMTB  DATA >A000-LOWHP,LOWHP
       DATA 0                      LIST TERMINATOR
*****************************************************************
* THE FOLLOWING TABLE IS A LIST OF "SERVICE_DIRECTORY,
* PORT_CONSTANTS" PAIRS THAT DEFINE THE I/O SUBSYSTEM TO
* BE INITIALIZED WHEN ROUTINE "D$INIT" IS CALLED;
* A WORD OF "0" TERMINATES THE LIST.
*****************************************************************
IODIR  EQU  $
*
*      INSERT LIST ENTRIES HERE.
*
       DATA 0                      LIST TERMINATOR
*
       END
```

FIGURE 6-1. CONFIG MODULE (Sheet 4 of 4).

### 6.2.1 Specification of System Parameters

A number of target system parameters have been collected into CONFIG so they can be conveniently modified by the user. Each parameter is described in one of the following paragraphs.

Parameter $RESTA is a data word that contains the address of a transfer vector for a BLWP instruction that will be executed if the procedure RE$START is invoked. A value of "0" will cause a level 0 interrupt to be simulated; a value of ">FFFC" will simulate a LREX instruction. Special restart processing may be specified via a user-defined transfer vector.

An LREX instruction causes a trap through the transfer vector at location >FFFC; it is often used for reloading or "warm starting" a system in which the level 0 interrupt is used for a power-up or "cold start". If high memory is in RAM, then the LREX vector must be initialized at run-time. If the data word $LREX contains a non-zero value, it is interpreted as the address of a (ROM) transfer vector that Rx will copy to >FFFC through >FFFF during system initialization. If the data word is 0 (i.e., high memory is ROM, or no LREX instructions will be used), no copy will be made.

Parameter $SYSCR permits the user to specify the action to be taken if a system crash occurs. A non-zero value of the data word $SYSCR is interpreted by the Executive Run Time Support as the address of an assembly language routine that will be invoked (via a BL instruction with register R0 containing the crash code) in case of a unrecoverable error. A value of zero results in a default routine being invoked that masks all interrupts and executes the IDLE instruction.

Parameter $DEFAU permits specification of a default exception handler that will be invoked if an exception occurs in a process for which no exception handler has been established. If a default is to be used, then the data word $DEFAU must contain the entry point address of that routine. A value of zero will cause a "no exception handler" system crash to occur.

The value in the data word $FILL is the pattern with which the heap will be initialized at power-up; the suggested value is the instruction "JMP $" (Hex value >10FF) which will sometimes stop errant execution.

Parameter $STKSZ is the default stack size (in words) that will be used if a "stacksize" concurrent parameter is not specified for process. To do this, the process start code (see 4.1.5) needs to be changed to this:

```
      REF S$PRCS
            .
            .
      MOV @<ga>,*R10+       PUSH FRAME SIZE IN BYTES
      MOV @<ga>,*R10+       PUSH LEXICAL LEVEL
      MOV @<ga>,*R10+       PUSH PROCESS PRIORITY
      MOV @$STKSZ,*R10+     USE DEFAULT STACK SIZE
      MOV @<ga>,*R10+       PUSH HEAP SIZE IN WORDS
```

The five words labeled $BOOTP are the parameters to the process creation routine S$PRCS that creates the "boot" program. The Executive Run Time Support begins execution in a program that "bootstraps" the system into execution by initializing system data structures and then invoking the "ghost" procedure GHOST$ which the user must customize to perform application dependent initialization and start the user's system (Section 6.3). Since the processing that is performed in the boot program is application-dependent, its stack size parameter in the $BOOTP parameter list may have to be adjusted by the user; the other four parameters will not require modification. (Since the stack region of the boot program will be reclaimed when procedure GHOST$ returns and the program terminates, the estimated stack size need not be exact.)

Parameter $RAMTB is a data word containing the address of the "RAM table" that is described in the following subsection.

Parameter $IODIR is a data word containing the address of the I/O subsystem directory that is described in 6.2.3.

### 6.2.2 Specification of RAM Locations

The module CONFIG contains a two part description of RAM of the target system. The symbol LOWRAM must be equated to the low boundary of RAM that is to be managed by Rx. The first part of the RAM description declares static data structures that are not to be included in the heap. This area begins at LOWRAM and contains interrupt and XOP workspaces. Any user-declared static (not COMMON) storage (e.g., LREX workspace) should be declared following these structures but preceding the symbol LOWHP which marks the end of the static data area and the beginning of the dynamically allocated heap area. The "RAM table" is a structure that contains the addresses and sizes of regions of RAM that are to be used for heap allocation. Each region is described by a pair of values, the first of which is the size of the segment and the second is its beginning address; a size of zero terminates the list. (Regions must be in ascending order and must not overlap.) The RAM table for a target system with RAM from Hex addresses >4000 to 9FFF, and >F000 to >FFFF looks like Figure 6-2.

```
RAMTB   DATA >6000,>4000    4000 - 9FFF
        DATA >1000,>F000    F000 - FFFF
        DATA 0
```

FIGURE 6-2. SIMPLE RAM TABLE

Figure 6-3 shows how this RAM table would be incorporated into CONFIG so the static area would be allocated from the first RAM region. The expression ">A000-LOWHP" calculates the space remaining in the first region after the static areas are allocated.

```
LOWRAM EQU  >4000
       ...
       DORG LOWRAM
*
IWP$0  BSS  32
       ...
LOWHP  EQU  $
       ...
RAMTB  DATA >A000-LOWHP,LOWHP
       DATA >1000,>F000
       DATA 0
       ...
```

FIGURE 6-3. USE OF RAM TABLE IN CONFIG MODULE.

COMMON regions of memory may be used by more than one routine to eliminate the passing of certain parameters. This may be done using the CSEG assembler directive. Common regions must exist in regions of RAM outside the Rx system heap to ensure the executive cannot allocate the region for other purposes. Therefore, CONFIG must not include any memory to be used as a common region; the value equated to COMMON should be changed so that memory for the common is not included in the RAM table, and the link control file (described in 6.5.1) should be changed to specify to specify the beginning of RAM.

Use of COMMON is not generally regarded as good programming practice, and should be avoided if possible.

### 6.2.3 Specification of the I/O Subsystem Directory

In his GHOST$ procedure, the user has the option to include a call to Procedure D$INIT, causing automatic initialization of I/O subsystems at power-up. The specific subsystems to be initialized must be enumerated in the I/O directory table in CONFIG. Each

subsystem is described by a pair of values. The first is the address of the service directory that defines the entry points of those routines that provide subsystem services. The second value is the address of the "port constants" associated with the subsystem. The directory is terminated by a value of zero where a service directory address is expected.

Figure 6-4 depicts a sample I/O subsystem directory that will cause two subsystems to be initialized automatically. A record-oriented terminal subsystem is specified by its service directory (T02$SD) and port constants (T02$PC). The interprocess communication subsystem is specified by its service directory (IPC$SD) and a NIL (0) port constants address.

```
IODIR  EQU  $
*
       REF  T02$SD,T02$PC
       DATA T02$SD,T02$PC
*
       REF  IPC$SD
       DATA IPC$SD,0
*
       DATA 0                    LIST TERMINATOR
```

FIGURE 6-4. I/O SUBSYSTEM DIRECTORY.

### 6.2.4 Example CONFIG Module

As an example consider the following system:

1) RAM in locations >B000 to >BFFF and >D000 to >DFFF

2) ROM in locations >0000 to >9FFF, >C000 to >CFFF and >FF00 to >FFFF

3) A user-defined restart routine. This routine requires a workspace (BGN$WP) and has an entry point (BGN$PC).

4) I/O subsystems for terminal communication and interprocess communication.

Figure 6-5 shows how the pertinent portions of CONFIG might be specified for this system.

```
          ...
LOWRAM    EQU   >B000              LOW BOUNDARY OF RAM
          ...
IWP$0     BSS   32
          ...
BAD$WP    BSS   32
*
BGN$WP    BSS   32
*
LOWHP     EQU   $
*
          ...
          DEF   $RESTA
$RESTA    DATA  RESTA
          ...
$RAMTB    DATA  RAMTB
          ...
$IODIR    DATA  IODIR
          ...
RAMTB     DATA  >C000-LOWHP,LOWHP
          DATA  >1000,>D000
          DATA  0                  LIST TERMINATOR
          ...
IODIR     EQU   $
*
          REF   T02$SD,T02$PC
          DATA  T02$SD,T02$PC
*
          REF   IPC$SD
          DATA  IPC$SD,0
*
          DATA  0                  LIST TERMINATOR
*
RESTA     DATA  BGN$WP,BGN$PC
          REF   BGN$PC
          ...
```

FIGURE 6-5. EXAMPLE CONFIG MODULE.

The RAM table of Figure 6-5 reflects the two RAM memory segments. Notice that the ROM memory segment addresses have no effect on CONFIG. The workspace BGN$WP has been declared in the static area of CONFIG, and the word $RESTA now points to the transfer vector labeled RESTA.

The variable length and user-defined structures have been added at the end of the ROM section of CONFIG. This is done to enable changes to be made to these structures without requiring that ROMs be reconstructed that reference the CONFIG module.

## 6.3 CUSTOMIZING THE "GHOST" PROCEDURE

Rx begins execution in a program called BOOT$ that "bootstraps" the system into execution by initializing system data structures and then invoking the "ghost" procedure GHOST$. The ghost procedure is obligated to START the user's SYSTEM module; it may be customized to perform application-dependent initialization. Figure 6-6 lists the default version of GHOST$ that is supplied with the Executive Run Time Support. (Procedure D$INIT is part of the File I/O subsystem components. This procedure will only need to be called if File I/O subsystems will be supported. (Reference the Device Independent File I/O Package manual, MP386 for detailed information). Procedure MSG$INIT is called to identify the pathname of the device that is to receive the output of the standard procedure MESSAGE. The "start systm$" statement activates the user's system since all system modules are given the entry point SYSTM$.

For most applications the default version of GHOST$ will be adequate. If certain initialization must be performed for a class of applications (e.g. special devices that must be initialized), it is appropriate that it be performed in the ghost procedure so it need not be repeated in each application. If it is known that I/O will not be used, then a slight saving in code space can be made by removing the calls to D$INIT and MSG$INIT in GHOST$. (If the I/O support library is not specified at link edit time, D$INIT and MSG$INIT will be resolved by "dummy" routines that perform no processing.)

```
*
        IDT  'GHOST$  '                          08/01/80  11:18:47
*
*
        DEF  GHOST$
        REF  D$INIT
        REF  MSG$IN
        REF  SYSTM$
        REF  CALL$
        REF  EXIT$P
*                                                  LC     HEX    CHAR
        PSEG
GHOST$  EQU  $
PR      EQU  R7
CODE    EQU  R8
LF      EQU  R9
SP      EQU  R10
L0      EQU  $
        DATA L0014-L0
        DATA L0036-L0
        DATA >0008                                0004    0008     ..
        DATA >0008                                0006    0008     ..
        DATA >4F50                                000A    4F50     OP
        DATA >4552                                000C    4552     ER
        DATA >4154                                000E    4154     AT
        DATA >4F52                                0010    4F52     OR
D0012   DATA >0008                                0012    0008     ..
*                                                   LC    WORD(S)
L0014   EQU  $
        DATA CALL$,D$INIT                         0014
        MOV  CODE,R15                             0018    C3C8
        AI   R15,>000A                            001A    022F     000A
        MOV  LF,R12                               001E    C309
        MOV  *R15+,*R12+                          0020    CF3F
        MOV  *R15+,*R12+                          0022    CF3F
        MOV  *R15+,*R12+                          0024    CF3F
        MOV  *R15+,*R12+                          0026    CF3F
        MOV  LF,*SP+                              0028    CE89
        MOV  @D0012-L0(CODE),*SP+                 002A    CEA8     0012
        DATA CALL$,MSG$IN                         002E
        DATA CALL$,SYSTM$                         0032
L0036   EQU  $
        B    @EXIT$P                              0036    0460     0000
        END
```

FIGURE 6-6. DEFAULT VERSION OF PROCEDURE GHOST$.

## 6.4 ASSEMBLY LANGUAGE INTERRUPT HANDLERS

Rx permits the user to handle interrupts in an efficient manner using Procedure ASSEMBLYEVENT. ASSEMBLYEVENT allows a specific assembly language routine to be given control when a particular

interrupt occurs. This routine has two methods by which to relinquish control after the interrupt has been handled. One causes the interrupt process to be resumed; the other causes the interrupt to be propagated to a routine in the Rx environment. This capability is designed to permit assembly language handlers to accumulate data associated with high frequency interrupts until it is appropriate to invoke a higher level handler. Using ASSEMBLYEVENT, interrupts can be handled in approximately 1/5 the amount of time normally required for interrupt handling (see Section 4, 4.3.7 for detailed information on Procedure ASSEMBlyevent).

## 6.5 LINKING THE APPLICATION SYSTEM

The Link Editor enables the user to link together only the modules which are required by the target application.

### 6.5.1 Control File Creation

A link control file must be created to input to the Link Editor, which specifies what application routines to link together and the location of the Rx Run-Time Support library. The link edit control file is created utilizing the source editor of the user's development system.

A sample link control file is included with each different version of Rx. This file specifies the file names of the user's application and CONFIG modules, and the file names of the Rx Kernel and libraries.

These must be specified in the following order:

1) INCLUDE The kernel RXKERN for normal processing, or DBKERN to use the standalone debugger.

2) INCLUDE the optional stream-lined E$PRCS0 process termination routine, if desired. This routine provides faster process termination at the expense of less thorough resource reclamation.

3) INCLUDE the user routines and CONFIG module in any order desired. If a customized version of GHOST$ is to be used, it should also be included here.

4) FIND the standard Rx routines needed in RX1OBJ.

5) FIND the channel routines needed in CHNOBJ. If channels are not being used, this step may be left out for a slightly faster link edit.

6) FIND the clock management routines needed in CLKOBJ. As with the channel routines this step may be left out if the clock is not being used. faster link edit.

7) Finally, FIND the Rx support routines in RX2OBJ. This must be the last step, because the standard routines, channel routines, and clock routines reference symbols defined in this library.

A template for the Link Edit Control File is presented below. Detailed information regarding the format and instructions used can be found in the user manuals for the respective link editors.

```
SYMT
TASK      <system name>                          ; Name of load module
INCLUDE   <device or volume>.RXKERN              ; or DBKERN
INCLUDE   <device or volume>.EPRCS0              ; Optional E$PRCS routine,
;                                                  use only with RXKERN
INCLUDE   <device or volume>.<config>            ; Configuration Module
INCLUDE   <device or volume>.<user app>          ; As many as needed
FIND      <device or volume>.RX1OBJ              ; Standard Routines
FIND      <device or volume>.CHNOBJ              ; Channel Library
FIND      <device or volume>.CLKOBJ              ; Clock Library
FIND      <device or volume>.RX2OBJ              ; Optional Routines
END
```

FIGURE 6-7. SAMPLE LINK EDIT CONTROL FILE.

In this example, <device or volume> stands for whatever pathname information is needed to accesss the specified file. A sample link control file, as well as a description of the files included in the release, is shipped with each copy of Rx.

If the Rx Standalone Debugger will be used, the user's Link Control file should include a reference to the DBKERN module rather than the RXKERN module. as shown in Figure 6-8.

```
SYMT
TASK      <system name>                          ; Name of load module
INCLUDE   <device or volume>.DBKERN              ; or RXKERN
INCLUDE   <device or volume>.<config>            ; Configuration Module
INCLUDE   <device or volume>.<user app>          ; As many as needed
FIND      <device or volume>.RX1OBJ              ; Standard Routines
FIND      <device or volume>.CHNOBJ              ; Channel Library
FIND      <device or volume>.CLKOBJ              ; Clock Library
FIND      <device or volume>.RX2OBJ              ; Optional Routines
END
```

FIGURE 6-8. SAMPLE LINK EDIT CONTROL FILE (USING STANDALONE DEBUGGER).

Often, the target system will contain a combination of RAM and ROM. All data and common segments within the application must be in RAM. The link edit control file must specify to the linkage editor where to place the different types of program segments. This can be accomplished through the use of the PROGRAM and DATA link editor commands. The PROGRAM command is used to specify where all procedure segments (designated via PSEG assembler directives) are to be placed. This command should be used to specify the starting address of ROM. The DATA command is used to specify where all data segments (designated via the DSEG assembler directive) are to be placed. This command should be used to specify the starting address of RAM. All common segments (designated by the CSEG assembler directive)are automatically placed following ay data segments, unless specifically located using the COMMON command. Figure 6-9 illustrates the use of PROGRAM and DATA commands in which ROM starts at address zero and the value >MMMM is the start address of RAM.

```
SYMT
TASK      <system name>                          ; Name of load module
PROGRAM   >0                                     ; Starting address of ROM
DATA      >mmmm                                  ; Starting address of RAM
INCLUDE   <device or volume>.RXKERN              ; or DBKERN
INCLUDE   <device or volume>.<config>            ; Configuration Module
INCLUDE   <device or volume>.<user app>          ; As many as needed
FIND      <device or volume>.RX1OBJ              ; Standard Routines
FIND      <device or volume>.CHNOBJ              ; Channel Library
FIND      <device or volume>.CLKOBJ              ; Clock Library
FIND      <device or volume>.RX2OBJ              ; Optional Routines
END
```

FIGURE 6-9. SAMPLE LINK EDIT CONTROL FILE (SPECIFYING RAM/ROM PARTITIONING).

### 6.5.2 Link Editor Execution

Once the link edit control file has been created using the development system's text edit facilities, the link editor must be executed using the link control file as input. The link editor will include the application modules along with only those modules of Rx necessary for the application system to operate.

There should be no unresolved references listed in the link editor listing output. If unresolved external references are detected by the link editor, the link control file should be rexamined to insure that all the user modules and the correct Rx libraries have been supplied.

## 6.6 TARGET (CONFIGURED) Rx APPLICATION

Upon completion of Link Editor execution, the specified output file will contain the final target application object module. Figure 6-10 reiterates all the steps necessary to produce an Rx load module. The module can now be tested and debugged using either the Rx Standalone Debugger or AMPL. These debugging methods are described in the following two sections.

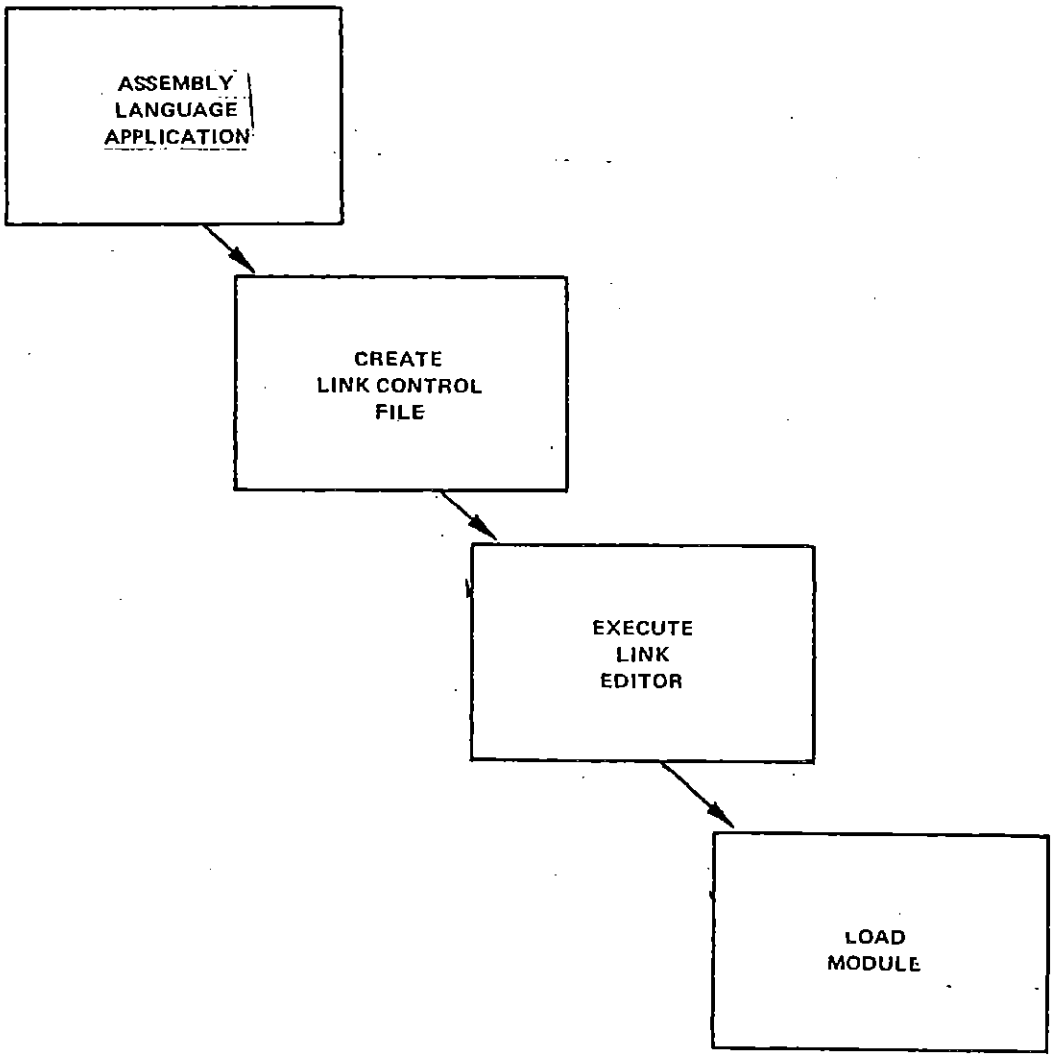



FIGURE 6-10. PRODUCING AN Rx LOAD MODULE.

# SECTION 7

# THE Rx STANDALONE DEBUGGER

## 7.1 GENERAL

This section describes the purpose, capabilities, and use of the standalone debugger supplied with Rx. The standalone debugger is designed to be included in the Rx target system software to aid in testing and locating errors. After the software has been debugged, the standalone debugger software can be removed from the system to reduce code size, or it can remain to assist in field testing or other software maintenance.

The standalone debugger supplies the fundamental tools necessary to debug concurrent and non-concurrent software which may be either ROM or RAM resident. Its commands and messages are very simple in order to conserve memory requirements. The standalone debugger offers the following features:

- o Inspect/change/dump memory contents
- o Inspect/change CRU data
- o Inspect/change hardware registers (WP, PC, ST)
- o Map concurrent processes
- o Display process record
- o Set and unset process entry traps
- o Trace concurrent process scheduling
- o Trap on process creation
- o Set and clear instruction level breakpoint
- o Simulate interrupts
- o Single/multi-step execution
- o Error reporting and recovery
- o Operates with most standard data terminals

The standalone debugger package requires 1780 bytes of procedure space and 56 bytes of data space. It communicates interactively with the operator via a data terminal connected to the target system. The

monitor may be entered at power-up, via interrupt level one, via the load interrupt, or called directly by the application software. A momentary switch connected on the backplane of the target system chassis can be used to supply either the level one or load stimulus. If RAM is located in high memory, the load vector will be used by the debugger and may not be used by the application software.

## 7.2 CONFIGURING A TARGET SYSTEM

Certain modifications must be made to the Rx software system to allow the use of the standalone debugger. These are including DBKERNEL (rather than RXKERNEL) and the debugger in the linked system, and supplying a data terminal at the proper I/O connector.

### 7.2.1 Link Control File

Figure 7-1 shows a typical link editor control file for an Rx system with debugger. INCLUDEd in the control file is the CONFIG module, the debug version of the the kernel (DBKERNEL), and the application object modules.

```
SYMT
TASK      <system name>                        ; Name of load module
INCLUDE   <device or volume>.DBKERN            ; or RXKERN
INCLUDE   <device or volume>.<config>          ; Configuration Module
INCLUDE   <device or volume>.<user app>        ; As many as needed
FIND      <device or volume>.RX1OBJ            ; Standard Routines
FIND      <device or volume>.CHNOBJ            ; Channel Library
FIND      <device or volume>.CLKOBJ            ; Clock Library
FIND      <device or volume>.RX2OBJ            ; Optional Routines
END
```

FIGURE 7-1. LINK CONTROL FILE (WITH DEBUGGER)

### 7.2.2 Data Terminal

A data terminal is needed by the standalone debugger for command entry and message display. Most asynchronous data terminals including teletypes, TI Silent 700 series terminals, and CRT type terminals with an EIA interface may be used. The data transfer rate of the terminal is sensed at power-up and is automatically set. The allowable data transfer rates are listed in Table 7-1. If a 1200 baud terminal is sensed, characters are output at a rate of 30/SEC to allow use with certain TI Silent 700 series data terminals. Character format consists of a start bit, 7 data bits, even parity bit, and two stop bits.

The standalone debugger software is designed to communicate via the main port on either a TM990/101 or TM990/100 microcomputer module. If using a custom configuration, this corresponds to a TMS9902

Asynchronous Controller located at a software CRU base of >80.

TABLE 7-1. ALLOWABLE DATA TRANSFER RATES

| |
|---|
| 19200 baud |
| 9600 baud |
| 4800 baud |
| 2400 baud |
| 1200 baud* |
| 600 baud |
| 300 baud |
| 110 baud |

* Actually transfers characters at 30 char/sec

## 7.3 USING THE DEBUGGER

The Rx Debugger serves as an interface between the user and the user's application system. As such, it allows the user to selectively display information about the state of a target system in execution and/or halt the target system to further examine or modify its environment. This section describes how to use the Rx Debugger to debug a target system which has been configured as in Section 6.2.

### 7.3.1 Getting Started

The Rx Debugger is given control of the target system when the system is first powered-up, and upon RESET. To initialize the system, the user should enter a carriage return to set the baud rate for the terminal being used. The Debugger will then respond with the heading

**RX2.0 STANDALONE DEBUGGER**

and wait
for a command to be input. Once the system has been initialized, the debugger may be reentered by an interrupt level one stimulus or, if high memory is RAM, by a LOAD stimulus. Either of these may be supplied by means of a momentary switch connected between the corresponding terminal and ground on the chassis backplane. The debugger responds with a "?" prompt upon reentry.

Whenever the Rx executive software detects an error condition, the debugger is entered and the message "ERR= nnnn" is issued. "nnnn" is the Rx error code detected. (Appendix B lists the Rx error codes.) If the momentary switch is not debounced, the switch may cause a spurious interrupt for which the system prints the message "ERR=nnnn". The

error code 'nnnn' has no meaning in this instance, and the debugger returns the '?' prompt. The 'GO' command given at this point would cause the system to idle. Therefore, a level 0 interrupt should be supplied in order to reinitialize the system. (This can be simulated using the debugger command 'SIMI 0'.) When the system is reinitialized, the 'GO' command returns to the debugger, and the heading and prompt are again shown.

### 7.3.2 Commands

The commands implemented by the Rx Debugger fall into three categories: an informatory message command (TP), system inspection and modification commands (IM, IC, IR, PD, SIMI and DAP), and control of execution commands (SC, ABP, DBP, SB, CB, GO and IS). The informatory message command (TP) displays information about the system in execution but does not halt the system or otherwise affect its execution. The system inspection and modification commands display information about a target system which has been halted and allow the user to modify its environment. The control of execution commands provide means for the user to selectively halt a target system and return to the Debugger. Any output provided by the debugger can be prematurely terminated by pressing the escape key of the data terminal. This causes the "?" prompt to be issued for a new command. The escape key can also be used to terminate entry of an improperly typed command or parameter.

The Rx Debugger implements the following commands:

| | |
|---|---|
| SC <flag> | Set Process Creation Trap |
| TP <flag> | Trace Process Scheduling |
| IM <addr,addr> | Inspect/Modify/Dump Memory |
| IC <base,width> | Inspect/Modify CRU |
| IR | Inspect/Modify Hardware Registers (WP, PC, ST) |
| PD <addr> | Process Record Dump |
| DAP | Display All Processes |
| ABP <addr> | Assign Process Breakpoint |
| DBP <addr> | Delete Process Breakpoint |
| SB <addr> | Set Breakpoint |
| CB | Clear Breakpoint |
| SIMI <level> | Simulate Interrupt |
| GO | Return to User Context |
| IS <count> | Instruction Step |

These commands are described in detail in the following sections.

#### 7.3.2.1 Process Creation Trap (SC)

SYNTAX: SC <flag>

This command is used to set or reset a flag which, when set, will cause the message "CREATE TRAP PR=nnnn" to be displayed and the target system to be halted whenever a new process is created. "nnnn"

is the address of the process record of the newly created process record.

The command has an optional argument which specifies how the flag is to be set: a zero enables the process creation trap and any non-zero value disables it. The default value (if no argument is specified) is to enable the trap.

#### 7.3.2.2 Trace Process Scheduling (TP)

SYNTAX: TP <flag>

This command is used to set or reset a flag which, when set, will cause the message "PRCS TRC PR=nnnn" to be displayed whenever an active process is suspended and a new process becomes active. "nnnn" is the address of the process record of the newly activated process. This command does not cause the target system to be halted when the message is displayed.

The command has an optional argument which specifies whether or not to trace process scheduling: a zero enables the process trace and any non-zero value disables it. The default value (if no argument is specified) is to enable the trace.

#### 7.3.2.3 Inspect/Modify/Dump Memory (IM)

SYNTAX: IM <addr,addr>

This command is used in two ways: to display a range of memory locations or to interactively display and/or modify memory locations. If two arguments are input and the second is greater than the first, a memory dump is performed from the first address to the second. Otherwise, the contents of the first address are displayed and the monitor waits for instructions on what to do next. At this point the user may assign a new hexadecimal value to the memory location by entering the new value terminated by one of the following:

1) A carriage return to return to the Debugger "?" prompt mode, or

2) A minus sign to cause the Debugger to back up one word and display that value, or

3) Anything else to display the next word of memory.

Example:

```
?IM 1A,2C <cr>    (Dump memory from >001A to >002C)
001A=0123 4567 89AB CDEF  FEDC BA98 7654 3210
002A=0466 2117
?IM 1C <cr>       (Inspect/Modify memory starting at >001C)
```

```
001C=4567  10FF <sp>
001E=89AB  -
001C=10FF  <sp>
001E=89AB  <sp>
0020=CDEF  <cr>
```

If no argument is specified the command will go into interactive mode at location zero.

### 7.3.2.4 Inspect/Modify CRU (IC)

SYNTAX: IC <base,width>

This command operates in the same manner as the IM command, with the difference that the first argument is considered the CRU base and the second argument is used as the transfer width. A width of zero is considered a width of sixteen. To change either of these arguments after the command has been entered it is necessary to enter a carriage return and re-enter the command with the new arguments.

### 7.3.2.5 Inspect/Modify Registers (IR)

SYNTAX: IR

This command displays the hardware registers WP, PC, and ST for the current user routine. The registers are displayed one at a time and the user is given the option of changing the register value by entering a new value terminated by one of the following:

2) Inspecting the next register by entering a space,

3) Inspecting the same register again by entering a minus sign, or

4) Returning to the Debugger "?" prompt mode by entering a carriage return.

Example:

```
?IR
WP=908C  906C <->
WP=906C  <sp>
PC=0480  121A <sp>
ST=000F  0000 <cr>
```

### 7.3.2.6 Process Record Dump (PD)

SYNTAX: PD <addr>

This command displays an unformatted process record beginning at location <addr>. If no address is specified, location zero is assumed. The meaning of each field in the process record is explained in Appendix A.

Example:

```
?PD FED6
FED6=FE36  0FBC  FF58  0000    0000  0000  0000  0000
FEE6=0000  0000  0000  0000    FF58  FBDA  FED6  FED6
FEF6=02E0  FEE2  0380  FEB6    0936  0000  0000  0000
FF06=0001  FC00  FF78  0000    0000  0000  0000  0000
FF16=EBAC  EBAC  FFAA  0000    FFFF  0001
```

### 7.3.2.7 Displa All Processes (DAP)

SYNTAX: DAP

This command displays a listing of all the processes currently in the system, starting with the active process. The listing displays the process record pointer, process identification field of the process record and current workspace pointer (WP) for each process. The low-order byte of the process identification field indicates the process number of the process in question; the high-order byte indicates its creator's process number. Example:

```
?DAP
PR=EBAC  ID=0102  WP=EB0C
PR=FB70  ID=0203  WP=FB7C
PR=FFAA  ID=0000  WP=FFB6
PR=B4C2  ID=0000  WP=B462
PR=FED6  ID=0001  WP=FE76
```

### 7.3.2.8 Assign Process Breakpoint (ABP)

SYNTAX: ABP <addr>

This command sets a trap on a process whose process record is at location <addr>, such that the message "PRCS TRAP PR=nnnn" is displayed and the target system returns to the Debugger prompt mode whenever the process indicated becomes active. "nnnn" is the process record address for the newly-activated process. This command has no default argument.

After a trap has been set the Assign Process Breakpoint command prints a list of the traps that are currently set. A maximum of four traps may be set at any one time.

### 7.3.2.9 Delete Process Breakpoint (DBP)

SYNTAX: DBP <addr>

This command removes a trap set by the Assign Process Breakpoint command for a process whose process record is at location <addr>. After the trap has been removed the command prints a list of the traps that remain set. If no value is entered, all process traps are removed.

### 7.3.2.10 Set Breakpoint (SB)

SYNTAX: SB <addr>

This command sets a breakpoint at a given memory location, such that the target system will halt, print a "BRK PT PC=nnnn" message and return to the Debugger prompt mode when the breakpoint is encountered. Only one breakpoint at a time may be resident in the system and is removed when encountered. Entering a breakpoint when one is already set causes the clearing of the old breakpoint. If no value is entered, the current breakpoint is displayed.

Note that breakpoints may not be set in ROM.

### 7.3.2.11 Clear Breakpoint (CB)

SYNTAX: CB

This command clears the current breakpoint if one is set.

### 7.3.2.12 Simulate Interrupt (SIMI)

SYNTAX: SIMI <level>

This command is used to simulate an interrupt of <level> to the user's system. The parameter must be a valid interrupt level in the range 0 <= level <= 15. If the interrupt is simulated, the new PC and WP are displayed.

### 7.3.2.13 Return To User Context (GO)

SYNTAX: GO

This command is used to start or resume execution of the user's target system. The first time GO is entered, execution begins at the beginning of the user's system; thereafter execution resumes wherever it was last suspended (as by a breakpoint or a trap).

### 7.3.2.14 Instruction Step (IS)

SYNTAX: IS <count>

This command single-steps a target system for a specified number of instructions and prints a "WP=nnnn PC=mmmm" message for each instruction executed. <count> specifies the number of instructions to be single-stepped; the default value (if none is specified) is one.

Note that Instruction Step will not work if high memory is ROM. FFFC-FFFF must be RAM, and accessible only to the debugger.

# SECTION 8

# DEBUGGING THE TARGET APPLICATION WITH AMPL

## 8.1 GENERAL

The Advanced Microprocessor Prototyping Laboratory (AMPL) provides hardware and software tools enabling the user to perform real-time debugging with in-circuit emulation capabilities (execution in the target environment monitored by the host development system). Hardware requirements for use of AMPL include an FS990/4 or FS990/10 Minicomputer System with floppy diskette storage and a Model 911 (or 913) Video Display Terminal (VDT) to provide operator interface. A Model 810 printer can be optionally added to provide hard copy production capability.

The FS990 systems include the AMPLUS software system. AMPLUS is an operating system containing, an assembler, a text editor, a link editor, and a compliment of file manager utilities.

Additional information on the AMPL system and the FS990 system (and peripherals) are available from the local Texas Instruments sales office or authorized distributor.

The purpose of the information that follows is to describe a set of predefined AMPL procedures providing a convenient user interface with target application systems supported by the Rx. Examples of system output will be provided for those routines that require it. Following the description of these procedures, a debug session (listing) is presented to demonstrate the procedures' actual use.

## 8.2 AMPL PROCEDURES

A set of AMPL II procedures are supplied the user. For those using AMPL I systems, instructions are provided to convert AMPL II procedures to AMPL I by removing comments from existing statements. These instructions are found in the source code for the AMPL II procedures.

There are three files containing AMPL procs, grouped according to these functions:

1) Basic commands INIT, GO, HALT, IS, SB, CB, SIMI, HELP

2) Process commands SC, TP, ABP, DBP, DAP, PD

3) Miscellaneous commands HP, RP, SM, MM, SP, SF, SH, SEMA

These are broken into three files to accomodate systems with limited memory, and normally all three sets of commands are used. The first set of commands must be read in, and either of the other files may also be used. If some other combination of procedures is desired, the user may edit the file containing the desired proc or procs and extract them. They may then be read in usinng the AMPL COPY command.

## 8.3 INSTRUCTION SIMULATION PROCEDURES

The following procedures simulate specific actions in the target system.

### 8.3.1 Procedure INIT

PURPOSE: Initialize the debugger with the origins of Rx Routines S$$PRC, INT$PC, RELINQ and SWITCH. These addresses are needed to set hooks in Rx code to implement the 'SC', 'TP' and 'ABP' commands. Init will also initialize the ready queue and the suspend HP queue to NIL.

CALLING SEQUENCE: INIT

The user will be prompted for the origins of the particular system routines via the form 'INIT'.

ARG 1 - The origin of routine S$$PRC
ARG 2 - The origin of routine INIT$PC
ARG 3 - The origin of routine RELINQ
ARG 4 - The origin of routine SWITCH

EXECUTION: The message 'INITIALIZATION COMPLETE' is printed after all initialization has been performed.

CONDITIONS: The user should enter zero for any routine not in his system.

AMPL I
EXAMPLE: ? INIT

INITIALIZATION COMPLETE

EXAMPLE: ?INIT

INITIALIZATION COMPLETE

### 8.3.2 Procedure HELP

PURPOSE: Display information about the commands available with the Rx target debugger.

CALLING SEQUENCE: HELP

CONDITIONS: None.

EXAMPLE: ? HELP

```
                    ********************************
                    * RX TARGET DEBUGGER COMMANDS *
                    ********************************

INIT(<load address>)              - INITIALIZE DEBUGGER
GO                                - START EMULATOR EXECUTION
HALT                              - HALT EMULATOR EXECUTION
IS                                - SINGLE-STEP INSTRUCTION(S)
SB(<ADDR<,ADDR...>>)              - SET SOFTWARE BREAKPOINT(S)
CB(<ADDR<,ADDR...>>)              - CLEAR SOFTWARE BREAKPOINT(S)
SC(<FLAG>)                        - HALT ON PROCESS CREATIONS
TP(<FLAG>)                        - TRACE PROCESS ACTIVATIONS
ABP(<PROCESS<,PROCESS..>>)        - ASSIGN PROCESS TRAP(S)
DBP(<PROCESS<,PROCESS..>>)        - DELETE PROCESS TRAP(S)
HP(<PROCESS RECD>)                - HOLD PROCESS
RP(PROCESS RECD)                  - RELEASE PROCESS
SIMI(LEVEL)                       - SIMULATE INTERRUPT
SM(ADDR<,DISP<,LENGTH>>)          - SHOW MEMORY
MM(ADDR,OLD,NEW)                  - MODIFY MEMORY
DAP                               - DISPLAY ALL PROCESSES
PD(PROCESS RECD)                  - DISPLAY PROCESS RECORD
SP(PROCESS RECD)                  - SHOW PROCESS
SF(ROUTINE ADDR)                  - SHOW STACK FRAME
SH(HEAP PACKET)                   - SHOW HEAP PACKET
SEMA(SEMAPHORE)                   - DISPLAY SEMAPHORE
```

### 8.3.3 Procedure SIMI

PURPOSE: Simulate an interrupt at a particular interrupt level in the target system.

CALLING SEQUENCE: SIMI(level)

where <level> is the level of the interrupt request.

EXECUTION: If the interrupt is allowed to occur (determined by checking ST), a BLWP is perfomed through the vector stored at the appropriate interrupt level. Also, the priority of the CPU is set to the level of the interrupt minus one. SIMI returns 'INTERRUPT IS NOT ALLOWED' if the interrupt was not allowed, and 'INTERRUPT SUCCESSFUL' if it was.

CONDITIONS: SIMI may be called whenever the emulator is halted. The interrupt is serviced only if the CPU's priority is numerically less than or equal to the level on which the interrupt is raised.

EXAMPLE: ? SIMI(0)
INTERRUPT SUCCESSFUL

## 8.4 BREAKPOINT PROCEDURES

The following procedures implement breakpoints in the target software.

### 8.4.1 Procedure SB

PURPOSE: Set a breakpoint or inspect all current breakpoints.

CALLING SEQUENCE:

Set Breakpoints: SB(addr,addr...) where addr is the address at which the system sets each breakpoint. ('addr' must be a RAM location.)

Check Breakpoints: SB

EXECUTION: The message 'BREAKPOINT SET AT addr' is printed for each breakpoint set. Up to 16 breakpoints may be set at any one time.

CONDITIONS: A breakpoint will not be set when the breakpoint table is full (the message noted above in "Execution" will not print).

### 8.4.2 Procedure CB

PURPOSE: Clear the given (or all) breakpoints.

CALLING SEQUENCE:

Clear The Breakpoints Given: CB(addr,addr...)

where (addr) is the address of the breakpoint to be cleared.

Clear All Breakpoints: CB

EXECUTION: The message 'BREAKPOINT CLEARED AT addr' is printed for each breakpoint cleared.

CONDITIONS: If user supplied address in call contains no breakpoint, the message noted above will not print.

### 8.4.3 Procedure GO

PURPOSE: Start Emulator Execution

CALLING SEQUENCE: GO

EXECUTION: Target system execution starts from a software breakpoint and continues up to the next software breakpoint encountered. The instruction at this encountered breakpoint is not executed. Software breakpoints are set by the SB (SET SOFTWARE BREAKPOINT) command and cleared by the CB (CLEAR BREAKPOINT) command.

CONDITIONS: The GO procedure uses an internal procedure called 'EBRUN' which uses both the emulator and trace modules. Users who do not have a trace module may use an alternate version of 'EBRUN' by editing the procedure file, commenting out the current version, and removing comment delimiters from the alternate version.

EXAMPLE:
```
? GO
PROCESS CREATED: PR = >9EDA

  ? GO
PROCESS ACTIVE:  PR = >9EDA
PROCESS CREATED: PR = >8BB0
```

### 8.4.4 Procedure SC

PURPOSE: Enable process creation monitoring by the 'GO' procedure.

CALLING SEQUENCE: SC(flag)

where (flag) ON = "MONITOR PROCESS CREATIONS", and flag OFF = "DO NOT MONITOR PROCESS CREATIONS". (Note that the default = ON.)

EXECUTION: When monitoring is in effect and 'GO' is entered, the emulator will be halted and message 'PROCESS CREATED : PR= >nnnn' will be printed every time a new process is created.

CONDITIONS: None.

### 8.4.5 Procedure TP

PURPOSE: Enables process activation monitoring by the 'GO'.

CALLING SEQUENCE: TP(flag)

where (flag) ON = "MONITOR PROCESS ACTIVATIONS", and flag OFF = "DO NOT MONITOR PROCESS ACTIVATIONS". (Note that the default = ON.)

EXECUTION: When monitoring is in effect and 'GO' is entered, the emulator will be halted and the message 'PROCESS ACTIVE: PR= >nnnn' will be printed every time a new process is created.

CONDITIONS: None.

## 8.5 REALTIME EXECUTIVE PROCEDURES

The following AMPL procedures use constructs of the Microprocessor Executive.

### 8.5.1 Procedure HP

PURPOSE: Hold an Rx process.

CALLING SEQUENCE: HP(addr)

where (addr) is the address of the process record of the process to be held. (Entering 'HP' by itself will print a list of all the currently held processes.)

EXECUTION: The process is removed from the ready queue or semaphore queue in a suspend queue that exists outside of the Rx system.

CONDITIONS: The process may not execute until it is released via the RP procedure. A process may not be held when the processor is halted at a process creation breakpoint; ABP or SB may be used to halt the processor at a later point to perform the HP operation.

### 8.5.2 Procedure RP

PURPOSE: Release a process held by HP.

CALLING SEQUENCE: RP(process)

where (process) is the process to be released.

EXECUTION: A process that has been held by HP is released from the suspend queue and either scheduled for execution or resuspended upon a semaphore that it was suspended upon when held.

CONDITIONS: A process may not be released when the processor is halted at a process creation breakpoint; ABP or SB may be used to halt the processor at a later point to perform the RP operation.

### 8.5.3 Procedure ABP

PURPOSE: Assign process traps.

CALLING SEQUENCE: ABP(addr)

where (addr) is the address of the process record of the process to be "trapped".

EXECUTION: A breakpoint is set on a given process so that the system is halted at the next instance in which the process becomes active. The message 'PROCESS TRAPPED: PR= >nnnn' will be printed when this occurs.

CONDITIONS: If no argument is supplied, a list of all process traps currently set is printed. A maximum of four process traps may be set at any one time.

EXAMPLE: ? ABP
```
        LIST OF ALL TRAPS CURRENTLY SET
          PROCESS >9EDA
          PROCESS >8BB0
```

### 8.5.4 Procedure DBP

PURPOSE: Delete process traps.

CALLING SEQUENCE: DBP(addr)

where (addr) is the address of a process record of a process for which a trap is to be removed.

EXECUTION:

CONDITIONS: If no argument is supplied, all currently set traps will be removed. Up to four traps may be removed at one time.

### 8.5.5 Procedure PD

PURPOSE: Print a formatted dump of an Rx process record.

CALLING SEQUENCE: PD(addr)

where (addr) is the address of the process record to be displayed.

EXECUTION: The procedure prints out a formatted listing of an Rx process record given its address.

CONDITIONS: None.

EXAMPLE:

```
    ?PD(05960)
  DUMP OF PROCESS RECORD AT >5060

  NEXTWS = >590C   LEX$8   = >0000   PC      =  >0C7A   FIRST   =  >0000
  CALL   = >0268   LEX$9   = >0000   ST      =  >D000   NS/OBJ  =  >0000(
  UNUSED = >0000   GLOBAL  = >595C   OUTPUT  =  >0000   OBJECT  =  >0000
  LEX$1  = >595C   STKMIN  = >517C   INPUT   =  >0000   PRLIST  =  >5CFC
  LEX$2  = >0000   STKMAX  = >592C   PRITY   =  >001E   LSTRTD  =  >0000
  LEX$3  = >0000   UNUSED  = >0000   CLKSRV  =  >0000   QNEXT   =  >0000
  LEX$4  = >0000   LWPI    = >02E0   MPXPTR  =  >5FF0   QUEUE   =  >0000
  LEX$5  = >0000   CNTEXT  = >596C   CL/RS   =  >0000   INTLVL  =  >0000
  LEX$6  = >0000   RTWP    = >0380   XHNDLR  =  >0000   CR/MY   =  >0304
  LEX$7  = >0000   WP      = >5FCE
```

### 8.5.6 Procedure SEMA

PURPOSE: Display information about a semaphore.

CALLING SEQUENCE: SEMA(addr)

where (addr)is the address of the semaphore.

EXECUTION: This procedure prints out a listing of a semaphore value given a semaphore address.

CONDITIONS: None.

### 8.5.7 Procedure SM

PURPOSE: Show memory.

CALLING SEQUENCE: SM(addr,<length>,<displacement>)

where (addr) is the address to start, <length> is the length to display (default=2), and <displacement> is the displacement to display (default=0)

EXECUTION: Shows the contents of one or more absolute memory locations.

CONDITIONS: None.

EXAMPLE: ? SM(>8BB0,>4C)

```
        >8BB0 (>0000)  >8B10  >0676  >9F5C  >9BD2    (..  .V  ./  .R  )
        >8BB8 (>0008)  >0000  >0000  >0000  >0000    (..  ..  ..  ..  )
         SAME AS LAST LINE
        >8BC8 (>0018)  >9BD2  >878C  >8BB0  >8BB0    (..  .<  .0  .0  )
        >8BD0 (>0020)  >02E0  >8BBC  >0380  >8B90    (..  ..  ..  ..  )
        > BD8 (>0028)  >0284  >000F  >0000  >0000    (..  ..  ..  ..  )
        >8BE0 (>0030)  >001E  >FC0F  >9F7C  >0000    (..  :.  .:  ..  )
        >8BE8 (>0038)  >0000  >0000  >0000  >0000    (..  ..  ..  ..  )
        >8BF0 (>0040)  >9B74  >9B74  >9FAE  >0000    (..  .t  ..  ..  )
        >8BF8 (>0048)  >0102  >0102                  (..  ..  .       )
```

### 8.5.8 Procedure MM

PURPOSE: Modify memory.

CALLING SEQUENCE: MM(addr,old,new)

where (addr) is the address to modify, (old) is the verification value, and (new) is the new value.

EXECUTION: Modifies the contents of any single (word) location in memory. Note that caution should be exercised when using this command.

CONDITIONS: None.

### 8.5.9 Single Step Instruction(s) (IS)

PURPOSE: Instruct the emulator to execute a specified number instruction and then halt.

CALLING SEQUENCE: IS(count)

where (count) specifies the number of instructions to be individually stepped through. When no argument is supplied, default is one.

EXECUTION: One or more instructions are executed in single-step mode, printing information about each instruction before it is executed. 'IS' prints the current instruction, the current WP, PC and ST, and the source and destination operands (if any), executes the current instruction and then displays the above information about the next INSTRUCTION TO BE EXECUTED.

CONDITIONS: The emulator comparison logic may not be utilized during use of the IS procedure. However, IS can be used with with the emulator trace logic and the trace module when only the instruction need be recorded.

### 8.5.10 Procedure SP

PURPOSE: Show process.

CALLING SEQUENCE: SP(process)

where (process) is the address of the process record.

EXECUTION: Displays information about stack and heap usage, process ID, exceptions outstanding, etc.

CONDITIONS: None.

EXAMPLE:
```
? SP(>8BB0)
   ? SP(>8BB0)
 SHOW PROCESS >8BB0

 STACK BASE = >878C  STACK LIMIT      = >8BB0  STACK BOUNDARY   = >8BB0
 STACK SIZE = >083F  STACK USED (MAX) = >087F  STACK USED (CUR) = >7FAF

 PRIORITY =     30
 NO OUTSTANDING EXCEPTIONS
 NEXT PROCESS IN LIST = >9B74    NEXT PROCESS IN QUEUE = >9FAE
 CREATOR'S ID = >01              MY ID = >02
```

### 8.5.11 Procedure SF

PURPOSE: Show stack frame.

CALLING SEQUENCE: SF(<addr>,<displacement>,<length>)

where (addr) is the address of the routine or process code (default=current routine), <displacement> is the displacement into the frame (default=0), <length> is the length to display (default=rest of frame).

EXECUTION: Displays the specified memory locations. The message 'STACK FRAME NOT FOUND' is printed if the requested routine or process is not in the system at the time SF is entered.

CONDITIONS: None.

### 8.5.12 Procedure SH

PURPOSE: Show heap packet.

CALLING SEQUENCE: SH(addr,<displacement>,<length>)

where <addr> is the address of the heap packet, <displacement> is the displacement into the packet (default=0), and <length> is the length to display (default=rest of packet).

CONDITIONS: None.

### 8.5.13 Procedure HALT

PURPOSE: Halt emulator (and trace module) execution.

CALLING SEQUENCE: HALT

EXECUTION: HALT will reset the default process to the currently active process.

CONDITIONS: None.

## 8.6 AMPL WALK-THROUGH DEBUGGING SESSION

The listing that follows demonstrates a debugging session with AMPL utilizing 'RXDEMO' as the program being debugged. Note that Appendix E supplies the user an assembler listing and link map of 'RXDEMO' for use with this session.

### 8.6.1 Getting Ready

Before proceeding with the debug session, a set of "compiled" AMPL II procedures should be built. This will provide for fast access to the AMPL emulator in future debug sessions. The same set of compiled procedures can be used on all Rx systems, therefore time spent will pay dividends in the future. To build this set of procedures, load AMPL and proceed as shown below:

```
? CLR(50)

? COPY('<pathname of RXAMPLA')        .. Load and "compile" the

? COPY('<pathname of RXAMPLB')        .. procedures which are

? COPY('<pathname of RXAMPLC')        .. provided with Rx

? COPY('<pathname of user proc>')     .. Repeat with any user procs

? SAVE('<pathname of saved procs>')   .. "SAVE" the compiled procs

? EXIT                                ..  Finished with this phase
```

CAUTION

DO NOT remove this disk during the debug session as the second generation of AMPL uses overlays on the disk which contains the debug session. At the beginning of a session, always reload AMPL to ensure the initial state of this disk is recorded by AMPL. For every other debug session, restore the procedures to the system.

Note that when attempting 'COPY', the user must first perform a CLR command with an operand of 50 to ensure that a symbol table of the appropriate size is created.

### 8.6.2 The Debug Session

The debug session begins as any other, with loading of the AMPL program (NOTE:comments are bracketed by asterisks, and are not to be typed in):

```
    *** OBTAIN A HARD COPY (if desired) ***

? LIST('LP')

    *** RESTORE THE SAVED PROCEDURES ***

? RSTR('<path name of saved Rx procs>')

    *** INITIALIZE THE BUFFER AND TRACE MODULES ***

? EINT('EM01',0,'TM01') .. 0 ==> target clock
```

```
                        .. 1 ==> host clock

        *** LOAD THE DEMONSTRATION PROGRAM ***

  ? LOAD('<path name of demo load module>',0)

        *** GET A LISTING OF THE SYMBOLS DEFINED ***

  ? MSYM

LOAD MODULE SYMBOLS:
        LOAD POINT: >0000, LENGTH: >150E  BYTES.

RXDEMO  PGMID >0000      RXKERN  IDT   >0000      CONFIG  IDT   >08D8
$BOOTP  LCL   >08E4      $DEFAU  LCL   >08DE      $FILL   LCL   >08E0
$IODIR  LCL   >08F0      $LREX   LCL   >08DA      $RAMTB  LCL   >08EE
$RESTA  LCL   >08D8      $STKSZ  LCL   >08E2      $SYSCR  LCL   >08DC
IODIR   LCL   >08F8      RAMTB   LCL   >08F2      RXDEMO  IDT   >08FA
EPILOG  LCL   >092C      FRMSIZ  LCL   >0902      HPSIZE  LCL   >090A
LEXLVL  LCL   >0904      PRIORI  LCL   >0906      PROLOG  LCL   >090C
RXDEMO  LCL   >08FA      STKSIZ  LCL   >0908      SYSTM$  LCL   >08FA
PRODUC  IDT   >0938      EPILOG  LCL   >0AF6      FRMSIZ  LCL   >0940
HPSIZE  LCL   >0948      LEXLVL  LCL   >0942      PRIORI  LCL   >0944
PRODUC  LCL   >0938      PROLOG  LCL   >09DC      SENDLP  LCL   >0A42
STKSIZ  LCL   >0946      CONSUM  IDT   >0B02      CONSUM  LCL   >0B02
EPILOG  LCL   >0BE6      FRMSIZ  LCL   >0B0A      HPSIZE  LCL   >0B12
LEXLVL  LCL   >0B0C      PRIORI  LCL   >0B0E      PROLOG  LCL   >0B48
STKSIZ  LCL   >0B10      WAITLP  LCL   >0B70      WAITIO  IDT   >0BF2
BAUDLP  LCL   >0C40      BAUDTB  LCL   >0C50      CHARLP  LCL   >0D24
CONTRL  LCL   >0C4E      CR      LCL   >0CE7      FOUND   LCL   >0CC2
LOADLP  LCL   >0C36      MATCH   LCL   >0C48      MSGENT  LCL   >0D20
MSGEXI  LCL   >0D3C      NULLCH  LCL   >0CE6      PAD     LCL   >0CD2
PADCR   LCL   >0CD0      SENDOF  LCL   >0CE2      TABLE   LCL   >0CE8
TBLLP   LCL   >0CB8      TESTSP  LCL   >0C10      TI$CIN  LCL   >0C70
TI$COT  LCL   >0C92      TI$MSG  LCL   >0D18      TI$SET  LCL   >0BF2
TIMELP  LCL   >0C26      WAITLP  LCL   >0C84      WTLP$1  LCL   >0CAE
WTLP$2  LCL   >0CD6      E$PRCS  IDT   >0D40      GHOST$  IDT   >0EA4
C$ACKN  IDT   >0EDC      C$ALLO  IDT   >0F0E      C$INIT  IDT   >0FAA
C$RECE  IDT   >1094      C$SEND  IDT   >1102      C$TERM  IDT   >1182
C$WAIT  IDT   >1212      C$$MSG  IDT   >1244      C$$HEA  IDT   >125E
CLK$TE  IDT   >1278      CKSEMA  IDT   >1314      D$INIT  IDT   >132C
D$TERM  IDT   >1330      EXIT$1  IDT   >1334      F$TERM  IDT   >1354
HP$SYS  IDT   >1358      INITSE  IDT   >1364      MSG$IN  IDT   >13AC
MY$MPX  IDT   >13B8      RT$ENT  IDT   >13C2      RT$EXI  IDT   >13D0
SETMAS  IDT   >13D8      TERMSE  IDT   >13F4      CK$SEM  IDT   >142E
EXCEPT  IDT   >1442      EXIT$O  IDT   >1460      HP$FRE  IDT   >146E
HP$NEW  IDT   >14B2
   100  ENTRIES IN TABLE.
```

```
        *** GET A LIST OF COMMANDS AVAILABLE ***

  ? HELP

            **********************************
            * RX TARGET DEBUGGER COMMANDS *
            **********************************

INIT<(LOAD ADDR)>           - INITIALIZE DEBUGGER
GO                          - START EMULATOR EXECUTION
HALT                        - HALT EMULATOR EXECUTION
IS<(COUNT)>                 - SINGLE-STEP INSTRUCTION(S)
SB(<ADDR<,ADDR...>>)        - SET SOFTWARE BREAKPOINT(S)
CB(<ADDR<,ADDR...>>)        - CLEAR SOFTWARE BREAKPOINT(S)
SC(<FLAG>)                  - HALT ON PROCESS CREATIONS
TP(<FLAG>)                  - TRACE PROCESS ACTIVATIONS
ABP(<PROCESS<,PROCESS..>>)  - ASSIGN PROCESS TRAP(S)
DBP(<PROCESS<,PROCESS..>>)  - DELETE PROCESS TRAP(S)
HP(<PROCESS RECD>)          - HOLD PROCESS
RP(PROCESS RECD)            - RELEASE PROCESS
SIMI(LEVEL)                 - SIMULATE INTERRUPT
SM(ADDR<,DISP<,LENGTH>>)    - SHOW MEMORY
MM(ADDR,OLD,NEW)            - MODIFY MEMORY
DAP                         - DISPLAY ALL PROCESSES
PD(PROCESS RECD)            - DISPLAY PROCESS RECORD
SP(PROCESS RECD)            - SHOW PROCESS
SF(ROUTINE ADDR)            - SHOW STACK FRAME
SH(HEAP PACKET)             - SHOW HEAP PACKET
SEMA(SEMAPHORE)             - DISPLAY SEMAPHORE

        *** INITIALIZE THE RX PROCS ***

  ? INIT
INITIALIZATION COMPLETE

        *** SIMULATE A LEVEL ZERO INTERRUPT ***
        ***        TO RESET THE SYSTEM       ***

  ? SIMI(0)
INTERRUPT SUCCESSFUL

        *** TRACE PROCESS ACTIVIATION ***

  ? TP

        *** HALT ON PROCESS CREATION ***

  ? SC
```

```
        *** START EMULATION ***

  ? GO
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS CREATED: PR = >9CE0

        *** GHOST$ PROCESS STARTED ***

  ? GO
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS CREATED: PR = >9C5E

        *** SYSTEM PROCESS STARTED ***

  ? GO
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9F72
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS CREATED: PR = >99FC

        *** PRODUCER PROCESS STARTED ***

  ? GO
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS CREATED: PR = >9794

        *** CONSUMER PROCESS STARTED ***

  ? GO
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >9C5E
PROCESS ACTIVE:  PR = >99FC

        *** TYPE A CARRIAGE RETURN ON THE TARGET SYSTEM     ***
        *** TO AUTO-BAUD THE TERMINAL, THEN TYPE ANY LETTER ***
        ***       TO BE INPUT TO THE PRODUCER PROCESS.      ***

PROCESS ACTIVE:  PR = >99FC
PROCESS ACTIVE:  PR = >99FC
PROCESS ACTIVE:  PR = >99FC
PROCESS ACTIVE:  PR = >99FC
PROCESS ACTIVE:  PR = >99FC
```

```
PROCESS ACTIVE:  PR = >9794
PROCESS ACTIVE:  PR = >9794
PROCESS ACTIVE:  PR = >9794
PROCESS ACTIVE:  PR = >9794
PROCESS ACTIVE:  PR = >99FC

        *** HIT <CMD> ON THE HOST SYSTEM ***
        ***      TO ENTER COMMAND MODE     ***

  ? HALT

        *** TURN OFF PROCESS TRACING ***

  ? TP(OFF)

  ? SC(OFF)

        *** DISPLAY ALL PROCESSES IN SYSTEM ***

  ? DAP
STATUS SUMMARY OF ALL EXISTING PROCESSES

                              LEXICAL      CURRENT
  PROCESS NUMBER  ADDRESS     LEVEL        STATUS          PRIORITY
       4          >99FC        1          ACTIVE             256
       2          >9CE0        1          READY            32767
       5          >9794        1          WAITING            256

    *** >99FC IS THE PROCESS RECORD ADDRESS FOR THE PRODUCER      ***
    *** >99E0 IS THE PROCESS RECORD ADDRESS FOR THE IDLE PROCESS ***
    *** >9974 IS THE PROCESS RECORD ADDRESS FOR THE CONSUMER      ***

        *** DISPLAY THE REGISTERS FOR THE CURRENT ROUTINE ***

  ? DR
R0  = >0000  R8  = >0C70  PC  = >0C86  / >16FE  JNE  WAITIO.WAITLP
R1  = >9560  R9  = >97E6  WP  = >99A2
R2  = >0000  R10 = >97E2  ST  = >900F
R3  = >5400  R11 = >0000
R4  = >0000  R12 = >0080
R5  = >0000  R13 = >99C2
R6  = >0000  R14 = >0A52
R7  = >99FC  R15 = >800F

        *** SET A BREAKPOINT IN PRODUC AFTER CALL TO TI$CIN ***

  ? SB(PRODUC.+>11A)

BREAKPOINT SET AT PRODUC.+>011A  / >CEA9  MOV  @RXDEMO.+>0002(R9),*R10+

        *** RESTART EMULATION ***

  ? GO
```

```
        *** NOW TYPE 'K' ON THE TARGET TERMINAL ***

BREAKPOINT ENCOUNTERED AT ADDRESS: PRODUC.+>011A  / >CEA9  MOV  @RXDEMO.+>00(
9),*R10+

        *** DISPLAY THE LOCAL FRAME OF THIS PROCEDURE ***
        ***  THIS WILL NOT WORK FOR ROUTINES WITH AN  ***
        ***             OPTIMIZED LINKAGE             ***

  ? SF
STACK FRAME FOR ROUTINE PRODUC.
>99F4 (>0000)  >0068  >956E  >9560          (.h  .n  .`      )

WORKSPACE REGISTERS ARE:
R0  = >0000  R8  = >0938  PC  = >0A52  / >CEA9  MOV  @RXDEMO.+>0002(R9),*R10+
R1  = >9560  R9  = >99F4  WP  = >99C2
R2  = >9560  R10 = >97E2  ST  = >800F
R3  = >0000  R11 = >0A52
R4  = >0000  R12 = >0000
R5  = >0000  R13 = >9C2A
R6  = >0000  R14 = >0928
R7  = >99FC  R15 = >000F

        *** DISPLAY THE MESSAGE BUFFER R1 POINTS ***
        ***    TO FOUR BYTES OF INFORMATION.     ***

  ? SM(R1,4)
>9560 (>0000)  >004B  >0068                 (.K  .h          )

        ***  A MESSAGE BUFFER IS PART OF A HEAP PACKET   ***
        ***  MSGBUF-4 POINTS TO THE START OF THE PACKET  ***
        *** SH (SHOW HEAP) WILL DISPLAY THE WHOLE PACKET ***

  ? SH(R1-4)
HEAP PACKET AT ADDRESS: >955C    SIZE: >000A
>955C (>0000)  >9554  >0000  >004B  >0068    (.T  ..  .K  .h  )
>9564 (>0008)  >0009                        (..              )

        ***  THE FIRST WORK OF A MESSAGE PACKET IS A  ***
        *** SEMAPHORE USED TO ASSURE EXCLUSIVE ACCESS ***
        ***    THERE ARE NO WAITERS ON IT BEFORE A    ***
        ***   MESSAGE HAS BEEN SENT TO THE CONSUMER   ***

  ? SEMA(>9554)
SEMAPHORE VALUE IS >9554
     THERE ARE NO WAITERS OR UNRECEIVED SIGNALS PRESENT
     THIS SEMAPHORE IS NOT ASSOCIATED WITH ANY INTERRUPTS
```

```
        *** IS(N) WILL SINGLE STEP EXECUTION 'N' STEPS ***

  ? IS

MOV  @RXDEMO.+>0002(R9),*R10+
    WP=>99C2   PC=>0A52   ST=>800F   SRC=>99F6  / >956E   DST=>97E2  / >0080
    WP=>99C2   PC=>0A56   ST=>800F   SRC=>99F6  / >956E   DST=>97E2  / >956E
MOV  R1,*R10+
  ? IS

MOV  R1,*R10+
    WP=>99C2   PC=>0A56   ST=>800F   SRC=>99C4  / >9560   DST=>97E4  / >9560
    WP=>99C2   PC=>0A58   ST=>800F   SRC=>99C4  / >9560   DST=>97E4  / >9560
BLWP *R7

        *** DON'T SINGLE STEP THROUGH A SUBROUTINE CALL ***
        *** IT WILL WORK, BUT IT WILL TAKE A LONG TIME  ***
        ***      TO GET BACK TO THE PRODUC PROCESS      ***

        *** INSTEAD, SET A BREAKPOINT AFTER THE CALL ***

  ? SB(PC+4)

BREAKPOINT SET AT PRODUC.+>0124  / >CEA8  MOV  @RXDEMO.+>0016(R8),*R10+
  ? GO
BREAKPOINT ENCOUNTERED AT ADDRESS: PRODUC.+>0124  / >CEA8  MOV  @RXDEMO.+>0016
8),*R10+

        *** STEP FIVE MORE INSTRUCTIONS ***

  ? IS(5)

MOV  @RXDEMO.+>0016(R8),*R10+
    WP=>99C2   PC=>0A5C   ST=>200F   SRC=>094E  / >0080   DST=>97E2  / >956E
    WP=>99C2   PC=>0A60   ST=>C00F   SRC=>094E  / >0080   DST=>97E2  / >0080
MOV  @RXDEMO.(R9),*R10+
    WP=>99C2   PC=>0A60   ST=>C00F   SRC=>99F4  / >0068   DST=>97E4  / >9560
    WP=>99C2   PC=>0A64   ST=>C00F   SRC=>99F4  / >0068   DST=>97E4  / >0068
MOV  R8,R2
    WP=>99C2   PC=>0A64   ST=>C00F   SRC=>99D2  / >0938   DST=>99C6  / >9560
    WP=>99C2   PC=>0A66   ST=>C00F   SRC=>99D2  / >0938   DST=>99C6  / >0938
AI   R2,RXDEMO.+>003D
    WP=>99C2   PC=>0A66   ST=>C00F   SRC=>99C6  / >0938
    WP=>99C2   PC=>0A6A   ST=>C00F   SRC=>99C6  / >0975
MOV  R2,*R10+
    WP=>99C2   PC=>0A6A   ST=>C00F   SRC=>99C6  / >0975   DST=>97E6  / >9566
    WP=>99C2   PC=>0A6C   ST=>C00F   SRC=>99C6  / >0975   DST=>97E6  / >0975
BLWP *R7
```

```
        *** THE ARGUMENTS HAVE BEEN PUSHED, DISPLAY  ***
        ***      THE TOP THREE ITEMS ON THE STACK     ***

  ? SM(R10-6,6)
>97E2 (>0000)  >0080  >0068  >0975          (..  .h  .u      )

        ***  >0975 IS A POINTER TO A STRING IN ROM,   ***
        *** DISPLAY THE STRING WHICH IS 15 CHARACTERS ***

  ? SM(>0975,15)
>0974 (>0000)  >0050  >524F  >4455  >4345    (.P  RO  DU  CE  )
>097C (>0008)  >5220  >5345  >4E44  >5320    (R   SE  ND  S   )

        *** SET ANOTHER BREAKPOINT AFTER THE NEXT CALL ***

  ? SB(PC+4)

BREAKPOINT SET AT PRODUC.+>0138  / >CEA8  MOV  @RXDEMO.+>0016(R8),*R10+
  ? GO

        *** THE TARGET TERMINAL JUST PRINTED 'PRODUCER SENDS ' ***

BREAKPOINT ENCOUNTERED AT ADDRESS: PRODUC.+>0138  / >CEA8  MOV  @RXDEMO.+>0016
8),*R10+

        *** NOW GET RID OF ALL BREAKPOINTS ***

  ? CB
CLEARING ALL BREAKPOINTS

        *** SHOW THE CONSUMER PROCESS ***

  ? SP(>9794)
SHOW PROCESS >9794

STACK BASE = >957C     STACK LIMIT      = >975C
STACK SIZE = >01C0     STACK USED (CUR) = >0004
PRIORITY =    256
NO OUTSTANDING EXCEPTIONS
NEXT PROCESS IN LIST = >99FC     NEXT PROCESS IN QUEUE = >9CE0
CREATOR'S ID = >03               MY ID = >05
```

```
        *** DUMP THE CONSUMER PROCESS RECORD ***

  ? PD(>9794)
DUMP OF PROCESS RECORD AT >9794  :

NEXTWS = >971C   LEX$8  = >0000   PC     = >0B60   FIRST  = >0000
CALL   = >028E   LEX$9  = >0000   ST     = >000F   NS/OBJ = >0100
UNUSED = >0000   GLOBAL = >978E   OUTPUT = >0000   OBJECT = >0000
LEX$1  = >978E   STKMIN = >957C   INPUT  = >0000   PRLIST = >99FC
LEX$2  = >0000   STKMAX = >975C   PRITY  = >0100   LSTRTD = >0000
LEX$3  = >0000   UNUSED = >0000   CLKSRV = >0000   QNEXT  = >9CE0
LEX$4  = >0000   LWPI   = >02E0   MPXPTR = >9FE6   QUEUE  = >9566
LEX$5  = >0000   CNTEXT = >971C   CL/RS  = >0000   INTLVL = >FFFF
LEX$6  = >0000   RTWP   = >0380   XHNDLR = >0000   CR/MY  = >0305
LEX$7  = >0000   WP     = >975C

        *** SET A PROCESS BREAKPOINT ON THE CONSUMER ***

  ? ABP(>9794)

  ? GO
PROCESS TRAPPED: PR = >9794

        *** SET A BREAKPOINT AFTER THE CALL TO C$RECEIVE ***

  ? SB(CONSUM.+>7E)

BREAKPOINT SET AT CONSUM.+>007E  / >C069  MOV  @RXDEMO.+>0002(R9),R1
  ? GO
BREAKPOINT ENCOUNTERED AT ADDRESS: CONSUM.+>007E  / >C069  MOV  @RXDEMO.+>0002
9),R1

        *** SHOW THE FRAME TO FIND THE ADDRESS OF THE MESSAGE ***

  ? SF
STACK FRAME FOR ROUTINE CONSUM.
>978E (>0000)  >956E  >9560                    (.n  .`          )

WORKSPACE REGISTERS ARE:
R0  = >0000   R8  = >0B02   PC  = >0B80   / >C069  MOV  @RXDEMO.+>0002(R9),R1
R1  = >9560   R9  = >978E   WP  = >975C
R2  = >9790   R10 = >957C   ST  = >800F
R3  = >0000   R11 = >0B80
R4  = >0000   R12 = >0000
R5  = >0000   R13 = >9C2A
R6  = >0000   R14 = >092C
R7  = >9794   R15 = >300F
```

```
        *** DUMP THE MESSAGE PACKET; REMEMBER THE PACKET ***
        ***   STARTS 4 BYTES BEFORE THE MESSAGE BUFFER   ***

  ? SH(R1-4)
HEAP PACKET AT ADDRESS: >955C    SIZE: >000A
>955C (>0000)  >9554  >0000  >004B  >0068     (.T  ..  .K  .h  )
>9564 (>0008)  >0009                          (..              )

        *** INDEED, THE CHARACTER IN THE ***
        ***   MESSAGE IS >4B, OR 'K'     ***

        *** SHOW THE ACCESS SEMAPHORE FOR THE MESSAGE ***

  ? SEMA(>9554)
SEMAPHORE VALUE IS >9554
     THERE ARE     1   WAITERS PRESENT
     THIS SEMAPHORE IS NOT ASSOCIATED WITH ANY INTERRUPTS

        *** THE CONSUMER HAS ALLOCATED THE MESSAGE BUFFER,  ***
        *** SO THE PRODUCER HAS TO WAIT UNTIL THE CONSUMER  ***
        *** IS FINISHED BEFORE IT CAN SEND THE NEXT MESSAGE ***

        *** NOW, LET THE PROGRAM RUN UNTIL IT FINISHES ***

  ? CB
CLEARING ALL BREAKPOINTS

  ? TP(OFF)

  ? SC(OFF)

  ? GO

        *** TYPE CHARACTERS ON THE TARGET TERMINAL UNTIL ***
        *** YOU ARE SATISFIED THAT THEY ARE SENT TO THE  ***
        ***       CONSUMER PROCESS, THEN TYPE A 'Z'.     ***

        *** HIT <CMD> ON THE HOST ***

  ? HALT

        *** SHOW THE PROCESSES LEFT IN THE SYSTEM ***

  ? DAP
STATUS SUMMARY OF ALL EXISTING PROCESSES

                          LEXICAL     CURRENT
  PROCESS NUMBER  ADDRESS  LEVEL      STATUS        PRIORITY
       2          >9CE0      1       ACTIVE          32767

        *** ONLY THE IDLE PROCESS IS LEFT ***

  ? EXIT
```

After the program is started, type a carriage return, and then type characters to be input to the PRODUCer process. To terminate the program, type a capital 'Z'.

# APPENDIX A

# Rx DATA STRUCTURES

## A.1 GENERAL

This appendix describes the data structures used by Rx. The user initialized data structures are discussed first, such as the RAM configuration table, the segment table, and the trap table. The following subsections cover the process record, global data structures, and process local data structures. The data structures used by the processor are discussed, such as the processor's registers and local variables, and stack areas.

Note that some records support the File I/O Decoder and I/O Subsystems specifically, while some contain fields pertaining to the File I/O package. For more information reference the Device Independent File I/O Package User's Manual, MP386.

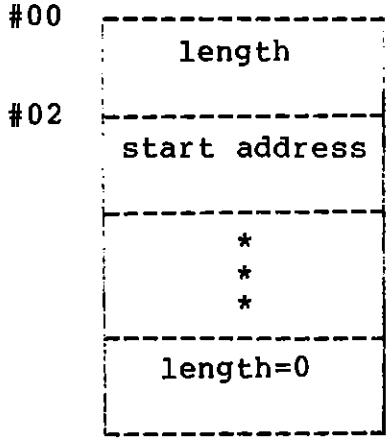
## A.2 USER INITIALIZED DATA STRUCTURES

The data structures described must be initialized by the user and defined in the "CONFIG" module. Further information on this configuration module can be found in Section VI.

### A.2.1 RAM Configuration Table

This table describes the configuration of RAM memory used as data space by Rx. It is included anywhere in the user's code space (ROM).

NOTE: This table MAY require descending addresses, and DOES require non-overlapping addresses.




Length of contiguous RAM (16-bit logical value in number of bytes)

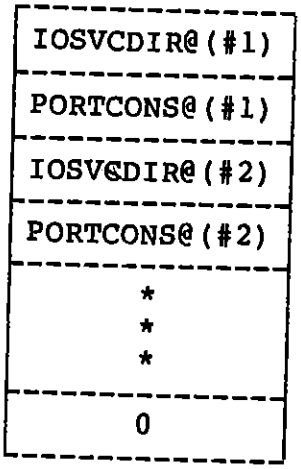
Address at which contiguous RAM starts

Length and start are repeated for each contiguous RAM area.

End of table is indicated by a length of zero.

### A.2.2 I/O Subsystem Directory

A table is included in the "CONFIG" module which lists pointers to service directory port constants defining an I/O subsystem. The last word in this directory is set to zero to terminate the list. This structure is subsystem dependent, and unless the "CONFIG" module is modified by the user, the default value is assumed. This default directory is empty and does not support an I/O subsystem.




Pointer to I/O service directory #1

Pointer to port constants list #1

Pointer to I/O service directory #2

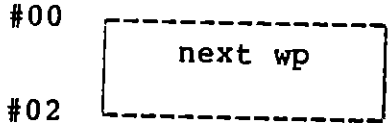
Pointer to port constants list #2

## A.3 Rx DATA STRUCTURES

The data structures described are used by Rx to manage processes, and the memory area associated with a process. The process record is the fundamental data structure used by Rx. From it one can get to all other data structures used by Rx. All data structures (except the process record shown here) are in alphabetical order.

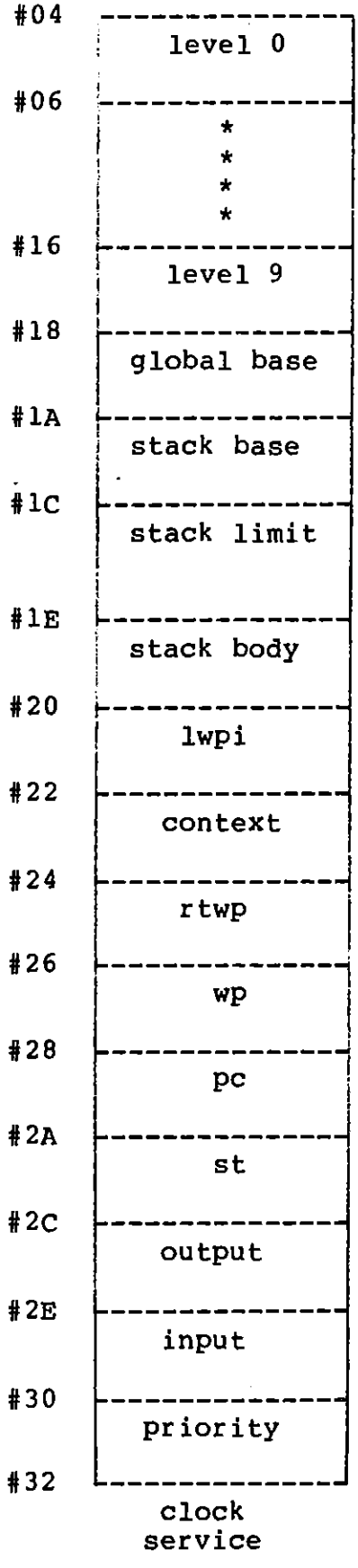
### A.3.1 Process Record

The process record is the fundamental structure which is used by Rx to access all other data structures. A unique process record exists for each instance of a process. Register 7 will always point to the process record of the currently active process. The layout of the process record is shown below:




Next workspace pointer to be used

Entry point of call handler




Display level 0 frame pointer

Intermediate level frame pointers

Display level 9 frame pointer

Stack frame of process

Minimum stack address for process

Maximum stack address for process

Not used

LWPI instruction (used in starting process)

Machine context of the process
(workspace pointer)

RTWP Instruction

The following are valid only if pointed to by the context field above.
The workspace pointer of the machine context

The program counter of the machine context

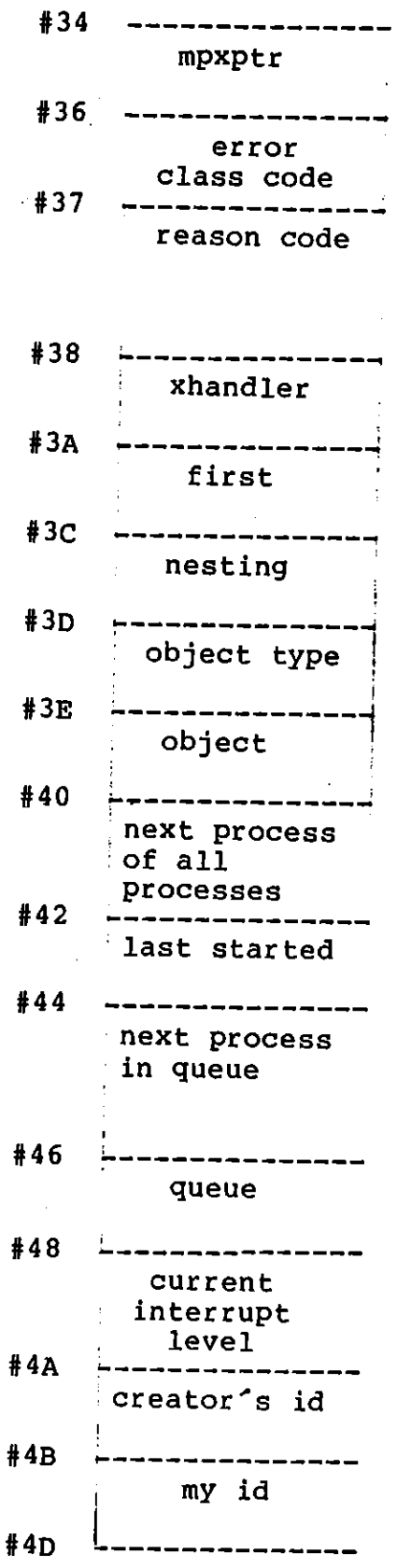
The status word of the machine context

Pointer to file descriptor for 'output'

Pointer to file descriptor for 'input'

Priority of the process

Pointer to the clock service record




Address of executive record

Error class code

Error reason code

Address of exception handler

Address of first Rx workspace

Rx nesting level

Type of first Rx structure

Pointer to the first Rx structure

Address of next process record in a circular, one-way list of ALL process records.

Address of process record last started by this process.

Address of next process record in a queue (semaphore or,scheduling queue) or nil if this process is the last member or is not in a queue.

Address of current or last semaphore queue

Interrupt currently being serviced. (-1 if no interrupt in progress)

See explanation below.

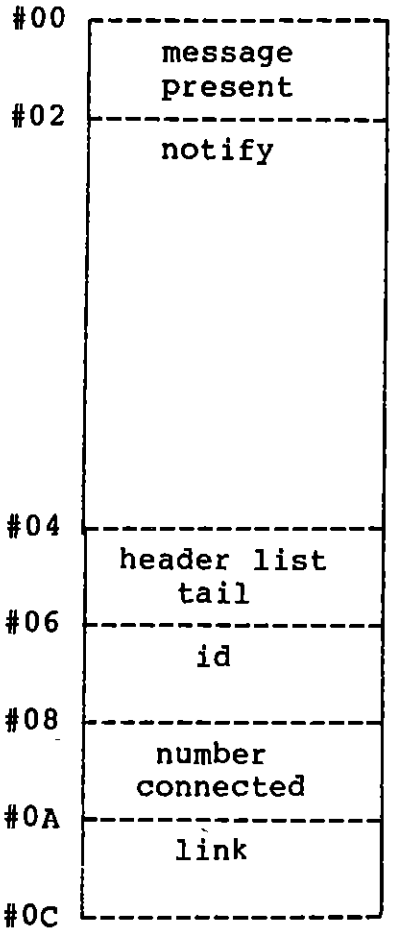
See explanation below.

The field of the process record called "my id" (displacement >4B) is set to a value as follows. A count is kept of all processes started (stored in process management record). If this count is less than 256, then it is stored in "my id" when the process is first created. If this count is greater than or equal to 256 at the time a process is first created, then the most significant byte of the count is stored in "my id" of the new process record.

The field of the process record called "creator's id" (displacement 4A) in a new process record is set to the value of "my id" of the process which created the new process.

### A.3.2 Channel Record

The following record is referenced by the executive record and subdirectory records. It is used to control the message flow from one process to another.




Address of a semaphore which signals the availability of a message on the channel

Address of a user defined semaphore (This field is set by a special channel routine. This channel routine will signal the notify semaphore if it has been initialized and a message was sent over the channel. This allows the user to create processes which can be notified when a message is present on one of several channels that the process services. The process can then perform conditional receives to determine which channel has the message)

Address of the last message in a channel's circular buffer queue of messages.

Integer corresponding to a channel's identification number.

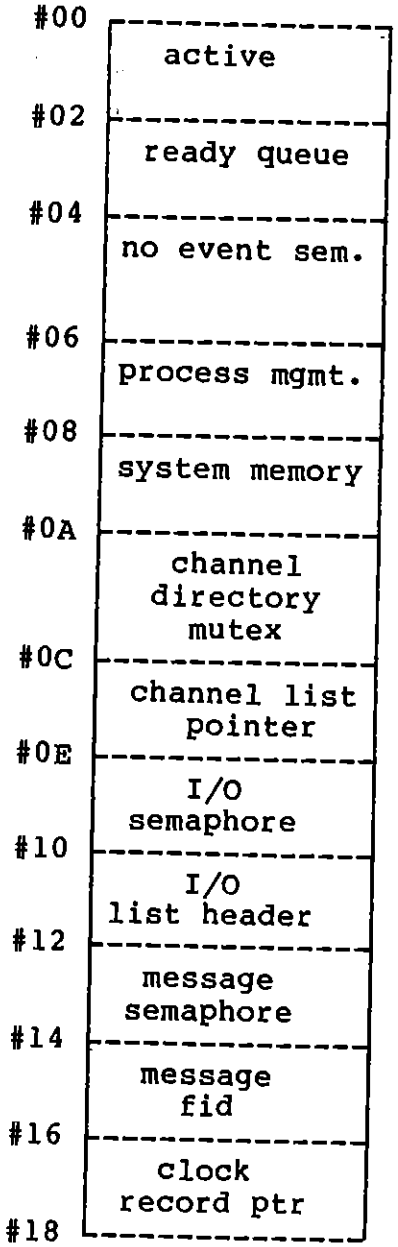
The number of readers and writers currently connected to the channel.

Address of next channel in list.

### A.3.3 Executive Record

The following record exists once in Rx

and points to all other fundamental data structures. Every process record has a pointer to this record.




Address of active process.

Address of ready queue. All ready processes are linked in this queue.

Address of NO EVENT semaphore (used by inter processing routines.) This semaphore has no waiters on it.

Address of process management record.

Address of heap record for all data space used by Rx

Address of the exclusive access semaphore for channel list

Pointer to the first channel on the channel list

I/O subsystem semaphore

I/O subsystem list header

"Message" file identifier semaphore

Address of file identifier for "message"

Pointer to the clock record

### A.3.4 File Identifier Record

Each process has a unique file identifier (FID) record associated with it, one for every file variable it declares. The file identifiers allow the process to access the data structures required to implement I/O for a particular subsystem. The FID records for a process are linked together.

| | | |
|---|---|---|
| #00 | link | Pointer to next FID in the linked list |
| #02 | subsystem | Pointer to the subsystem record associated with the file |
| #04 | status | Status of the file |
| #06 | state | State of the file |
| #08 | variables | Pointer to the subsystem dependent variable record |
| #0A | global frame | Address of the process's global frame |
| #0C | | |

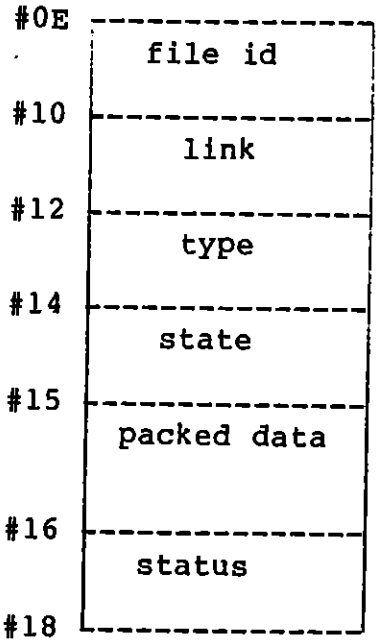
### A.3.5 FID Variables Record

The file identifier (FID) variables record is a subsystem dependent structure. It contains the variables required for a given subsystem. These variables may be different for different subsystems, so the FID variables record will be unique for different subsystems. An example is the IPC$FID variables record described in subsection A.3.8.1.

### A.3.6 File Descriptor

A file is implemented by the following record. In Microprocessor Pascal a file variable is a pointer to a file descriptor. This record is not used in RXKERNEL.

| | | |
|---|---|---|
| #00 | fill | Not used |
| #02 | file name | Pointer to the string containing the file's name |
| #04 | file name length | Length of the file's name |
| #06 | next | Pointer to next element in the buffer |
| #08 | last | Pointer to the last element in the buffer |
| #0A | line buffer | Address of the file's buffer |




Address of file identification record (FID)

Pointer to next file descriptor defined within the same process

Packed record defining file format, record format, file usage, and file compression

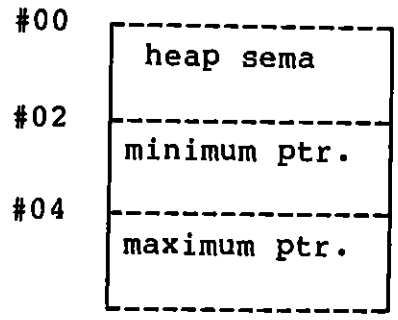
File state ( closed, open for reading, open for writing, or eof )

Terminate on error flag ( 1 byte )
File kind ( sequential, text, or random )
I/O in progress flag ( 1 byte )

Error class and reason

### A.3.7 Heap Record

Each heap is administered through the following heap record. A heap record is referenced from each process record.



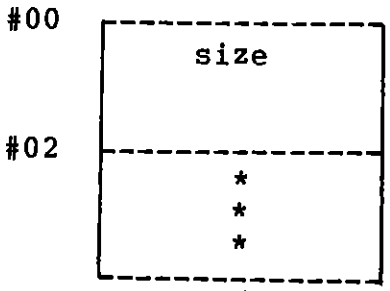

Address of semaphore ensuring mutually exclusive access to heap data structures

Value of smallest valid heap pointer

Value of largest valid heap pointer

#### A.3.7.1 Free Heap Packet

A heap packet which is not allocated has the following format.

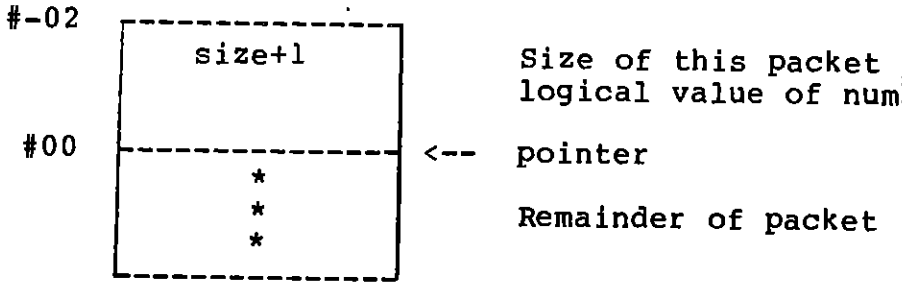



Size of this packet (16-bit logical value of number of bytes)

Remainder of packet

### A.3.7.2 Allocated Heap Packet

A heap packet, which is allocated by a process, is referenced by the process through a pointer and has the following format.

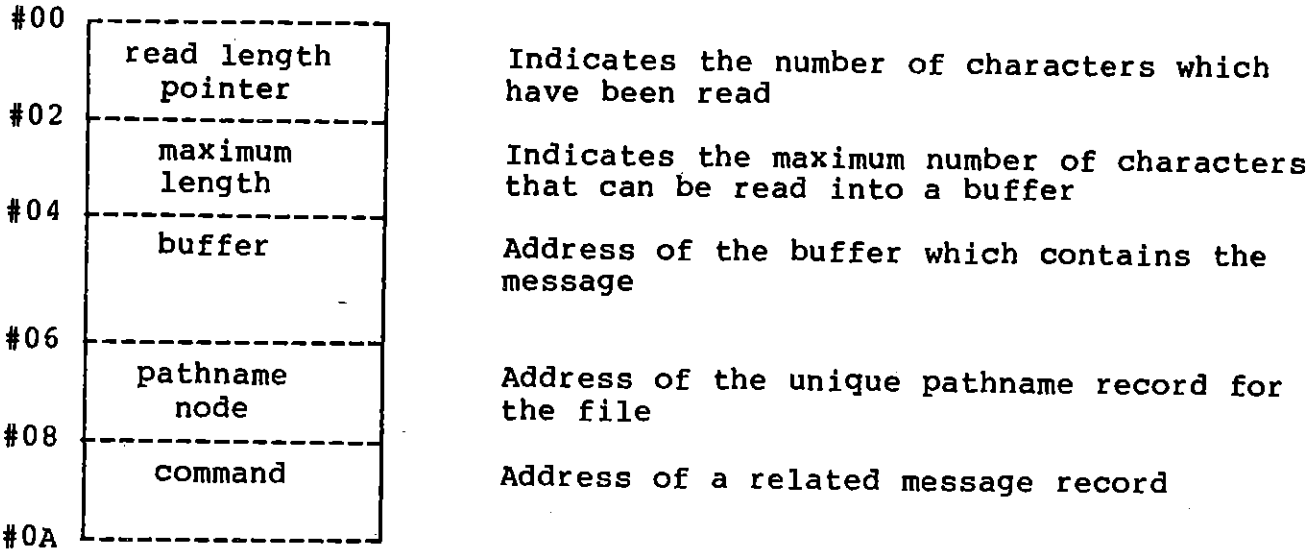



### A.3.8 Interprocess Data Structures

These data structures are used exclusively to implement interprocess communication. The following data structures allow messages to be passed through channels. Note that these specifically apply to the File I/O Decoder and I/O Subsystems.

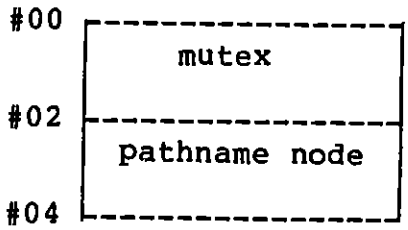
### A.3.8.1 IPC$FID Variables Record

This record is accessed through a FID record. It contains the addresses of read parameters, the file's message buffer, and a pointer to the pathname record.



### A.3.8.2 IPC$Port Variables Record

This record is accessed through an IPC-subsystem record. It points to a linked list of pathname records, each containing the unique characteristics of a particular file.

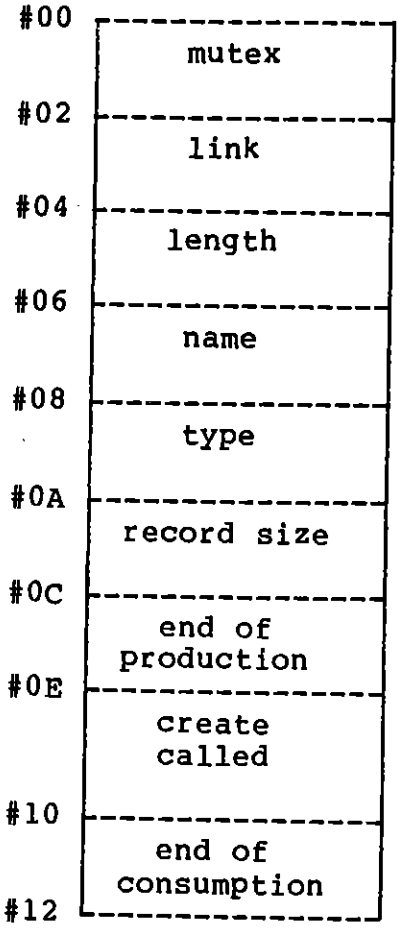



Address of the semaphore used to insure mutual exclusion when accessing the list

Address of the first pathname in the list

### A.3.8.3 Pathname Record

This record is accessed through either the pathname node field of the process IPC$FID variable record or the IPC$port variables record. The pathname record contains characteristics unique to a given file. Also contained are values used to access and synchronize access to the file's channel.




Address of the semaphore used to ensure exclusive access to the pathname record

Address of next pathname in linked list

Number of characters in the file's name

Address of the string containing the file's name

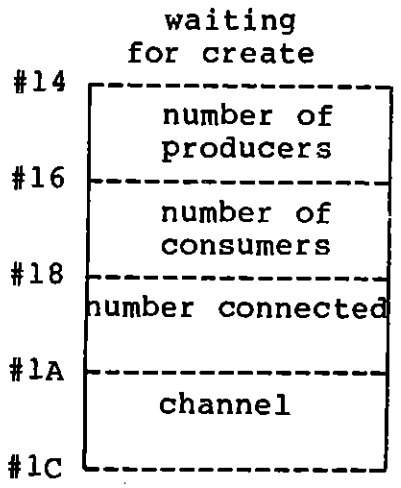
Packed record defining file format, record format, file usage, and file compression

Maximum number of characters in a logical record

Boolean indicates if all producers have closed on a channel

Boolean indicates if file has been created

Points to a semaphore used to synchronize the closing of producers




Points to a semaphore used to synchronize the creation of a file

The number of processes writing to a file

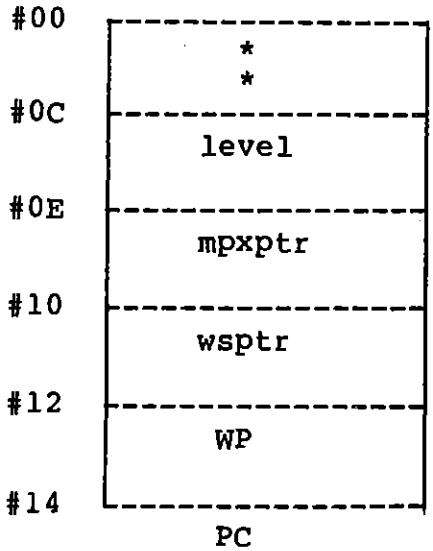
The number of processes reading a file

The number of processes connected to a file

Address of the unique channel associated with this pathname

### A.3.9 Interrupt Record

The interrupt records in Rx are stored in the workspace for the individual interrupt levels. In particular, R11 and R12 contain the addresses of the externalevent and altexternalevent semaphores. If no semaphore has been associated with the interrupt level, both of the registers will point to the NOEVENT semaphore (where there are no waiters present). R9 and R10 contain the 9900 context of the assembly event handler. If no handler has been specified, the WP portion of the vector is set to 0.



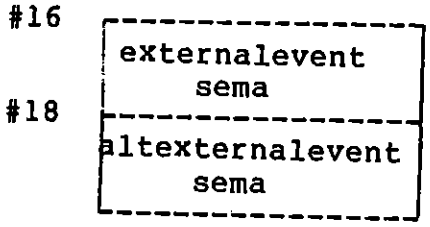

Interrupt level

Address of Rx executive record

Context workspace pointer

9900 context of interrupt handler
If no handler, WP is set to zero

Program counter of interrupt handler

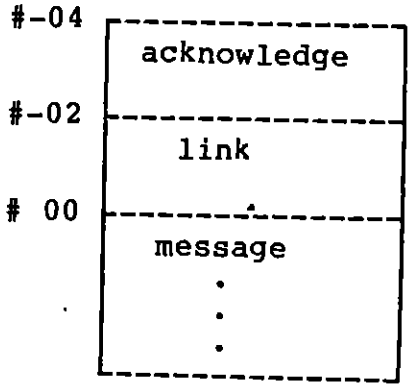



Address of EXTERNALEVENT semaphore or NOEVENT

Address of ALTEXERNALEVENT semaphore or NOEVENT

### A.3.10 Message Descriptor

Interprocess messages are passed by means of pointers to the starting address of the message field in the message descriptor. The two words which precede the message field are used by the ipc$ routines for synchronization and are normally not accessed by the user. The message descriptor is defined as follows:



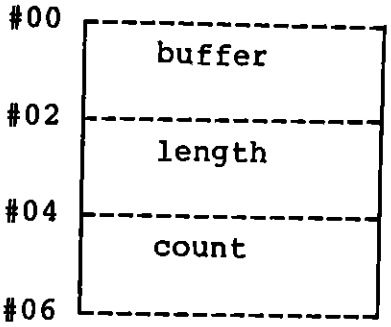

Semaphore signalled when a message has been processed

Address of next message ( used in the circular buffer of a channel )

Variable length message field

### A.3.11 Message Record

Interprocess communication data is transmitted through a message record.



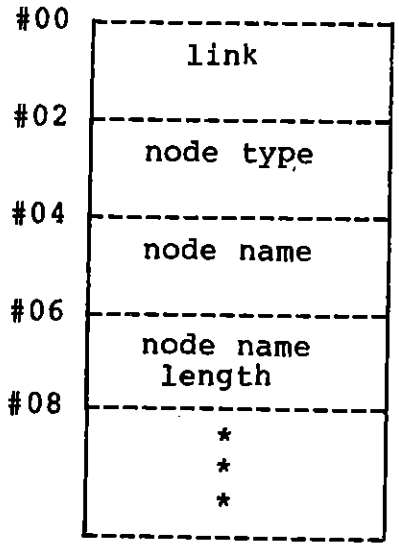

Address of the message sent

Number of total bytes in the message field

Number of bytes actually sent in the message field

### A.3.12 Node Header Record

The node header record defines one or more nodes associated with a given port. It is referenced by port constants records. Note that this record applies specifically to the File I/O Decoder and I/O subsystems. For further information, reference the Device Independent File I/O Package User's Manual, MP386.




Points to the next node header record

Indicates the type of the node

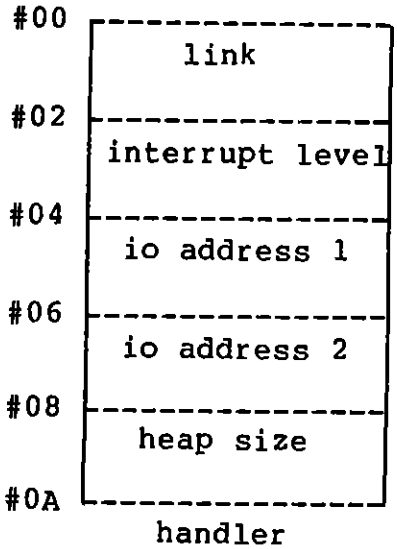
Pointer to the string containing the node's name

Length of the node's name

Subsystem dependent fields

### A.3.13 Port Constants Record

This structure contains the constant values necessary in performing I/O operations with a particular device. Note that this record applies specifically to the File I/O Decoder and I/O Subsystems. For further information, reference the Device Independent File I/O Package, MP 386.




Pointer to next port constants record

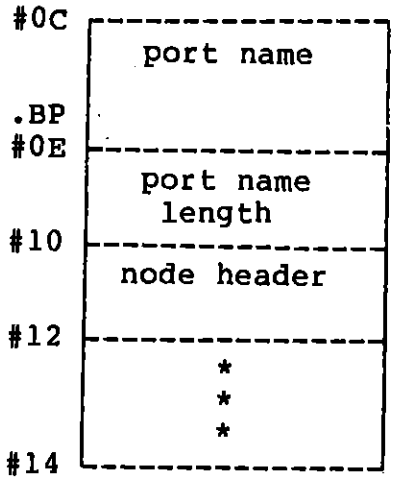
Indicates the interrupt level of a device

Port address 1 ( used when appropriate to specify a memory mapped I/O port )

Port address 2 ( used when appropriate to specify a memory mapped I/O port )

Size of the heap packet allocated to the subsystem

Address of interface handler




Address of the string containing the port's name

Length of the port's name

Pointer to the associated node header record

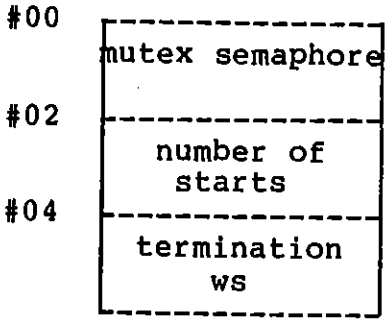
Subsystem dependent fields

### A.3.14 Port Variables Record

The port variables record contains the variables necessary for I/O with a device. Since the required variables may be different for different devices, this structure is subsystem dependent. Note that this record applies to the File I/O Decoder and I/O Subsystems. Reference the Device Independent File I/O Conversion Package, MP386, for further information.

### A.3.15 Process Management Record

The following record is referenced by the Executive Record and exists once in Rx.



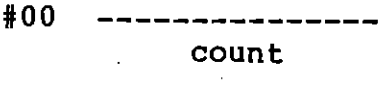

Address of semaphore ensuring mutual exclusion in process management

Number of started processes

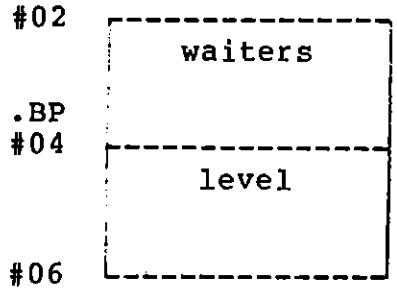
Address of termination workspace

### A.3.16 Semaphore Record

A semaphore is a pointer to a semaphore record described below.




Count of semaphore (returned by SEMAVALUE function).

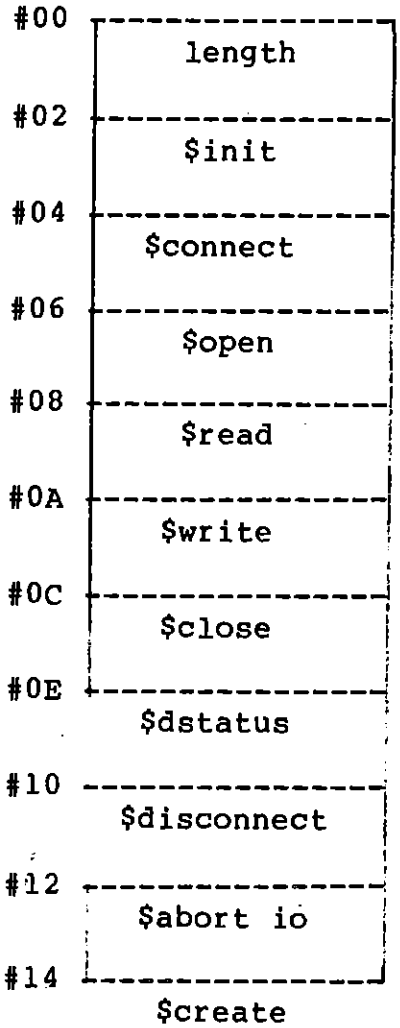



Address of queue record in which waiting processes are enqueued.

Maximum value of priority which may wait on this semaphore (32767 except when associated with an interrupt.)

### A.3.17 Service Directory Record

The service directory record provides the capability of accessing different code for different subsystem types. The fields in this record contain the addresses of the various subsystem routines. For example, the service directory record for the IPC subsystem contains the starting locations for each of the IPC routines.




Total size of this record ( Currently not used )

Address of xxx$init routine

Address of xxx$connect routine

Address of xxx$open routine

Address of xxx$read routine

Address of xxx$write routine

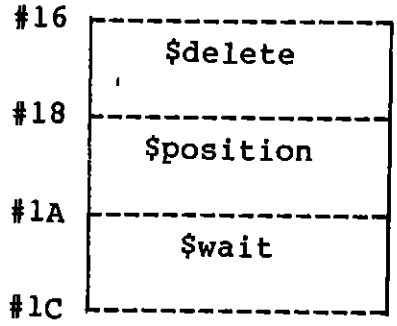
Address of xxx$close routine

Address of xxx$dstatus routine

Address of xxx$disconnect routine

Address of xxx$abortio routine

Address of xxx$create routine



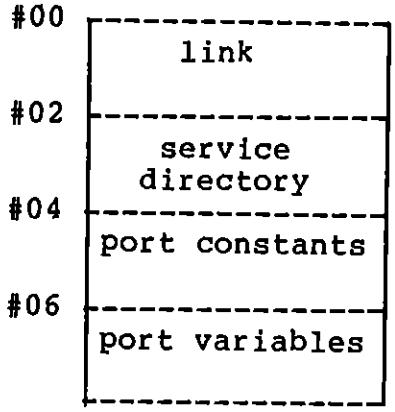

Address of xxx$delete routine

Address of xxx$position routine

Address of xxx$wait routine

### A.3.18 Subsystem Record

The subsystem record contains a pointer to the service directory record which is associated with a particular subsystem. Pointers to the port constants and variables are also included in the subsystem data structure.




Pointer to the next subsystem record

Pointer to the associated service directory record

Pointer to the subsystem dependent port constants record

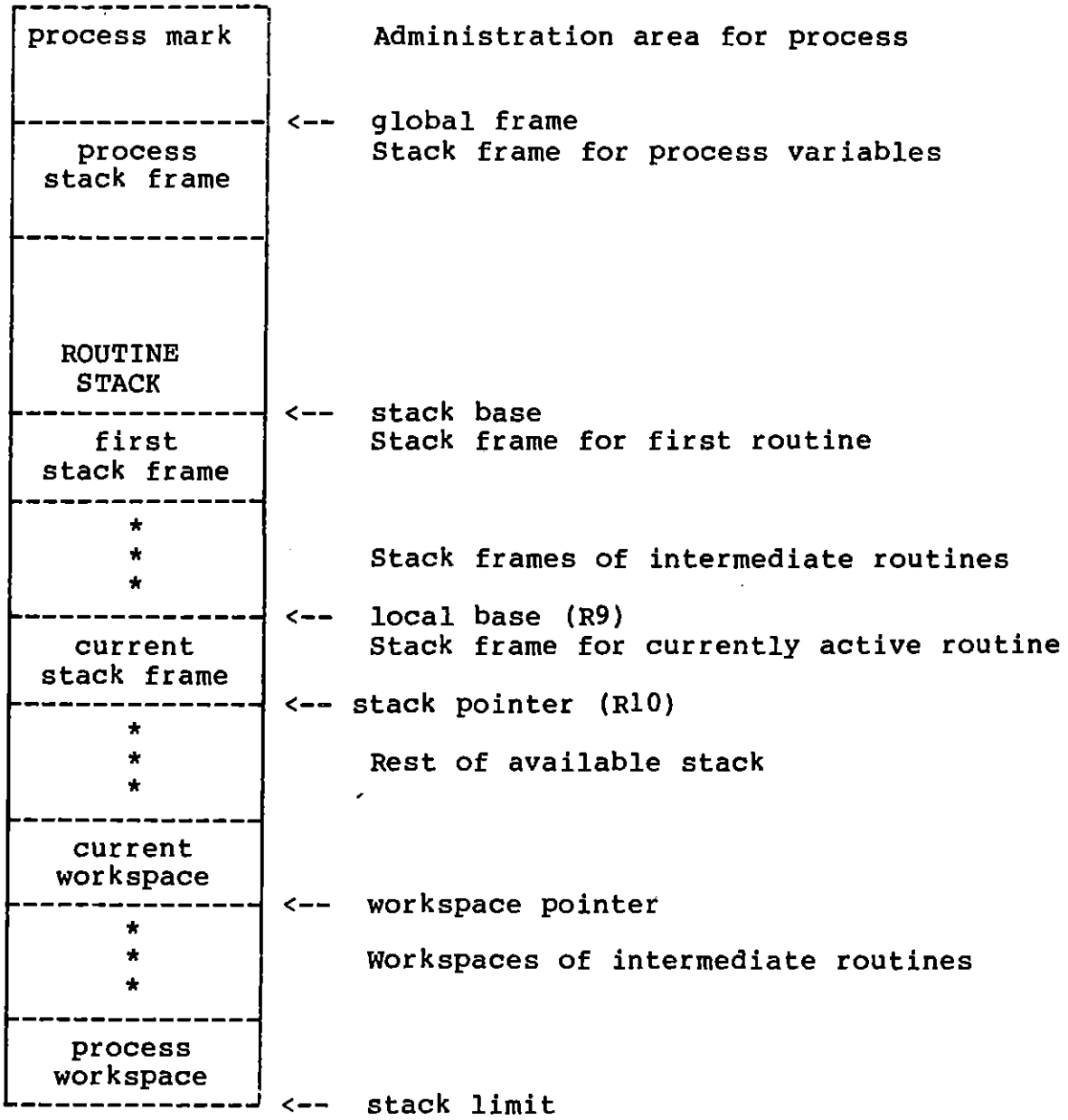
Pointer to the subsystem dependent port variables record

## A.4 STACK DATA STRUCTURES

The data structures described are used by Rx to maintain the stack structures in memory. These stack structures are necessary to support concurrency and reentrancy.

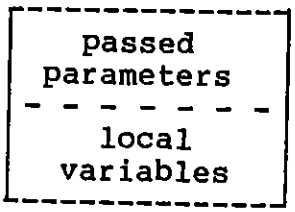
### A.4.1 Process Stack

The stack for a process is allocated as two separate regions, the first is the stack frame for the process, and the second is the stack to be used by routines which are called from the process. The two regions have the following format:




### A.4.2 Stack Frame

Each stack frame contains the values of the variables and parameters for a given routine. The stack frame consists of two regions, either of which may be zero length. A stack frame is shown on the following page:

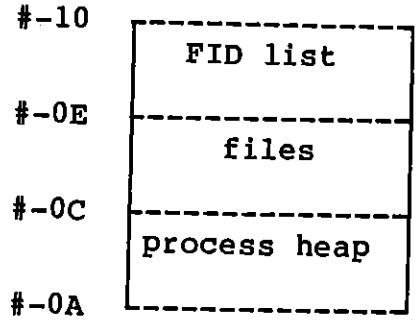



<-- R9
Parameters pushed onto stack before calling routine

Reserved storage for variables used within the routine

### A.4.3 Process Mark

The administration area below the frame of a process is used to hold information about the process.

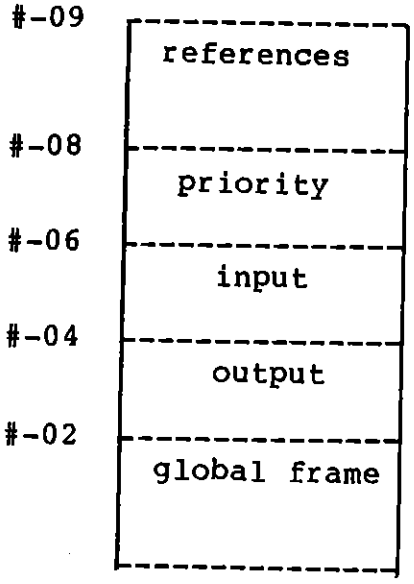



Pointer to the list of FID records

Pointer to the list of files which is created during process termination

Address of heap record for this process

Boolean which is true if this process created this process's heap, or is false if this process is sharing a nested heap (1 bit).
Boolean equal to process heap termination flag (1 bit).
Boolean equal to process start termination flag (1 bit).
Boolean indicates if process has completed (1 bit).
Lexical level of this process (5 bits).




Number of references to this process frame by processes which can address this frame.

Priority of this process

Pointer to this process's input file

Pointer to this process's output file

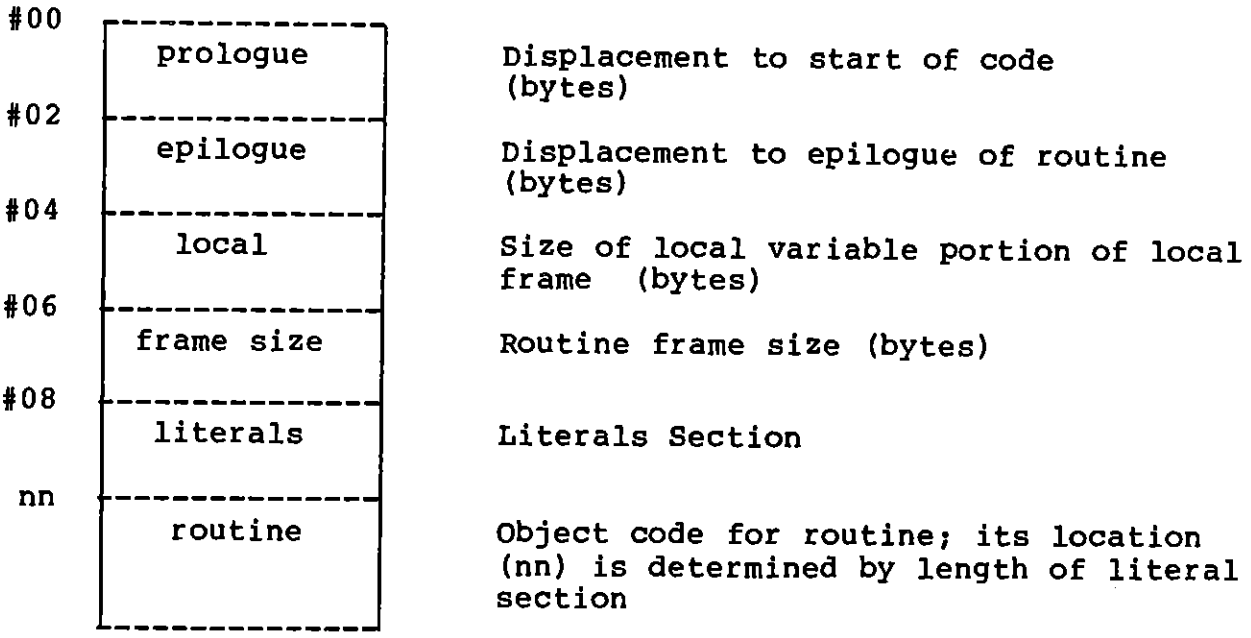
Address of global frame of lexical parent if lexical level is 2 or greater. (otherwise zero)
<--- Global frame pointer

## A.5 ROUTINE DESCRIPTORS (CONTAINED IN PROLOGUE OF THE CODE)

The routine descriptor contains information about a procedure, function, or process which is used when it is called. This information includes data such as the start of code, data frame size, parameter size, end of code, and the number of registers that need to be initiallized.

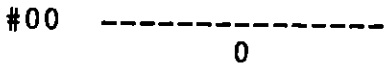
### A.5.1 Standard Procedure/Function Descriptor

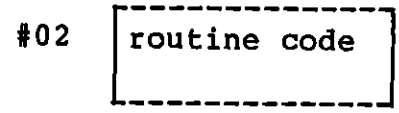
The procedure/function descriptor is used by the linkage routines when the routine is called. The first four fields of the process descriptor record are fixed and are always generated by the compiler. The remaining fields following the frame size may or may not be generated by the compiler and included in a process descriptor record. This will occur if the compiler can optimize the use of these fields in the code. The standard linkage supports the passing of parameters or local variables.




### A.5.2 Optimized Procedure/Function Descriptor

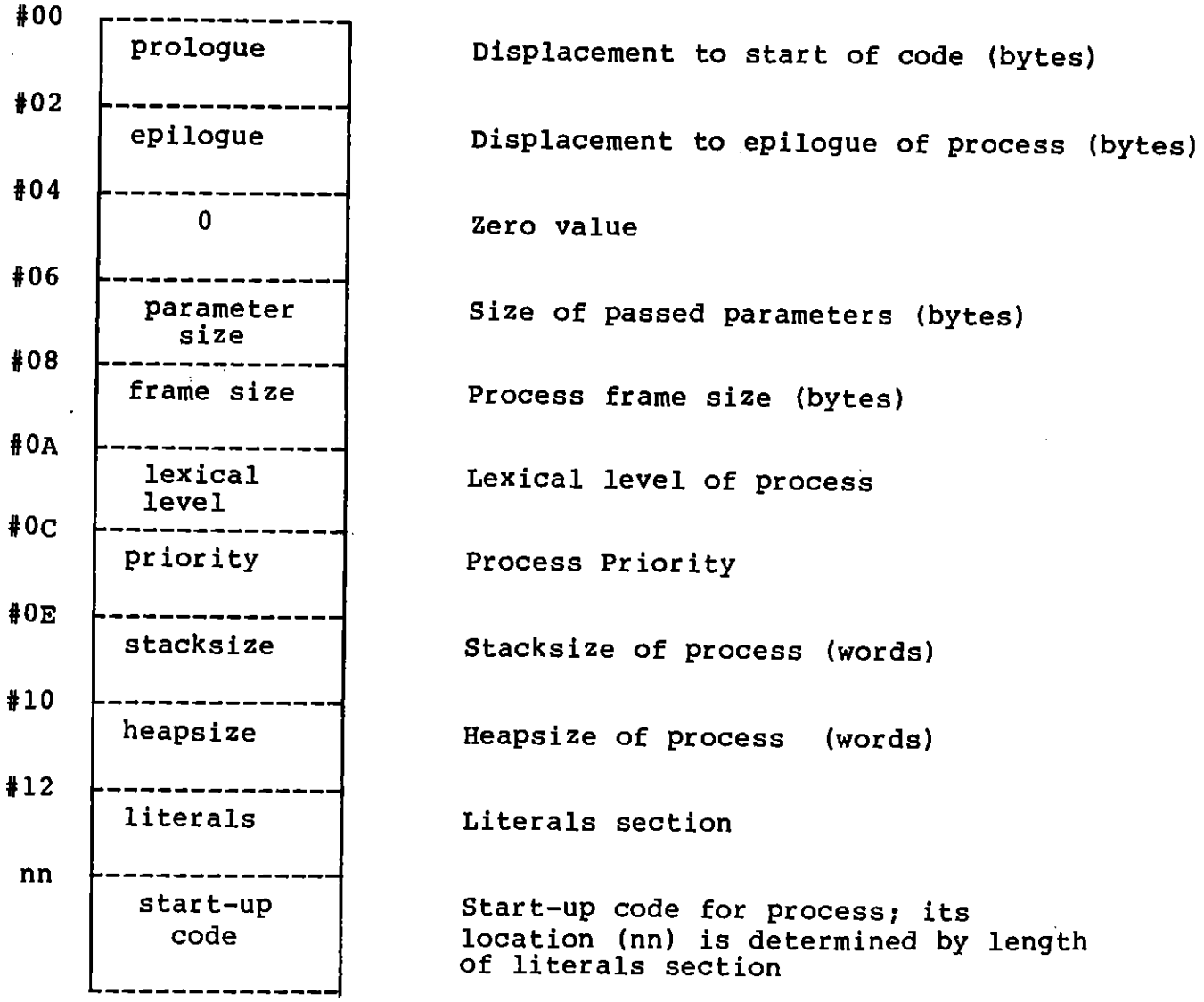
The optimized procedure/function linkage executes faster than the standard linkage. However, this linkage does not directly support passing of parameters or local variables.






### A.5.3 Process Descriptor

The data structure for a process is basically the same as that for a procedure or function, containing several additional parameters for system use. Note that the Executive will only look at the first four fields. The remaining fields can be used as the programmer wishes. If the remaining fields are used, it is suggested that they be listed in the order shown below.

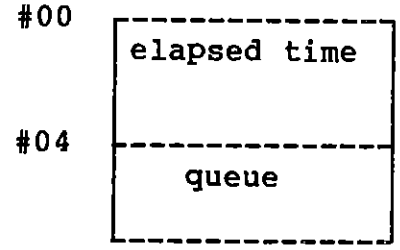


## A.6 CLOCK DATA STRUCTURES

The clock process (CLKINT) maintains an elapsed time field in memory along with a list of time elements which should be signalled at certain specified times.

### A.6.1 Clock Record

The clock process increments a two-word integer value every 10.0053 milliseconds. This two word integer is stored in a portion of common data space with the following structure:

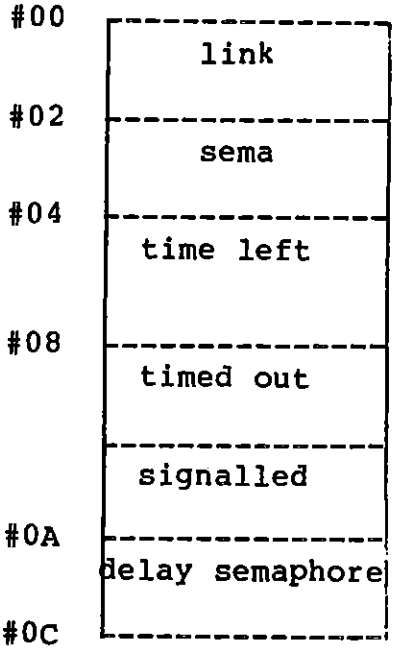



Integer length of current execution time (two word value with most significant word first)

Address of the first clock service record waiting on the clock for some event

### A.6.2 Clock Service Record

A clock service record is created for each process performing a timed wait or delay function. These time elements are maintained in a list pointed to by "queue" in the clock record structure. When a clock service record is initially inserted into this list a field is set indicating a length of time that the structure is to remain in the waiting queue. The time is decremented on each interrupt. When the length of time to wait is zero the clock service record is removed from the queue list and the event (such as a delay) is signalled.




Address of next clock service record in the waiting queue (twait_q list)

Address of semaphore to be signalled when time has expired

Time remaining until the semaphore should be signalled (two word value with most significant word first)

Boolean flag set to true when the 'time left' reaches zero

Boolean flag set to true when 'sema' is signalled

Address of user semaphore used to implement delay procedure

# APPENDIX B

# Rx ERRORS CODES, ERROR RECOVERY, AND EXCEPTION HANDLING

## B.1 GENERAL

As a process executes, it may encounter an exception such as division by zero or subscript out of range. The ability of a process to deal with and possibly recover from exceptions is called exception handling. The mechanism by which a process can recover from exceptions and reprocess lost work is explained in this section.

## B.2 EXECUTIVE RTS DETECTED ERRORS

The Rx RTS passes error codes to RxBUG when a system failure occurs, and a message is displayed by RxBug in the form:

ERR = xxyy

where xx is the class code, and yy the reason code.

All class codes are unique, with the expection of System Crash Codes and Run-Time Support errors. Both are: 00. The difference is distinguishable by the fact that System Crash Codes do not appear in the process record error field; i.e., >36 bytes from the start of the process record.

If a SHOWHEAP command is performed on the process record of the active process, the error code will be in the fourth row at the fourth position . A zero in this position means no error was written to the process record.

NOTE: a zero in the fourth position of the fourth row does not mean no error occurred, but simply that one wasn't detected. A DR command shows all registers. Check Register 0 of the current workspace to identify such an error. A non-zero value in Register 0 is indicative of a System Crash error. Table B-1 lists all RTS-detected errors and their meanings. Table B-2 explains the meaning of each reason code and provides remedial action, if applicable.

TABLE B-1. ERROR CODES.

| CODE Class (xx) | CODE Reason (yy) | MEANING |
|---|---|---|
| **SYSTEM CRASH CODES (00) (APPEAR IN R0 ONLY)** | | |
| 00 | 01 | Unable To Boot System |
| 00 | 02 | No Exception Handler |
| 00 | 03 | No Interrupt Handler |
| 00 | 04 | Illegal Interrupt or XOP |
| 00 | 05 | Scheduling Queue In Error |
| 00 | 06 | ROM/RAM Partitioning Error |
| 00 | 07 | Process List In Error |
| 00 | 08 | Heap In Error |
| **RUN-TIME SUPPORT ERRORS (00)** | | |
| 00 | 02 | Stack Overflow |
| 00 | 04 | Division By Zero |
| 00 | 05 | Floating Point Error |
| 00 | 06 | Set Element Out Of Bounds |
| 00 | 07 | Assert Error |
| 00 | 08 | Missing OTHERWISE In CASE |
| 00 | 09 | Array Index Error |
| 00 | 0A | Pointer Equals NIL |
| 00 | 0B | Subrange Assignment Error |
| 00 | 0C | LONGINT Array Index Error |
| 00 | 0D | LONGINT Subrange Error |
| 00 | 14 | Halt Called |
| 00 | 15 | String Index Error |
| 00 | 16 | String Length Error |
| **USER DEFINED ERRORS (01)** | | |
| 01 | YY | Defined by the user as a parameter to the Rx procedure: EXCEPTION |

(Continued)

TABLE B-1. ERROR CODES (Continued).

| CODE Class (xx) | Reason (yy) | MEANING |
|---|---|---|
| | | SCHEDULING ERRORS (02) |
| 02 | 01 | Invalid Queue |
| 02 | 02 | Priority Error |
| | | SEMAPHORE ERRORS (03) |
| 03 | 01 | Invalid Semaphore Structure |
| 03 | 02 | Count Error |
| 03 | 03 | Operation Error |
| 03 | 04 | Count Overflow |
| 03 | 05 | Priority Error |
| 04 | 02 | Level Invalid |
| 04 | 03 | Semaphore Invalid |
| 04 | 04 | No Interrupt Handler |
| 04 | 05 | No Dedicated Interrupt Workspace (Rx 1.1) |
| | | PROCESS MANAGEMENT ERRORS (05) |
| 05 | 03 | Invalid priority |
| 05 | 04 | Negative Stack Size |
| 05 | 0B | No Memory |
| | | EXCEPTION HANDLING ERRORS (06) |
| 06 | 01 | Handler Not Established In Process |
| 06 | 02 | Handler Cannot Have Parameters |
| 06 | 03 | Not Enough Memory For Handler |

(Continued)

TABLE B-1. ERROR CODES (Continued)

| CODE Class (xx) | Reason (yy) | MEANING |
|---|---|---|
| | | **MEMORY MANAGEMENT ERRORS (07)** |
| 07 | 01 | Heap Pointer Invalid |
| 07 | 02 | Heap Overflow |
| 07 | 03 | Invalid Packet Pointer |
| 07 | 04 | Attempt To Free A Packet |
| | | **FILE I/O ERRORS (08)** |
| 08 | 01 | File Not In State For Reading |
| 08 | 02 | File Not In State For Writing |
| 08 | 03 | Sequential End Of File |
| 08 | 04 | Error In Opening File |
| 08 | 05 | Read Error |
| 08 | 06 | Write Error |
| 08 | 07 | No Memory For File Descriptor |
| 08 | 08 | No Memory For Pathname |
| 08 | 09 | File Not Closed |
| 08 | 0A | Invalid Parameter Passed To F$STLENGTH |
| 08 | 0B | Not A Text File |
| | | **TEXT I/O ERRORS (09)** |
| 09 | 01 | Text Conversion Parameter Out Of Range |
| 09 | 02 | Text Conversion Field Width Too Large |
| 09 | 03 | Incomplete Text Conversion Data |
| 09 | 04 | Invalid Character In Text Field |
| 09 | 05 | Text Conversion Value Too Large |
| 09 | 06 | Text Read Past End Of File |
| 09 | 07 | Text Field Exceeds Record Size |
| | | **CHANNEL ERRORS (10)** |
| 10 | 01 | No Memory For Buffers |
| 10 | 02 | No Memory For Semaphore |
| 10 | 03 | No Memory For Channels |

(Continued)

TABLE B-1. ERROR CODES (Continued).

| CODE Class (xx) | CODE Reason (yy) | MEANING |
|---|---|---|
| | | FILE I/O DECODER ERRORS (11) |
| 11 | 01 | Empty File ID List |
| 11 | 02 | File ID Not Found |
| 11 | 03 | File Not Released |
| | | INTERPROCESS COMMUNICATION ERRORS (12) |
| 12 | 01 | No Heap For Pathname Record |
| 12 | 02 | No Heap For Name Field |
| 12 | 03 | No Heap For File Variable Record |

---

## B.2 EXPLANATION OF REASON CODES

Table B-2 provides a detailed explanation for each reason code given in Table -1.

---

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

### SYSTEM CRASH CODES (00)

Non-recoverable errors are defined to cause a system crash, which results in the execution of the system crash code that is specified by parameter $SYSCR in the user's CONFIG module (See Section 8). A description of these errors follow:

01 Executive RTS is unable to boot the system, probably because of insufficient memory.

02 A system, program, or process fails without having established an exception handler.

03 An interrupt occurs at a level for which no handler has been specified at the time of the occurrence of the interrupt.

04 An unimplemented interrupt or XOP occurs and cannot be serviced.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

05 The scheduling queue has been destroyed; further scheduling is impossible.

06 RAM made available to Executive RTS is found to be in error. An address specified to be RAM is either bad, ROM, or unimplemented memory.

07 The process list is in error.

In attempting to deallocate the resources associated with a process, the process record of that process could not be found in the list of all active processes. The probable cause is the destruction of system data structures.

08 The heap pointer is invalid. An invalid heap packet pointer was encountered while attempting to deallocate a process.

RUN-TIME SUPPORT ERRORS (00)

02 Stack Overflow: This error occurs when the allocated stack memory region is exhausted. The problem can normally be remedied by increasing the stack size parameter.

04 Division By Zero: This error occurs when division by zero is detected. The offending expression should be checked and corrected to avoid this error.

05 Floating Point Error: This error occurs when a REAL value is too large or too small to be represented. The range of absolute values that can be represented is about 1.0E-78 to 1.0E75.

06 Set Element Out Of Bounds: This error indicates that a member of a set has an ordinal value less than 0 or greater than 1023. This problem can be solved by restructuring the set or breaking it into more than one set if necessary.

07 Assert Error: This error occurs when the expression in an ASSERT statement evaluates to "false". Either the expression was improperly formed or a logical error occured at some point in the program.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

08 Missing OTHERWISE in CASE: This error occurs when the selector expression in a CASE statement does not evaluate to any of the case labels present and there is no OTHERWISE clause to be used as the default statement. If there are no logical errors in the program, an OTHERWISE clause should be added so that unanticipated label values will be handled uniformly.

09 Array Index Error: This error occurs when an array index is out of bounds for the array. The error may have been caused by an incorrectly formed index expression(s). Alternatively, the array definition may be incorrect.

10 Pointer Equals NIL: This error occurs when a reference is attempted through a pointer which has the value NIL. (No check is made to ensure that the pointer points to a valid ((allocate)) heap packet.) To avoid this error, make sure that all pointers have a valid, non-NIL value before they are used.

11 Subrange Assignment Error: This error occurs when a subrange variable is given a value that is outside its range. This could be the result of an unanticipated assignment or function result. Expressions should be examined to ensure that their values are in bounds; alternatively, the subrange bounds may have to be altered.

12 LONGINT Array Index Error: This error occurs when a LONGINT array index is out of bounds.

13 LONGINT Subrange Error: This error occurs when a LONGINT subrange variable is given a value that is outside its range.

20 HALT called: The procedure HALT has been called.

## USER-DETECTED ERRORS (01)

A user error can be forced by calling the routine EXCEPTION with its class code parameter set to 1 to denote a user-detected error. The process which executes this routine fails with some designated reason code (as specified by the user as a parameter to EXCEPTION).

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

SCHEDULING ERRORS (02)

01 Invalid Queue: This error should not be seen by the user. It indicates a system error which probably resultd from Rx code being accidently modified.

02 Priority Error: This error occurs if SETPRIORITY is called with an interrupt priority (in the range 0 to 15). The priority of a process cannot be set to an interrupt priority.

SEMAPHORE ERRORS (03)

01 Invalid Semaphore: This error indicates that a structure that was passed to a semaphore routine is not a valid semaphore. The error occurs primarily in cases when a semaphore is used before it has been initialized by INITSEMAPHORE or after it has been terminated by TERMSEMAPHORE; otherwise it is a run-time support error which may be a result of system data structures being accidently destroyed.

02 Count Error: This error can occur when INITSEMAPHORE is called with a count value that is not in the range 0 to 32767. A semaphore cannot be initialized to a negative value.

03 Operation Error: This error can occur when a semaphore operation is attempted and fails. TERMSEMAPHORE can produce this error if the semaphore being terminated has waiters.

04 Count Overflow: This error occurs whenever the counter associated with a given semaphore becomes equal to 32767, meaning that no more events can be signaled until some waiting processes perform a WAIT.

05 Priority Error: This error occurs when a process attempts to WAIT on a semaphore that is associated with an interrupt and the priority of the process is less urgent than the level of the interrupt.

INTERRUPT ERRORS (04)

02 Level Invalid: This error occurs when the priority passed to one of the routines ALTEXTERNALEVENT, EXTERNALEVENT, NOALTEXTERNALEVENT, NOEXTERNALEVENT, ASSEMBLYEVENT, or NOASSEMBLYEVENT is not in the range 1 to 15.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

03 Semaphore Invalid: This error results if an invalid semaphore is passed to ALTEXTERNALEVENT or EXTERNALEVENT.

04 Interrupt Not Handled: This error occurs when an interrupt is signaled and there is no process waiting to service the interrupt.

PROCESS MANAGEMENT ERRORS (05)

03 Not Started - Invalid Priority: This error occurs when it is not possible to start a user process because the priority given in the concurrent characteristics for the process is not in the range 0 through 32766.

04 Not Started - Negative Stacksize: The stacksize given in the concurrent characteristics for the process must be non-negative.

11 Not Started - No Memory: This error indicates there was not sufficient memory for allocation of data structures necessary for starting the process.

EXCEPTION HANDLING ERRORS (06)

01 Exception Handler Not Established From Program Or Process: The procedure ONEXCEPTION must be called from a program or process module, not a procedure or function.

02 Exception Handler Cannot Have Parameters: The routine identified as an exception handler must not have parameters.

03 Exception Handler Local Variables Too Large: The exception handler specified to ONEXCEPTION requires more stack space than is available in the process in which it will execute. Either increase the STACKSIZE concurrent characteristic or decrease the size of the local variables of the exception handler.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

MEMORY MANAGEMENT ERRORS (07)

01 Invalid Heap: This error should only occur if the integrity of the user's system heap is accidently destroyed either by run-time support code or by the user's code.

02 Heap Overflow: This error indicates that the available heap space has been exhausted.

03 Heap Packet Error: This error occurs when a heap packet is passed to a routine such as DISPOSE and the heap packet is invalid.

04 Invalid Packet Error: This error occurs when an attempt is made to free a packet that is no longer allocated.

FILE I/O ERRORS (08)

The following errors pertain to (Pascal) file management.

01 File Is Not Open For Reading: This error occurs when an attempt was made to read from a file which was not open for reading. A file is opened for reading by using RESET.

02 File Is Not Open For Writing: This error occurs when an attempt is made to write to a file which was not open for writing. A file is opened for writing by using REWRITE.

03 Sequential Read Past End Of File: This error occurs when an attempt is made to read past the end of a sequential file.

04 Open Error: This error occurs when an attempt to open a file fails.

05 Read Error: This error occurs when a read operation fails.

06 Write Error: This error occurs when a write operation fails.

07 No Memory For File Descriptor: This error occurs when there is not sufficient memory space with which to allocate a file descriptor.

08 No Memory For Pathname: This error occurs when system memory cannot be obtained in which to save the pathname associated with a file variable.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

09 File Not Closed: This error occurs when a file routine, which must only operate on a closed file (F$STLENGTH for example) is passed an open file variable.

10 Invalid Parameter Passed To F$STLENGTH: This error occurs if F$STLENGTH was called with a component length less than one.

11 Not A Text File: This error occurs when a text file operation is attempted on a sequential or random file. ( For example, F$STLENGTH can only operate on text files.)

TEXT I/O ERRORS (09)

01 Text Conversion; Parameter Out Of Range: This error occurs when a parameter to an encode or decode routine is out of range. For example, the index parameter must be a positive integer.

02 Text Conversion; Field Width Too Large: This error occurs when a field width in a write statement is larger than the logical record length of the file.

03 Text Conversion; Incomplete Data: This error occurs when a data value read or decoded is syntactically incomplete, for example, the value "1.0E" given for a real number.

04 Text Conversion; Invalid Character In Text Field: This error occurs when a field being read contains a character which is invalid for the particular data type, for example, the character "." when reading an integer value.

05 Text Conversion; Value Too Large: This error occurs when some data value being read is too large to be represented as the particular data type, for example, attempting to read "32768" as an integer value.

06 Text Read Past End Of File: This error occurs when an attempt is made to read past the end of a text file.

07 Text Field Exceeds Record Size: This error occurs when a specified field width is greater than the logical record size of the file.

TABLE B-2. EXPLANATION OF ERROR REASON (yy) CODES.

CHANNEL ERRORS (10)

01 No Memory For Buffers: This error occurs when there is no memory available for allocating message buffers in calls to C$ALLOCATE.

02 No Memory For Semaphores: This error occurs when there is no memory available for allocating channel synchronization semaphores. This error may be generated from calls to C$INIT or C$ALLOCATE.

03 No Memory For Channels: This error occurs when there is no memory available for allocating channel records in calls to C$INIT.

FILE I/O DECODER ERRORS (11)

These errors are returned by the I/O decoder routines. Complete information on I/O decoder errors can be found in the Device Independent File I/O (DIF) User's Manual.

01 Empty File Identifier List: The list of all file identifiers allocated in a process was empty when D$FIDRELEASE was called to deallocate a file identifier record. The probable cause is that D$DISCONNECT has been called to deallocate a file identifier that has already been deallocated.

02 File Identifier Not Found: D4FIDRELEASE was unable to find a file identifier in the list of all file identifier of the process in which it was allocated. The probable cause is that D$DISCONNECT has been called to deallocate a file identifier that has already been deallocated.

03 File Identifier Not Released: Routine D$TERM has been called during process termination to close and disconnect all files that are active; for some file the associated file identifier was not deallocated. The probable cause is an I/O subsystem whose disconnect entry did not call D$FIDRELEASE.

INTERPROCESS COMMUNICATION ERRORS (12)

01 No Heap For Pathname Record: This error occurs in IPC$CONNECT when there is no memory available for allocating pathname records.

TABLE B-2. EXPLANATION OF ERROR REASON (yy).

| | |
|---|---|
| 02 | No Heap For Name field: This error occurs in IPC$CONNECT when there is no memory available for allocating the name field found in the channel record. |
| 03 | No Heap For File Variable Record: This error occurs in IPC$CONNECT when there is no memory available for allocating the file variable record. |
| 04 | No Heap For Port Variables: This error occurs in IPC$INIT when there is no memory available for allocating the port variable record. |

# APPENDIX C

# Rx ROUTINE TEMPLATES

## C.1 GENERAL

This section describes the Rx linkage conventions, and contains templates for Rx processes, standard procedures and functions, and optimized procedures and functions. Items in upper case must be specified as shown; items in lower case must be specified by the user (i.e. the user must choose names for labels, and values for stack and heap sizes, etc.). Lines starting with asterisks ("*") are comments and need not be included in the source. Parts of the templates are numbered, and these numbers refer to notes in paragraph C.7.

## C.2 TEMPLATE FOR A PROCESS

The following template specifies the format of an Rx process.

```
(1) ---> IDT  'proces'
*************************************************************
*                    -- MODULES DEFINED --                  *
*************************************************************
(2) ---> DEF  proces
*************************************************************
*                  -- EXTERNAL REFERENCES --                *
*************************************************************
        |REF  CALL$
(3) ---> |REF  S$PRCS
        |REF  E$PRCS
        |REF  EXIT$P
         REF  ...       OTHER SYSTEM AND
         REF  ...       USER DEFINED
         REF  ...       SYMBOLS REFERENCED
*************************************************************
*                       -- REGISTERS --                     *
*************************************************************
|PR     EQU  7          REGISTER 7 POINTS TO PROCESS RECORD
(4) |CODE   EQU  8          REGISTER 8 POINTS TO CODE BASE
|LF     EQU  9          REGISTER 9 POINTS TO LOCAL FRAME
|SP     EQU  10         REGISTER 10 IS STACK POINTER
```

```
*****************************************************************
*                    -- PROCESS DESCRIPTOR   --                 *
*****************************************************************
(5)  PSEG
(6) proces EQU  $
(7)        DATA prolog-proces       OFFSET TO FIRST STATEMENT
(8)        DATA epilog-proces       OFFSET TO TERMINATION CODE
(9)        DATA 0                   ZERO FOR PROCESSES
(10)       DATA >nnnn               SIZE OF ARGUMENTS (BYTES)
(12) frmsiz DATA >nnnn              TOTAL FRAME SIZE (BYTES)
(13) lexlvl DATA >nnnn              LEXICAL LEVEL
(14) priori DATA >nnnn              PROCESS PRIORITY
(15) stksiz DATA >nnnn              PROCESS STACKSIZE (WORDS)
(16) hpsize DATA >nnnn              PROCESS HEAPSIZE (WORDS)
*****************************************************************
*                        -- USER DATA --                        *
*****************************************************************
(17)       ...
           ...        any user defined constants appear here
           ...
*****************************************************************
*                    -- PROCESS START CODE --                   *
*****************************************************************
(18) prolog EQU  $
           MOV  @frmsiz-proces(CODE),*SP+ PUSH FRAME SIZE, BYTES
           MOV  @lexlvl-proces(CODE),*SP+ PUSH LEXICAL LEVEL
(19)       MOV  @priori-proces(CODE),*SP+ PUSH PRIORITY
           MOV  @stksiz-proces(CODE),*SP+ PUSH STACKSIZE, WORDS
           MOV  @hpsize-proces(CODE),*SP+ PUSH HEAPSIZE, WORDS
           DATA CALL$,S$PRCS              START THIS PROCESS
*****************************************************************
*                        -- USER CODE --                        *
*****************************************************************
           ...
           ...                        Sample references to arguments
           ...
(20)       MOV  *LF,@<ga>             GET 1ST ARGUMENT
           MOV  @2(LF),@<ga>          GET 2ND ARGUMENT
           ...
           ...                        A typical process call:
           ...
(21)       MOV  R1,*SP+               PASS 1ST ARGUMENT
           MOV  R2,*SP+               PASS 2ND ARGUMENT
           DATA CALL$,routin          CALL THE PROCEDURE
           ...
           ...                        'routin' must appear above
           ...                        in a REF statement.
           ...
```

```
*************************************************************
*                -- PROCESS TERMINATION CODE --               *
*************************************************************
(23) epilog EQU  $
       (  MOV  @lexlvl-proces(CODE),*SP+  PUSH LEXICAL LEVEL
(24)   (  DATA CALL$,E$PRCS               TERMINATE THIS PRCS
       (  B    @EXIT$P                    EXIT
(29)    END
```

## C.3 TEMPLATE FOR A STANDARD PROCEDURE

The following template defines the format for a standard Rx procedure.

```
(1)     IDT  'proced'
*************************************************************
*                  -- MODULES DEFINED --                    *
*************************************************************
(2)     DEF  proced
*************************************************************
*                 -- EXTERNAL REFERENCES --                 *
*************************************************************
       (REF  CALL$
       (REF  S$PRCS
(3)    (REF  E$PRCS
       (REF  EXIT$P
        REF  ...              OTHER SYSTEM AND
        REF  ...              USER DEFINED
        REF  ...              SYMBOLS REFERENCED
*************************************************************
*                      -- REGISTERS --                      *
*************************************************************
(4)  (PR     EQU  7           R7 POINTS TO PROCESS RECORD
     (CODE   EQU  8           R8 POINTS TO CODE BASE
     (LF     EQU  9           R9 POINTS TO LOCAL FRAME
     (SP     EQU  10          R10 IS THE STACK POINTER
*
*************************************************************
*                -- PROCEDURE DESCRIPTOR  --                *
*************************************************************
(5)  ->  PSEG
(6)  -> proced EQU  $
(7)  ->  DATA prolog-proced     OFFSET TO FIRST STATEMENT
(8)  ->  DATA epilog-proced     OFFSET TO TERMINATION CODE
(11) ->  DATA >nnnn             SIZE OF LOCAL VARS (BYTES)
(12) ->  DATA >nnnn             LOCAL FRAME SIZE (BYTES)
*************************************************************
*                      -- USER DATA --                      *
*************************************************************
(17) ->  (...
         (...       any user defined constants appear here
         (...
```

```
*****************************************************************
*                        -- USER CODE --                        *
*****************************************************************
(18)→prolog EQU  $
          ...
          ...                          Sample references to arguments
          ...
(20)→     MOV  *LF,@<ga>               GET 1ST ARGUMENT
          MOV  @2(LF),@<ga>            GET 2ND ARGUMENT
          ...
          ...                          A typical procedure call:
          ...
          MOV  R1,*SP+                 PASS 1ST ARGUMENT
(21)→     MOV  R2,*SP+                 PASS 2ND ARGUMENT
          DATA CALL$,routin            CALL THE PROCEDURE
          ...
          ...                          'routin' must appear above
          ...                          in a REF statement.
          ...
*****************************************************************
*              -- PROCEDURE TERMINATION CODE --                 *
*****************************************************************
(23)→epilog EQU  $
(25)→     B    @EXIT$P                 EXIT
(29)→     END
```

## C.4 TEMPLATE FOR A STANDARD FUNCTION

The following template defines the format of a standard Rx function.

```
(1)→      IDT  'functi'
*****************************************************************
*                    -- MODULES DEFINED --                      *
*****************************************************************
(2)→      DEF  functi
*****************************************************************
*                  -- EXTERNAL REFERENCES --                    *
*****************************************************************
          REF  CALL$
(3)→      REF  S$PRCS
          REF  E$PRCS
          REF  EXIT$n                 (EXIT$1 OR EXIT$2)
          REF  ...                    OTHER SYSTEM AND
          REF  ...                    USER DEFINED
          REF  ...                    SYMBOLS REFERENCED
*****************************************************************
*                       -- REGISTERS --                         *
*****************************************************************
PR        EQU  7                      R7 POINTS TO PROCESS RECORD
(4)→CODE  EQU  8                      R8 POINTS TO CODE BASE
LF        EQU  9                      R9 POINTS TO LOCAL FRAME
SP        EQU  10                     R10 IS THE STACK POINTER
```

```
*****************************************************************
*                  -- PROCESS DESCRIPTOR   --                   *
*****************************************************************
(5)  ---->  PSEG
(6)  functi EQU  $
(7)  ---->  DATA prolog-functi       OFFSET TO FIRST STATEMENT
(8)  ---->  DATA epilog-functi       OFFSET TO TERMINATION CODE
(11) ---->  DATA >nnnn               SIZE OF LOCAL VARS (BYTES)
(12) ---->  DATA >nnnn               LOCAL FRAME SIZE (BYTES)
*****************************************************************
*                       -- USER DATA --                         *
*****************************************************************
(17) ---->  ...
            ...        any user defined constants appear here
            ...
*****************************************************************
*                       -- USER CODE --                         *
*****************************************************************
(18) prolog EQU  $
            ...
            ...                      Sample references to arguments
            ...
(20) ---->  MOV  *LF,@<ga>           GET 1ST ARGUMENT
            MOV  @2(LF),@<ga>        GET 2ND ARGUMENT
            ...
            ...                      A typical function call:
            ...
            MOV  R1,*SP+             PASS 1ST ARGUMENT
            MOV  R2,*SP+             PASS 2ND ARGUMENT
(22) ---->  MOV  R3,*SP+             PASS 3RD ARGUMENT
            DATA CALL$,functi        CALL THE FUNCTION
            MOV  *SP,@RESULT         SP POINTS TO RESULT (R10)
            ...
            ...                      'functi' must appear above
            ...                      in a REF statement.
            ...
*****************************************************************
*              -- FUNCTION TERMINATION CODE --                  *
*****************************************************************
(23) epilog EQU  $
(26) ---->  B    @EXIT$n              RETURN RESULT OF LENGTH n
            DATA >mmmm               OFFSET INTO LOCAL STORAGE
(29) ---->  END                        OF THE DESIRED RESULT
```

## C.5 TEMPLATE FOR AN OPTIMIZED PROCEDURE

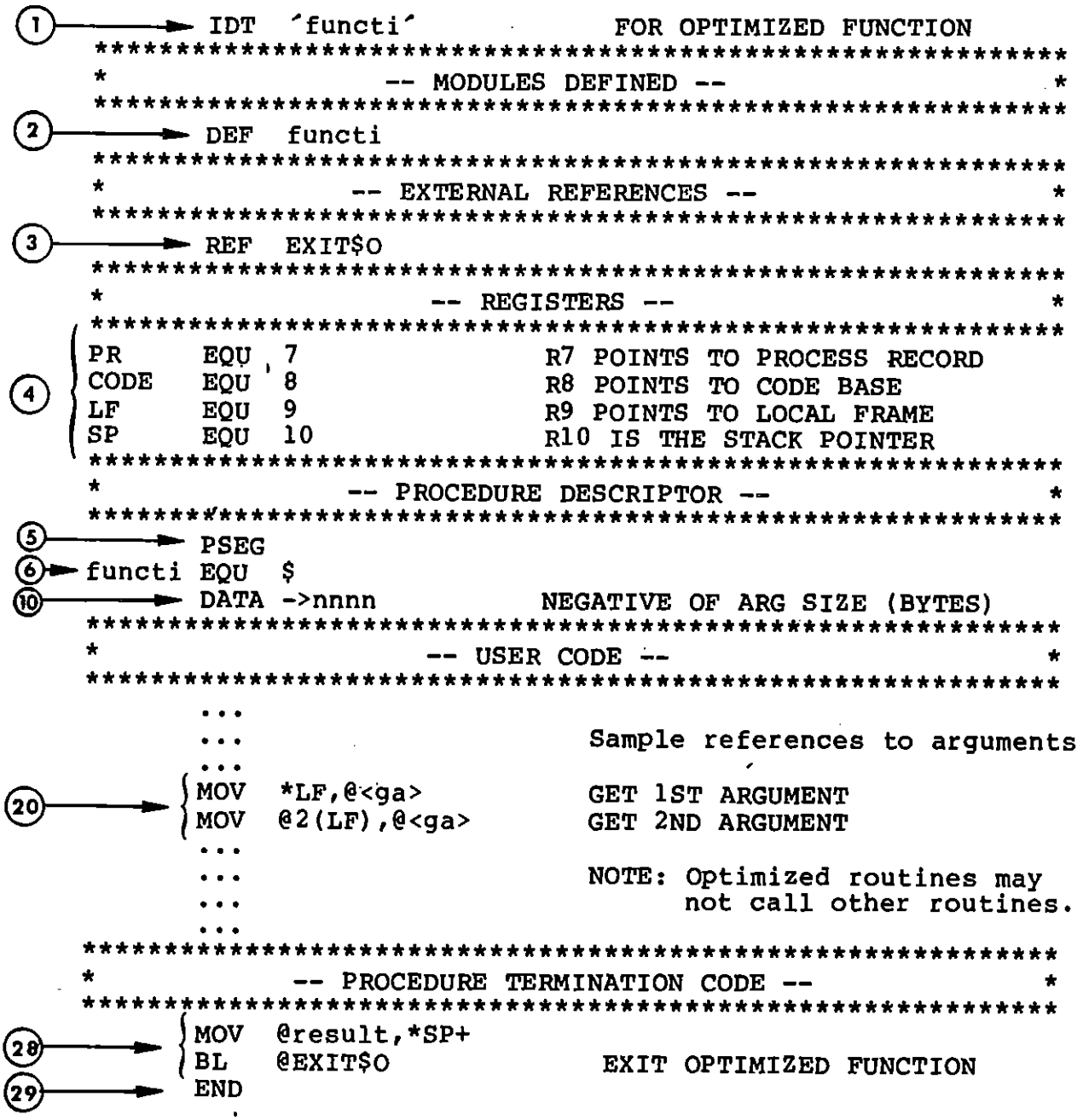
The following template defines the format of an optimized Rx routine linkage (sometimes called a "special" linkage). A negative or zero value in the first word of the routine indicates that the routine uses an optimized linkage. A new workspace is allocated, and registers R7, R8, R9, and R10 are initialized, but no new stack frame is allocated. Instead, the routine uses the caller's stack. The caller's stack pointer is reset, by adding the first word in the routine to the

caller's R10. An optimized procedure may not call any other procedures or functions.

```
(1)  IDT  'proced'              FOR OPTIMIZED PROCEDURE
*****************************************************
*                    -- MODULES DEFINED --                *
*****************************************************
(2)  DEF  proced
*****************************************************
*                     -- REGISTERS --                     *
*****************************************************
PR      EQU  7                  R7 POINTS TO PROCESS RECORD
(4) CODE    EQU  8              R8 POINTS TO CODE BASE
LF      EQU  9                  R9 POINTS TO LOCAL FRAME
SP      EQU  10                 R10 IS THE STACK POINTER
*****************************************************
*                  -- PROCEDURE DESCRIPTOR --             *
*****************************************************
(5)  PSEG
(6) proced EQU  $
(10) DATA ->nnnn          NEGATIVE OF ARG SIZE (BYTES)
*****************************************************
*                     -- USER CODE --                     *
*****************************************************
       ...
       ...                      Sample references to arguments
       ...
(20)   MOV  *LF,@<ga>           GET 1ST ARGUMENT
       MOV  @2(LF),@<ga>        GET 2ND ARGUMENT
       ...
       ...                      NOTE: Optimized routines may
       ...                            not call other routines.
       ...
*****************************************************
*              -- PROCEDURE TERMINATION CODE --           *
*****************************************************
(27) RTWP                       RETURN TO CALLER
(29) END
```

## C.6 TEMPLATE FOR AN OPTIMIZED FUNCTION

The following template specifies the format of an optimized function. An optimized function may only return a single word result (a standard function may return a one, two, or four word result). The restrictions which apply to an optimized procedure also apply to an optimized function.


1
IDT 'functi' FOR OPTIMIZED FUNCTION
-- MODULES DEFINED --
2
DEF functi
-- EXTERNAL REFERENCES --
3
REF EXIT$O
-- REGISTERS --
4
PR EQU 7 R7 POINTS TO PROCESS RECORD
CODE EQU 8 R8 POINTS TO CODE BASE
LF EQU 9 R9 POINTS TO LOCAL FRAME
SP EQU 10 R10 IS THE STACK POINTER
-- PROCEDURE DESCRIPTOR --
5
PSEG
6
functi EQU $
10
DATA ->nnnn NEGATIVE OF ARG SIZE (BYTES)
-- USER CODE --
Sample references to arguments
20
MOV *LF,@<ga> GET 1ST ARGUMENT
MOV @2(LF),@<ga> GET 2ND ARGUMENT
NOTE: Optimized routines may
not call other routines.
-- PROCEDURE TERMINATION CODE --
28
MOV @result,*SP+
BL @EXIT$O EXIT OPTIMIZED FUNCTION
29
END

## C.7 NOTES

1) An IDT statement which gives the module its name.

2) Each module must contain a DEF directive, which must be the same six letter (or less) label that defines the beginning of the descriptor.

3) Any external modules (user or executive) that are used by the process must be REFed as being external.

4) Register equates for the four Rx defined registers: (PR) process record pointer, (CODE) code base, (LF) local frame pointer, and (SP) stack pointer.

5) A PSEG statement is needed to declare the beginning of the program code.

6) A label which defines the beginning of the descriptor section of the routine. This label should be set to the first six characters of the routine name unless the process is at the outermost level and is therefore a system. The main routine or system should be labelled "SYSTM$".

7) An assembler defined constant which specifies the number of bytes from the beginning of the descriptor section to the beginning of the prologue section (i.e., the offset to the first executable statement).

8) An assembler defined constant which specifies the number of bytes from the beginning of the descriptor section to the beginning of the epilogue section (i.e., the offset to the termination code).

9) A zero value constant necessary for proper linkage.

10) The size in bytes of the parameters that have been passed to this process.

11) The size in bytes of the local storage required by this procedure/ function.

12) The total stackframe size in bytes. This includes the size of any local variables plus the passed parameters.

13) The lexical level is an integer which represents the level of nesting at which this process is operating. The outermost level (the SYSTEM level) is designated as level 0. Any processes that are started by the SYSTEM level would be level 1 and any started by these would be level 2, etc. Note that lexical level nesting conventions must be STRICTLY obeyed.

14) Process priority is an integer greater than 0 and less than 32767. Priorities 1 through 15 correspond to interrupt level priorities which are allocatable to a user's process. The lower the numerical value, the higher the process' urgenc-.

15) The stack size indicates the amount of memory in words that is to be allocated to this process for stack frames and workspaces of routines called within this process. The actual numerical value needed by any specific process will vary according to what routines the process calls. To start the process will generally take around 150 words in addition any other requirements that the process might have.

16) The heap size indicates the amount of memory in words that the process requires for dynamically allocated variables and stacks of nested processes. If the heap size is set to zero, the process takes any heap it needs from its parent's heap. However, a process that has no heap specifically allocated to it (when it is started), will not be able to call another process. The implication of this is that a process must have enough heap for itself and all of its lexical descendants. The system level process should have enough heap for all processes and is often specified as zero to indicate that the system level process can take all available memory as its heap.

17) The user may define any number of data words which may be used as constants by the process.

18) A label which defines the first executable statement of the routine.

19) Process prologue code required to start the process.

20) Sample references to arguments passed by caller.

21) Sample call to another routine.

22) Sample call to another function.

23) A label which defines the beginning of the epilogue code.

24) Epilogue code required to terminate a process in Rx.

25) Epilogue code required to terminate a standard procedure.

26) Epiloque code required to terminate a standard function. The value of 'n' is the number of words that the function result occupies (e.g., for an integer, 'n' would be 1; for a real number, 2); valid values are 1, 2, and 4. The data word following the branch is used by the exit handler to indicate the byte displacement into the local storage of the result to be returned.

27) Epilogue code required to terminate an optimized procedure.

28) Epilogue code required to terminate an optimized function.

29) End statement required to terminate a source module.

# APPENDIX D

## Rx SIZE BREAKDOWN

### D.1 FUNCTIONAL UNIT (MODULE NAMES) SIZE IN DECIMAL BYTES

1) Target configuration description (CONFIG).....34

2) Rx Minimum.....3034

3) Clock Routines.....720

4) Channel Routines (INIT, ALLOC, DISPOSE, SEND, WAIT RECEIVE, ACKNOWLEDGE).....844

5) RX with Clock.....4492 + 34 (CONFIG)

6) Rx with Channels.....4140 + 34 (CONFIG)

7) Rx with everything.....5410 + 34 (CONFIG)

# APPENDIX E

## RXDEMO: ASSEMBLER LISTING AND LINK MAP

The assembler listing and link map that follows is provided for use with the AMPL Debugger Walkthrough (Section VIII).

```
XDEMO     SDSMAC 3.3.0 79.312   10:05:30 THURSDAY, MAY 07, 1981.
XDEMO -- DEMONSTRATION PROGRAM FOR RX 2.0                          PAGE 0002
0002                    IDT   'RXDEMO'
0003                    DEF   RXDEMO
0004                    DEF   SYSTM$
0005          ************************************************************
0006          *                                                          *
0007          *                                                          *
0008          *                RXDEMO: RX 2.0 DEMONSTRATION              *
0009          *                                                          *
0010          *    PURPOSE:  THIS PROGRAM DEMONSTRATES THE USE OF THE    *
0011          *              RX 2.0 REAL TIME EXECUTIVE TO IMPLEMENT     *
0012          *              A PRODUCER / CONSUMER PAIR OF PROCESSES.    *
0013          *                                                          *
0014          *              TWO PROCESSES ARE STARTED, 'PRODUC' AND     *
0015          *              'CONSUM'. 'PRODUC' AUTO-BAUDS THE USER      *
0016          *              TERMINAL AND ALLOCATES CHANNEL 1. IT THEN   *
0017          *              SENDS THE BAUD RATE OF THE TERMINAL TO THE  *
0018          *              CONSUMER OVER THE CHANNEL. IT THEN READS    *
0019          *              CHARACTERS FROM THE TERMINAL AND SEND THEM  *
0020          *              TO THE CONSUMER. THE CONSUMER PROCESS       *
0021          *              PRINTS OUT MESSAGES ABOUT THE CHARACTERS    *
0022          *              IT HAS BEEN SENT. WHEN THE PRODUCER READS   *
0023          *              AND SENDS A 'Z', THE PRODUCER TERMINATES.   *
0024          *              WHEN THE CONSUMER RECEIVES A 'Z', IT TERMI- *
0025          *              NATES.                                      *
0026          *                                                          *
0027          *    CALLS:    PRODUC, CONSUM                            *
0028          *                                                          *
0029          *    NOTES:    TERMINAL I/O IS DONE THROUGH A SET OF FOUR *
0030          *              SIMPLE WAIT LOOP DRIVEN I/O ROUTINES. THESE *
0031          *              CANNOT BE USED IN AN INTERRUPT ENVIRONMENT; *
0032          *              FOR SUCH PURPOSES, THE 'DEVICE INDEPENDENT  *
0033          *              FILE I/O' COMPONENT PACKAGE IS AVAILABLE.   *
0034          *                                                          *
              ************************************************************
```

```
RXDEMO     SDSMAC 3.3.0 79.312   10:05:30 THURSDAY, MAY 07, 1981
RXDEMO -- DEMONSTRATION PROGRAM FOR RX 2.0                    PAGE 0003
0036                ***************************************************
0037                *                -- EXTERNAL REFERENCES --          *
0038                ***************************************************
0039                       REF  CALL$
0040                       REF  S$PRCS
0041                       REF  E$PRCS
0042                       REF  EXIT$P
0043                       REF  PRODUC
0044                       REF  CONSUM
0045                ***************************************************
0046                *                    -- REGISTERS --                *
0047                ***************************************************
0048       0007  PR     EQU  7         REGISTER 7 POINTS TO PROCESS RECORD
0049       0008  CODE   EQU  8         REGISTER 8 POINTS TO CODE BASE
0050       0009  LF     EQU  9         REGISTER 9 POINTS TO LOCAL FRAME
0051       000A  SP     EQU  10        REGISTER 10 IS STACK POINTER
0052                *
0053                ***************************************************
0054                *                -- PROCESS DESCRIPTOR  --          *
0055                ***************************************************
0056 0000                  PSEG
0057       0000' SYSTM$ EQU  $                    REQUIRED IN MAIN PROCESS
0058       0000' RXDEMO EQU  $                    NAME OF MODULE
0059 0000 0012             DATA PROLOG-RXDEMO     OFFSET TO FIRST STATEMENT
0060 0002 0032             DATA EPILOG-RXDEMO     OFFSET TO TERMINATION CODE
0061 0004 0000             DATA 0                 ZERO FOR PROCESSES
0062 0006 0000             DATA >0000             SIZE OF ARGUMENTS (BYTES)
0063 0008 0000  FRMSIZ     DATA >0000             TOTAL FRAME SIZE (BYTES)
0064 000A 0001  LEXLVL     DATA >0001             LEXICAL LEVEL
0065 000C 0100  PRIORI     DATA >0100             PROCESS PRIORITY
0066 000E 0100  STKSIZ     DATA >0100             PROCESS STACKSIZE (WORDS)
0067 0010 0000  HPSIZE     DATA >0000             PROCESS HEAPSIZE (WORDS)
0068                ***************************************************
0069                *                    -- USER DATA --                *
0070                ***************************************************
0071                *
0072                * NONE
0073                *
```

```
RXDEMO     SDSMAC 3.3.0 79.312   10:05:30 THURSDAY, MAY 07, 1981.
RXDEMO -- DEMONSTRATION PROGRAM FOR RX 2.0                    PAGE 0004
 0075              ************************************************************
 0076              *                  -- PROCESS START CODE --                *
 0077              ************************************************************
 0078      0012'  PROLOG EQU  $
 0079  0012 CEA8         MOV  @FRMSIZ-RXDEMO(CODE),*SP+ PUSH FRAME SIZE, BYTES
       0014 0008
 0080  0016 CEA8         MOV  @LEXLVL-RXDEMO(CODE),*SP+ PUSH LEXICAL LEVEL
       0018 000A
 0081  001A CEA8         MOV  @PRIORI-RXDEMO(CODE),*SP+ PUSH PRIORITY
       001C 000C
 0082  001E CEA8         MOV  @STKSIZ-RXDEMO(CODE),*SP+ PUSH STACKSIZE, WORDS
       0020 000E
 0083  0022 CEA8         MOV  @HPSIZE-RXDEMO(CODE),*SP+ PUSH HEAPSIZE, WORDS
       0024 0010
 0084  0026 0000         DATA CALL$,S$PRCS              START THIS PROCESS
       0028 0000
 0085              ************************************************************
 0086              *                       -- USER CODE --                    *
 0087              ************************************************************
 0088              *
 0089  002A 0026'         DATA CALL$,PRODUC               START PRODUCER
       002C 0000
 0090  002E 002A'         DATA CALL$,CONSUM               START CONSUMER
       0030 0000
 0091              *
 0092              ************************************************************
 0093              *               -- PROCESS TERMINATION CODE --               *
 0094              ************************************************************
 0095      0032'  EPILOG EQU  $
 0096  0032 CEA8         MOV  @LEXLVL-RXDEMO(CODE),*SP+  PUSH LEXICAL LEVEL
       0034 000A
 0097  0036 002E'         DATA CALL$,E$PRCS              TERMINATE THIS PRCS
       0038 0000
 0098  003A 0460          B    @EXIT$P                   EXIT
       003C 0000
 0099                     END
O ERRORS,     NO WARNINGS
```

```
PRODUC     SDSMAC 3.3.0 79.312   10:08:10 THURSDAY, MAY 07, 1981.
PRODUC -- PRODUCER PROCESS FOR DEMONSTRATION                    PAGE 000
 0002                    IDT  'PRODUC'
 0003              ********************************************************
 0004              *
 0005              *                PRODUC: THE PRODUCER PROCESS
 0006              *
 0007              *   PURPOSE: THIS PROCESS FIRST CALLS TI$SET TO FIND THE
 0008              *            BAUD RATE OF THE USER TERMINAL, AND THEN
 0009              *            ALLOCATES CHANNEL 1. THE BAUD RATE FLAG FROM
 0010              *            TI$SET IS SENT TO THE CONSUMER PROCESS OVER
 0011              *            THE CHANNEL (THE CONSUMER NEEDS TO KNOW THE
 0012              *            BAUD RATE OF THE TERMINAL SO THAT IT CAN ALSO
 0013              *            PERFORM I/O).
 0014              *
 0015              *            THEN PRODUCER THEN READS CHARACTERS FROM THE
 0016              *            TERMINAL AND SENDS THEM TO THE CONSUMER OVER
 0017              *            THE CHANNEL. WHEN THE PRODUCER RECEIVES A
 0018              *            'Z', IT SENDS THE MESSAGE, WAITS FOR ACKNOW-
 0019              *            LEDGEMENT, AND TERMINATES. THE CONSUMER WILL
 0020              *            ALSO TERMINATE WHEN IT RECEIVES A 'Z'.
 0021              *
 0022              *   CALLS:   TI$SET, TI$CIN, TI$COT, TI$MSG
 0023              *
 0024              ********************************************************
 0025                     DEF  PRODUC
 0026              ********************************************************
 0027              *                -- EXTERNAL REFERENCES --             *
 0028              ********************************************************
 0029                     REF  CALL$    CALL SYMBOL
 0030                     REF  S$PRCS   START PROCESS
 0031                     REF  E$PRCS   END PROCESS
 0032                     REF  EXIT$P   EXIT
 0033                     REF  C$INIT   ALLOCATE A CHANNEL
 0034                     REF  C$TERM   FREE A CHANNEL
 0035                     REF  C$ALLO   ALLOCATE A MESSAGE BUFFER
 0036                     REF  C$SEND   SEND A MESSAGE
 0037                     REF  C$WAIT   WAIT FOR ACKNOWLEDGEMENT
 0038                     REF  TI$SET   SET BAUD RATE
 0039                     REF  TI$CIN   READ CHARACTER
 0040                     REF  TI$COT   WRITE CHARACTER
 0041                     REF  TI$MSG   WRITE STRING
```

```
PRODUC     SDSMAC 3.3.0 79.312   10:08:10 THURSDAY, MAY 07, 1981.
PRODUC -- PRODUCER PROCESS FOR DEMONSTRATION                    PAGE 0003
0043                ***************************************************
0044                *                    -- REGISTERS --                  *
0045                ***************************************************
0046       0001     MP      EQU  1       POINTER TO MESSAGE BUFFER
0047       0002     TEMP    EQU  2       USED TO CALCULATE ADDRESS OF VARIABLES
0048                *
0049       0007     PR      EQU  7       REGISTER 7 POINTS TO PROCESS RECORD
0050       0008     CODE    EQU  8       REGISTER 8 POINTS TO CODE BASE
0051       0009     LF      EQU  9       REGISTER 9 POINTS TO LOCAL FRAME
0052       000A     SP      EQU  10      REGISTER 10 IS STACK POINTER
0053                ***************************************************
0054                *              -- MAPPING FOR LOCAL FRAME ---         *
0055                ***************************************************
0056 0000                   DORG 0       OFFSET FROM LF (R9)
0057                *
0058 0000           BAUD    BSS  2       BAUD RATE FLAG FROM TI$SET
0059 0002           CHNLID  BSS  2       CHANNEL ID (PTR TO CHANNEL)
0060 0004           MSGBUF  BSS  2       POINTER TO MESSAGE BUFFER
0061       0006     LFLEN   EQU  $
0062                *
0063                ***************************************************
0064                *           -- MAPPING FOR MESSAGE BUFFER --          *
0065                ***************************************************
0066 0000                   DORG 0       OFFSET FROM MSGBUF
0067                *
0068 0000           CSLOT   BSS  2       LOW ORDER BYTE IS ONE CHARACTER MESSAGE
0069 0002           BSLOT   BSS  2       ONE WORD FOR BAUD RATE FLAG
0070       0004     MBLEN   EQU  $
0071                *
0072                ***************************************************
0073                *                -- PROCESS DESCRIPTOR  --            *
0074                ***************************************************
0075 0000                   PSEG
0076       0000     PRODUC  EQU  $
0077 0000 00A4              DATA PROLOG-PRODUC      OFFSET TO FIRST STATEMENT
0078 0002 01BE              DATA EPILOG-PRODUC      OFFSET TO TERMINATION CODE
0079 0004 0000              DATA 0                  ZERO FOR PROCESSES
0080 0006 0000              DATA >0000              SIZE OF ARGUMENTS (BYTES)
0081 0008 0006     FRMSIZ   DATA LFLEN              TOTAL FRAME SIZE (BYTES)
0082 000A 0001     LEXLVL   DATA >0001              LEXICAL LEVEL
0083 000C 0100     PRIORI   DATA >0100              PROCESS PRIORITY
0084 000E 0100     STKSIZ   DATA >0100              PROCESS STACKSIZE (WORDS)
0085 0010 0000     HPSIZE   DATA >0000              PROCESS HEAPSIZE (WORDS)
```

```
PRODUC      SDSMAC 3.3.0 79.312   10:08:10 THURSDAY, MAY 07, 1981.
PRODUC -- PRODUCER PROCESS FOR DEMONSTRATION                    PAGE 0004
0087                ***************************************************
0088                *                    -- USER DATA --                  *
0089                ***************************************************
0090        0012    CHNAME EQU  $-PRODUC
0091 0012  0001            DATA 1                  USE CHANNEL 1
0092        0014    MSGLEN EQU  $-PRODUC
0093 0014  0004            DATA MBLEN              LENGTH OF MESSAGE BUFFER
0094        0016    PORT01 EQU  $-PRODUC
0095 0016  0080            DATA >0080              CRU BASE OF PORT NUMBER 1
0096        0018    Z      EQU  $-PRODUC
0097 0018  005A            DATA >5A                Z IN ASCII
0098                *
0099        001A    HELLO  EQU  $-PRODUC
0100 001A    0D            BYTE >0D,>0A,>0A        <CR><LF><LF>
     001B    0A
     001C    0A
0101 001D    52            TEXT 'RX 2.0 DEMONSTRATION PROGRAM'
0102 0039    0D            BYTE >0D,>0A,>0A        <CR><LF><LF>
     003A    0A
     003B    0A
0103 003C    00            BYTE 0                  <NUL> TO TERMINATE STRING
0104                *
0105        003D    SENDMS EQU  $-PRODUC
0106 003D    50            TEXT 'PRODUCER SENDS '
0107 004C    00            BYTE 0
0108                *
0109        004D    ACKMSG EQU  $-PRODUC
0110 004D    50            TEXT 'PRODUCER RECEIVES ACKNOWLEDGEMENT OF '
0111 0072    00            BYTE 0
0112                *
0113        0073    GOODBY EQU  $-PRODUC
0114 0073    50            TEXT 'PRODUCER TERMINATES'
0115 0086    0D            BYTE >0D,>0A,>0A        <CR><LF><LF>
     0087    0A
     0088    0A
0116 0089    45            TEXT 'END OF DEMONSTRATION'
0117 009D    0D            BYTE >0D,>0A,>0A        <CR><LF><LF>
     009E    0A
     009F    0A
0118 00A0    00            BYTE 0                  <NUL> TO TERMINATE STRING
0119                *
0120        00A1    CRLF   EQU  $-PRODUC
0121 00A1    0D            BYTE >0D,>0A,>00        <CR><LF><NUL>
     00A2    0A
     00A3    00
0122 00A4                  EVEN                    CODE MUST START ON WORD BNDRY
```

```
0124                ****************************************************
0125                *                  -- PROCESS START CODE --                *
0126                ****************************************************
0127        00A4'   PROLOG EQU  $
0128  00A4  CEA8          MOV   @FRMSIZ-PRODUC(CODE),*SP+ PUSH FRAME SIZE, BYTES
      00A6  0008
0129  00A8  CEA8          MOV   @LEXLVL-PRODUC(CODE),*SP+ PUSH LEXICAL LEVEL
      00AA  000A
0130  00AC  CEA8          MOV   @PRIORI-PRODUC(CODE),*SP+ PUSH PRIORITY
      00AE  000C
0131  00B0  CEA8          MOV   @STKSIZ-PRODUC(CODE),*SP+ PUSH STACKSIZE, WORDS
      00B2  000E
0132  00B4  CEA8          MOV   @HPSIZE-PRODUC(CODE),*SP+ PUSH HEAPSIZE, WORDS
      00B6  0010
0133  00B8  0000          DATA CALL$,S$PRCS               START THIS PROCESS
      00BA  0000
0134                ****************************************************
0135                * SET BAUD RATE FLAG 'BAUD'                          *
0136                ****************************************************
0137  00BC  CEA8          MOV   @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      00BE  0016
0138  00C0  C089          MOV   LF,TEMP                 PUSH PTR TO BAUD RATE FLAG
0139  00C2  0222          AI    TEMP,BAUD                : (LOCAL FRAME+OFFSET)
      00C4  0000
0140  00C6  CE82          MOV   TEMP,*SP+                :
0141  00C8  00B8'         DATA CALL$,TI$SET             AUTO-BAUD THE TERMINAL
      00CA  0000
0142                ****************************************************
0143                * WRITE 'RX 2.0 DEMONSTRATION PROGRAM#0D#0A#0A#0A'   *
0144                ****************************************************
0145  00CC  CEA8          MOV   @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      00CE  0016
0146  00D0  CEA9          MOV   @BAUD(LF),*SP+          PUSH BAUD RATE FLAG
      00D2  0000
0147  00D4  C088          MOV   CODE,TEMP               PUSH PTR TO MESSAGE
0148  00D6  0222          AI    TEMP,HELLO               : (CODE BASE+OFFSET)
      00D8  001A
0149  00DA  CE82          MOV   TEMP,*SP+                :
0150  00DC  00C8'         DATA CALL$,TI$MSG             WRITE THE STRING
      00DE  0000
0151                ****************************************************
0152                * ALLOCATE CHANNEL 'CHNAME', PUT RESULT IN 'CHNLID'   *
0153                ****************************************************
0154  00E0  CEA8          MOV   @CHNAME(CODE),*SP+ PUSH NAME OF CHANNEL (1)
      00E2  0012
0155  00E4  C089          MOV   LF,TEMP                 PUSH PTR TO CHANNEL ID
0156  00E6  0222          AI    TEMP,CHNLID              : (LOCAL FRAME+OFFSET)
      00E8  0002
0157  00EA  CE82          MOV   TEMP,*SP+                :
0158  00EC  00DC'         DATA CALL$,C$INIT             ALLOCATE THE CHANNEL
      00EE  0000
```

```
PRODUC     SDSMAC 3.3.0 79.312   10:08:10 THURSDAY, MAY 07, 1981.
PRODUC -- PRODUCER PROCESS FOR DEMONSTRATION                    PAGE 000
 0160               ************************************************************
 0161               * ALLOCATE MESSAGE BUFFER 'MSGBUF'
 0162               ************************************************************
 0163 00F0 CEA8           MOV  @MSGLEN(CODE),*SP+ PUSH LENGTH OF BUFFER
      00F2 0014
 0164 00F4 C089           MOV  LF,TEMP            PUSH PTR TO MSGBUF
 0165 00F6 0222           AI   TEMP,MSGBUF         : (LOCAL FRAME+OFFSET)
      00F8 0004
 0166 00FA CE82           MOV  TEMP,*SP+           :
 0167 00FC 00EC'          DATA CALL$,C$ALLO
      00FE 0000
 0168               ************************************************************
 0169               * @(BSLOT+MSGBUF) := 'BAUD'
 0170               ************************************************************
 0171 0100 C069           MOV  @MSGBUF(LF),MP    MOVE VALUE TO REGISTER MP
      0102 0004
 0172 0104 C869           MOV  @BAUD(LF),@BSLOT(MP) PUT BAUD RATE INTO MESSAGE
      0106 0000
      0108 0002
 0173               ************************************************************
 0174               * UNTIL 'CHAR' EQ 'Z' READ CHARACTER 'CHAR'
 0175               ************************************************************
 0176 010A CEA8   SENDLP MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      010C 0016
 0177 010E C081           MOV  MP,TEMP            PUSH PTR TO CHARACTER
 0178 0110 0222           AI   TEMP,CSLOT          : (MSG BUF PTR+OFFSET)
      0112 0000
 0179 0114 CE82           MOV  TEMP,*SP+           :
 0180 0116 00FC'          DATA CALL$,TI$CIN       READ A CHARACTER
      0118 0000
 0181               ************************************************************
 0182               *   SEND 'MSGBUF' OVER 'CHNLID'
 0183               ************************************************************
 0184 011A CEA9           MOV  @CHNLID(LF),*SP+   PUSH CHANNEL ID
      011C 0002
 0185 011E CE81           MOV  MP,*SP+            PUSH MSG BUFFER PTR
 0186 0120 0116'          DATA CALL$,C$SEND       SEND MESSAGE
      0122 0000
```

```
0188                ***************************************************
0189                * WRITE 'PRODUCER SENDS c'                        *
0190                ***************************************************
0191 0124 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     0126 0016
0192 0128 CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     012A 0000
0193 012C C088            MOV  CODE,TEMP          PUSH PTR TO MESSAGE
0194 012E 0222            AI   TEMP,SENDMS         : (CODE BASE+OFFSET)
     0130 003D
0195 0132 CE82            MOV  TEMP,*SP+           :
0196 0134 0120'           DATA CALL$,TI$MSG       WRITE THE STRING
     0136 00DE'
0197                *
0198 0138 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     013A 0016
0199 013C CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     013E 0000
0200 0140 CEA1            MOV  @CSLOT(MP),*SP+    PUSH THE CHARACTER
     0142 0000
0201 0144 0134'           DATA CALL$,TI$COT       WRITE THE CHARACTER
     0146 0000
0202                *
0203 0148 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     014A 0016
0204 014C CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     014E 0000
0205 0150 C088            MOV  CODE,TEMP          PUSH PTR TO MESSAGE
0206 0152 0222            AI   TEMP,CRLF           : (CODE BASE+OFFSET)
     0154 00A1
0207 0156 CE82            MOV  TEMP,*SP+           :
0208 0158 0144'           DATA CALL$,TI$MSG       WRITE THE STRING
     015A 0136'
0209                ***************************************************
0210                *   WAIT FOR ACKNOWLEDGEMENT OF 'MSGBUF'          *
0211                ***************************************************
0212 015C CE81            MOV  MP,*SP+            PUSH MSH BUFFER PTR
0213 015E 0158'           DATA CALL$,C$WAIT       WAIT FOR ACKNOWLEDGMENT
     0160 0000
```

```
0215                ****************************************************************
0216                * WRITE 'PRODUCER RECEIVES ACKNOWLEDGEMENT OF c'               *
0217                ****************************************************************
0218 0162 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     0164 0016
0219 0166 CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     0168 0000
0220 016A C088            MOV  CODE,TEMP          PUSH PTR TO MESSAGE
0221 016C 0222            AI   TEMP,ACKMSG         : (CODE BASE+OFFSET)
     016E 004D
0222 0170 CE82            MOV  TEMP,*SP+           :
0223 0172 015E'           DATA CALL$,TI$MSG       WRITE THE STRING
     0174 015A'
0224                *
0225 0176 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     0178 0016
0226 017A CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     017C 0000
0227 017E CEA1            MOV  @CSLOT(MP),*SP+    PUSH THE CHARACTER
     0180 0000
0228 0182 0172'           DATA CALL$,TI$COT       WRITE THE CHARACTER
     0184 0146'
0229                *
0230 0186 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
     0188 0016
0231 018A CEA9            MOV  @BAUD(LF),*SP+     PUSH BAUD RATE FLAG
     018C 0000
0232 018E C088            MOV  CODE,TEMP          PUSH PTR TO MESSAGE
0233 0190 0222            AI   TEMP,CRLF           : (CODE BASE+OFFSET)
     0192 00A1
0234 0194 CE82            MOV  TEMP,*SP+           :
0235 0196 0182'           DATA CALL$,TI$MSG       WRITE THE STRING
     0198 0174'
```

```
PRODUC     SDSMAC 3.3.0 79.312   10:08:10 THURSDAY, MAY 07, 1981.
PRODUC -- PRODUCER PROCESS FOR DEMONSTRATION                    PAGE 006
 0237              ************************************************************
 0238              * END { UNTIL 'CHAR' EQ 'Z' }                               *
 0239              ************************************************************
 0240 019A 8A21          C     @CSLOT(MP),Z(CODE)  IS CHARACTER 'Z'?
      019C 0000
      019E 0018
 0241 01A0 16B4          JNE   SENDLP              NO: LOOP
 0242              ************************************************************
 0243              * FREE 'CHNLID' (AUTOMATICALLY FREES 'MSGBUF')              *
 0244              ************************************************************
 0245 01A2 CEA9          MOV   @CHNLID(LF),*SP+    PUSH CHANNEL ID
      01A4 0002
 0246 01A6 0196'         DATA  CALL$,C$TERM        FREE THE CHANNEL
      01A8 0000
 0247              ************************************************************
 0248              * WRITE 'PRODUCER TERMINATES'                               *
 0249              ************************************************************
 0250 01AA CEA8          MOV   @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      01AC 0016
 0251 01AE CEA9          MOV   @BAUD(LF),*SP+      PUSH BAUD RATE FLAG
      01B0 0000
 0252 01B2 C088          MOV   CODE,TEMP           PUSH PTR TO MESSAGE
 0253 01B4 0222          AI    TEMP,GOODBY          : (CODE BASE+OFFSET)
      01B6 0073
 0254 01B8 CE82          MOV   TEMP,*SP+            :
 0255 01BA 01A6'         DATA  CALL$,TI$MSG        WRITE THE STRING
      01BC 0198'
 0256              ************************************************************
 0257              *                -- PROCESS TERMINATION CODE --             *
 0258              ************************************************************
 0259      01BE' EPILOG EQU  $
 0260 01BE CEA8          MOV   @LEXLVL-PRODUC(CODE),*SP+  PUSH LEXICAL LEVEL
      01C0 000A
 0261 01C2 01BA'         DATA  CALL$,E$PRCS                TERMINATE THIS PRCS
      01C4 0000
 0262 01C6 0460          B     @EXIT$P                     EXIT
      01C8 0000
 0263                    END
NO ERRORS,     NO WARNINGS
```

```
0002                    IDT  'CONSUM'
0003                ****************************************************
0004                *
0005                *              CONSUM: THE CONSUMER PROCESS
0006                *
0007                *   PURPOSE: THIS PROCESS FIRST ALLOCATES CHANNEL 1, AND
0008                *            THEN WAITS FOR A MESSAGE TO APPEAR ON THE
0009                *            CHANNEL. WHEN IT RECEIVES A MESSAGE, IT TAKES
0010                *            THE CHARACTER AND BAUD RATE FROM THE MESSAGE
0011                *            AND PRINTS THE CHARACTER ON THE USR TERMINAL.
0012                *            WHEN THE CONSUMER RECEIVES A 'Z', IT PRINTS
0013                *            THE CHARACTER AND TERMINATES.
0014                *
0015                *   CALLS:   TI$COT, TI$MSG
0016                *
0017                ****************************************************
0018                       DEF  CONSUM
0019                ****************************************************
0020                *                -- EXTERNAL REFERENCES --
0021                ****************************************************
0022                       REF  CALL$     CALL SYMBOL
0023                       REF  S$PRCS    START PROCESS
0024                       REF  E$PRCS    END PROCESS
0025                       REF  EXIT$P    EXIT
0026                       REF  C$INIT    ALLOCATE A CHANNEL
0027                       REF  C$TERM    FREE A CHANNEL
0028                       REF  C$RECE    RECEIVE A MESSAGE
0029                       REF  C$ACKN    ACKNOWLEDGE A MESSAGE
0030                       REF  TI$COT    WRITE CHARACTER
0031                       REF  TI$MSG    WRITE STRING
0032                ****************************************************
0033                *                      -- REGISTERS --
0034                ****************************************************
0035      0001      MP       EQU  1       POINTER TO MESSAGE BUFFER
0036      0002      TEMP     EQU  2       USED TO CALCULATE ADDRESS OF VARIABLES
0037                *
0038      0007      PR       EQU  7       REGISTER 7 POINTS TO PROCESS RECORD
0039      0008      CODE     EQU  8       REGISTER 8 POINTS TO CODE BASE
0040      0009      LF       EQU  9       REGISTER 9 POINTS TO LOCAL FRAME
0041      000A      SP       EQU  10      REGISTER 10 IS STACK POINTER
```

```
CONSUM     SDSMAC 3.3.0 79.312   10:09:00 THURSDAY, MAY 07, 1981.
CONSUM -- CONSUMER PROCESS FOR DEMONSTRATION                    PAGE 0003
0043                ***************************************************
0044                *              -- MAPPING FOR LOCAL FRAME ---           *
0045                ***************************************************
0046 0000                 DORG 0          OFFSET FROM LF (R9)
0047                *
0048 0000           CHNLID BSS  2         CHANNEL ID (PTR TO CHANNEL)
0049 0002           MSGBUF BSS  2         POINTER TO MESSAGE BUFFER
0050      0004      LFLEN  EQU  $
0051                *
0052                ***************************************************
0053                *             -- MAPPING FOR MESSAGE BUFFER --          *
0054                ***************************************************
0055 0000                 DORG 0          OFFSET FROM MSGBUF
0056                *
0057 0000           CSLOT  BSS  2         LOW ORDER BYTE IS ONE CHARACTER MESSAGE
0058 0002           BSLOT  BSS  2         ONE WORD FOR BAUD RATE FLAG
0059      0004      MBLEN  EQU  $
0060                *
0061                ***************************************************
0062                *                  -- PROCESS DESCRIPTOR   --           *
0063                ***************************************************
0064 0000                 PSEG
0065      0000      CONSUM EQU  $
0066 0000 0046            DATA PROLOG-CONSUM      OFFSET TO FIRST STATEMENT
0067 0002 00E4            DATA EPILOG-CONSUM      OFFSET TO TERMINATION CODE
0068 0004 0000            DATA 0                  ZERO FOR PROCESSES
0069 0006 0000            DATA >0000              SIZE OF ARGUMENTS (BYTES)
0070 0008 0004      FRMSIZ DATA LFLEN             TOTAL FRAME SIZE (BYTES)
0071 000A 0001      LEXLVL DATA >0001             LEXICAL LEVEL
0072 000C 0100      PRIORI DATA >0100             PROCESS PRIORITY
0073 000E 0100      STKSIZ DATA >0100             PROCESS STACKSIZE (WORDS)
0074 0010 0000      HPSIZE DATA >0000             PROCESS HEAPSIZE (WORDS)
```

```
CONSUM     SDSMAC 3.3.0 79.312   10:09:00 THURSDAY, MAY 07, 1981.
CONSUM -- CONSUMER PROCESS FOR DEMONSTRATION                    PAGE 0004
0076              ************************************************************
0077              *                        -- USER DATA --                   *
0078              ************************************************************
0079              *
0080       0012   CHNAME EQU  $-CONSUM
0081  0012 0001          DATA 1                    USE CHANNEL 1
0082              *
0083       0014   MSGLEN EQU  $-CONSUM
0084  0014 0004          DATA MBLEN                LENGTH OF MESSAGE BUFFER
0085              *
0086       0016   PORT01 EQU  $-CONSUM
0087  0016 0080          DATA >0080                CRU BASE OF PORT NUMBER 1
0088              *
0089       0018   Z      EQU  $-CONSUM
0090  0018 005A          DATA >5A                  Z IN ASCII
0091              *
0092       001A   RECMSG EQU  $-CONSUM
0093  001A   43          TEXT 'CONSUMER RECEIVES '
0094  002C   00          BYTE 0
0095              *
0096       002D   TERMSG EQU  $-CONSUM
0097  002D   43          TEXT 'CONSUMER TERMINATES'
0098  0040   0D          BYTE >0D,>0A              <CR><LF>
      0041   0A
0099  0042   00          BYTE 0                    <NUL> TO TERMINATE STRING
0100              *
0101       0043   CRLF   EQU  $-CONSUM
0102  0043   0D          BYTE >0D,>0A,>00          <CR><LF><NUL>
      0044   0A
      0045   00
0103  0046               EVEN                      CODE MUST START ON WORD BNDRY
```

```
CONSUM     SDSMAC 3.3.0 79.312   10:09:00 THURSDAY, MAY 07, 1981.
CONSUM -- CONSUMER PROCESS FOR DEMONSTRATION                    PAGE 0005
 0105            ************************************************************
 0106            *                  -- PROCESS START CODE --                *
 0107            ************************************************************
 0108      0046' PROLOG EQU  $
 0109 0046 CEA8        MOV  @FRMSIZ-CONSUM(CODE),*SP+ PUSH FRAME SIZE, BYTES
      0048 0008
 0110 004A CEA8        MOV  @LEXLVL-CONSUM(CODE),*SP+ PUSH LEXICAL LEVEL
      004C 000A
 0111 004E CEA8        MOV  @PRIORI-CONSUM(CODE),*SP+ PUSH PRIORITY
      0050 000C
 0112 0052 CEA8        MOV  @STKSIZ-CONSUM(CODE),*SP+ PUSH STACKSIZE, WORDS
      0054 000E
 0113 0056 CEA8        MOV  @HPSIZE-CONSUM(CODE),*SP+ PUSH HEAPSIZE, WORDS
      0058 0010
 0114 005A 0000        DATA CALL$,S$PRCS              START THIS PROCESS
      005C 0000
 0115            ************************************************************
 0116            * ALLOCATE CHANNEL 'CHNAME', PUT RESULT IN 'CHNLID'        *
 0117            ************************************************************
 0118 005E CEA8        MOV  @CHNAME(CODE),*SP+ PUSH NAME OF CHANNEL (1)
      0060 0012
 0119 0062 C089        MOV  LF,TEMP           'PUSH PTR TO CHANNEL ID
 0120 0064 0222        AI   TEMP,CHNLID         : (LOCAL FRAME+OFFSET)
      0066 0000
 0121 0068 CE82        MOV  TEMP,*SP+           :
 0122 006A 005A'       DATA CALL$,C$INIT       ALLOCATE THE CHANNEL
      006C 0000
```

```
CONSUM     SDSMAC 3.3.0 79.312   10:09:00 THURSDAY, MAY 07, 1981.
CONSUM -- CONSUMER PROCESS FOR DEMONSTRATION                    PAGE 000
 0124              ************************************************************
 0125              * UNTIL 'CHAR' EQ 'Z' RECEIVE CHARACTER MESSAGE
 0126              ************************************************************
 0127 006E CEA9    WAITLP MOV  @CHNLID(LF),*SP+   PUSH CHANNEL ID
      0070 0000
 0128 0072 C089           MOV  LF,TEMP            PUSH PTR TO MSGBUF
 0129 0074 0222           AI   TEMP,MSGBUF         : (LOCAL FRAME+OFFSET)
      0076 0002
 0130 0078 CE82           MOV  TEMP,*SP+           :
 0131 007A 006A'          DATA CALL$,C$RECE       WAIT FOR MESSAGE
      007C 0000
 0132              ************************************************************
 0133              * WRITE 'CONSUMER RECEIVES c'
 0134              ************************************************************
 0135 007E C069           MOV  @MSGBUF(LF),MP    MOVE VALUE TO REGISTER MP
      0080 0002
 0136 0082 CEA8           MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      0084 0016
 0137 0086 CEA1           MOV  @BSLOT(MP),*SP+    PUSH BAUD RATE FLAG
      0088 0002
 0138 008A C088           MOV  CODE,TEMP          PUSH PTR TO MESSAGE
 0139 008C 0222           AI   TEMP,RECMSG         : (CODE BASE+OFFSET)
      008E 001A
 0140 0090 CE82           MOV  TEMP,*SP+           :
 0141 0092 007A'          DATA CALL$,TI$MSG       WRITE THE STRING
      0094 0000
 0142              *
 0143 0096 CEA8           MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      0098 0016
 0144 009A CEA1           MOV  @BSLOT(MP),*SP+    PUSH BAUD RATE FLAG
      009C 0002
 0145 009E CEA1           MOV  @CSLOT(MP),*SP+    PUSH THE CHARACTER
      00A0 0000
 0146 00A2 0092'          DATA CALL$,TI$COT       WRITE THE CHARACTER
      00A4 0000
 0147              *
 0148 00A6 CEA8           MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      00A8 0016
 0149 00AA CEA1           MOV  @BSLOT(MP),*SP+    PUSH BAUD RATE FLAG
      00AC 0002
 0150 00AE C088           MOV  CODE,TEMP          PUSH PTR TO MESSAGE
 0151 00B0 0222           AI   TEMP,CRLF           : (CODE BASE+OFFSET)
      00B2 0043
 0152 00B4 CE82           MOV  TEMP,*SP+           :
 0153 00B6 00A2'          DATA CALL$,TI$MSG       WRITE THE STRING
      00B8 0094'
 0154              ************************************************************
 0155              * ACKNOWLEDGE RECEIPT OF THE MESSAGE                        *
 0156              ************************************************************
 0157 00BA CE81           MOV  MP,*SP+            PUSH MESSAGE BUFFER PTR
 0158 00BC 00B6'          DATA CALL$,C$ACKN       ACKNOWLEDGE THE MESSAGE
      00BE 0000
```

```
0159                ***************************************************************
0160                * END { UNTIL 'CHAR' EQ 'Z' }                                   *
0161                ***************************************************************
0162 00C0 8A21            C     @CSLOT(MP),Z(CODE)  IS CHARACTER 'Z'?
     00C2 0000
     00C4 0018
0163 00C6 16D3            JNE   WAITLP              NO: LOOP
```

```
CONSUM     SDSMAC 3.3.0 79.312   10:09:00 THURSDAY, MAY 07, 1981.
CONSUM -- CONSUMER PROCESS FOR DEMONSTRATION                    PAGE 0007
 0165                ********************************************************
 0166                * FREE 'CHNLID'                                          *
 0167                ********************************************************
 0168 00C8 CEA0            MOV  @CHNLID,*SP+       PUSH CHANNEL ID
      00CA 0000
 0169 00CC 00BC'           DATA CALL$,C$TERM       FREE THE CHANNEL
      00CE 0000
 0170                ********************************************************
 0171                * WRITE 'CONSUMER TERMINATES'                            *
 0172                ********************************************************
 0173 00D0 CEA8            MOV  @PORT01(CODE),*SP+ PUSH CRU BASE OF PORT 1
      00D2 0016
 0174 00D4 CEA1            MOV  @BSLOT(MP),*SP+    PUSH BAUD RATE FLAG
      00D6 0002
 0175 00D8 C088            MOV  CODE,TEMP          PUSH PTR TO MESSAGE
 0176 00DA 0222            AI   TEMP,TERMSG         : (CODE BASE+OFFSET)
      00DC 002D
 0177 00DE CE82            MOV  TEMP,*SP+
 0178 00E0 00CC'           DATA CALL$,TI$MSG       WRITE THE STRING
      00E2 00B8'
 0179                ********************************************************
 0180                *              -- PROCESS TERMINATION CODE --            *
 0181                ********************************************************
 0182      00E4'  EPILOG EQU  $
 0183 00E4 CEA8            MOV  @LEXLVL-CONSUM(CODE),*SP+  PUSH LEXICAL LEVEL
      00E6 000A
 0184 00E8 00E0'           DATA CALL$,E$PRCS               TERMINATE THIS PRCS
      00EA 0000
 0185 00EC 0460            B    @EXIT$P                    EXIT
      00EE 0000
 0186                      END
NO ERRORS,     NO WARNINGS
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                    PAGE 00C
 0002                    IDT    'WAITIO'
 0003                    DEF    TI$SET
 0004                    DEF    TI$CIN
 0005                    DEF    TI$COT
 0006                    DEF    TI$MSG
 0007              *
 0008              ***************************************************************
 0009              *                                                             *
 0010              *      TI$SET:     SET THE BAUD RATE OF A TERMINAL            *
 0011              *                                                             *
 0012              *                                                             *
 0013              *   PURPOSE:   THIS ROUTINE AUTO-BAUDS A TERMINAL ATTACHED    *
 0014              *              TO A 9902 INTERFACE.  WHEN CALLED, IT INPUTS   *
 0015              *              A CHARACTER (EITHER AN 'A' OR A <CR>) AND      *
 0016              *              TESTS THE LENGTH OF THE START BIT TO FIND      *
 0017              *              THE BAUD RATE. THE 9901 TIMER IS USED TO       *
 0018              *              TIME THE START BIT.                            *
 0019              *                                                             *
 0020              *   CALLING_SEQUENCE:                                         *
 0021              *              PUSH CRU ADDRESS OF PORT                       *
 0022              *              PUSH POINTER TO BAUD RATE FLAG                 *
 0023              *              CALL TI$SET                                    *
 0024              *                                                             *
 0025              *              MOV  @<CRU ADDRESS>,*SP+                       *
 0026              *              MOV  @<ADDRESS OF BAUD RATE FLAG>,*SP+         *
 0027              *              DATA CALL$,TI$SET                              *
 0028              *                                                             *
 0029              *   INPUTS:    THE CRU ADDRESS OF THE PORT IS USUALLY >80     *
 0030              *              FOR PORT ONE AND >180 FOR PORT TWO.            *
 0031              *                                                             *
 0032              *   OUTPUTS:   TI$SET INITIALIZES THE 9902 PORT AND SETS      *
 0033              *              THE BAUD RATE. THE BAUD RATE FLAG (THE         *
 0034              *              SECOND PARAMETER) IS SET TO THE PROPER         *
 0035              *              VALUE WITH WHICH TO SET THE XDR AND RDR        *
 0036              *              VALUES OF THE 9902. VALUES USED ARE:           *
 0037              *                                                             *
 0038              *                   BAUD:      INIT CHAR (12 BITS)          *
 0039              *                   19200          >01A                     *
 0040              *                    9600          >034                     *
 0041              *                    4800          >068                     *
 0042              *                    2400          >0D0                     *
 0043              *                    1200          >A10                     *
 0044              *                     600          >340                     *
 0045              *                     300          >4D0                     *
 0046              *                     110          >638                     *
 0047              *                                                             *
 0048              *   EXCEPTIONS: NONE.                                         *
 0049              *                                                             *
 0050              *   CALLS:      NONE.                                         *
 0051              *                                                             *
 0052              *   REFERENCES: TMS 9901 PROGRAMMABLE SYSTEMS INTERFACE       *
 0053              *               COPYRIGHT 1978 BY TI INC., NUMBER MP003       *
```

```
0054            *                                                              *
0055            *                TMS 9902 ASYNCHRONOUS COMMUNICATIONS          *
0056            *                     CONTROLLER DATA MANUAL,                  *
0057            *               COPYRIGHT 1978 BY TI INC., NUMBER MP004        *
0058            *                                                              *
0059            ****************************************************************
```

```
0061            * ROUTINE LIST:
0062            *       TI$SET
0063            *
0064            *
0065            *       EQUATES
0066            *
0067     000D   LDIR    EQU     13      9902 - LOAD INTERVAL REGISTER
0068     000F   RIN     EQU     15      9902 - RECEIVE INPUT BIT
0069     0010   RTSON   EQU     16      9902 - REQUEST TO SEND ON
0070     0012   RIENB   EQU     18      9902 - RECEIVER INTERRUPT ENABLE
0071     0015   RBRL    EQU     21      9902 - RECEIVER BUFFER REGISTER LOADED
0072     0016   XBRE    EQU     22      9902 - TRANSMIT BUFFER REGISTER EMPTY
0073     001F   RESET   EQU     31      9902 - RESET BIT
0074     0100   CB9901  EQU     >100    9901 CRU BASE
0075            *
0076     0001   BAUDP   EQU     1
0077     0002   COUNT   EQU     2
0078     0003   TBLPTR  EQU     3
0079     0004   TIMVAL  EQU     4
0080            *
0081     0007   PR      EQU     7
0082     0008   CODE    EQU     8
0083     0009   LF      EQU     9
0084     000A   SP      EQU     10
0085            *
0086     000C   CRUBAS  EQU     12
0087            *
```

```
0089 0000                PSEG
0090       0000' TI$SET EQU $
0091 0000 0000           DATA   0                        OPTIMIZED LINKAGE
0092                     ********************************************************
0093                     *                    GET ARGUMENTS                     *
0094                     ********************************************************
0095 0002 C2AD           MOV    @SP*2(R13),SP            GET CALLER'S SP
     0004 0014
0096 0006 022A           AI     SP,-4                    RESET SP
     0008 FFFC
0097 000A CB4A           MOV    SP,@SP*2(R13)            RESTORE CALLER'S SP
     000C 0014
0098 000E C31A           MOV    *SP,CRUBAS               SET CRU BASE OF PORT
0099 0010 C06A           MOV    @2(SP),BAUDP             SET BAUD POINTER
     0012 0002
0100                     ********************************************************
0101                     *                 INITIALIZE THE 9902                  *
0102                     ********************************************************
0103 0014 1D1F           SBO    RESET                    RESET THE 9902
0104 0016 3220           LDCR   @CONTRL,8                LOAD CONTROL REGISTER
     0018 005C'
0105 001A 1E0D           SBZ    LDIR                     DISABLE INTERVAL TIMER
0106                     ********************************************************
0107                     *                  WAIT FOR CHARACTER                  *
0108                     ********************************************************
0109 001C 04C2           CLR    COUNT                    CLEAR TIMER COUNT
0110 001E 1F0F TESTSP    TB     RIN                      SPACE?
0111 0020 13FE           JEQ    TESTSP                   NO: TEST AGAIN
0112                     ********************************************************
0113                     *             INITIALIZE 9901 AND TIME SPACE BIT       *
0114                     ********************************************************
0115 0022 020C           LI     CRUBAS,CB9901            SET CRUBASE AT TMS9901
     0024 0100
0116 0026 0704           SETO   TIMVAL                   INITIAL TIMER VALUE >3FFF
0117                     *                                : CONTROL BIT IS GETS 1
0118 0028 1E00           SBZ    0                        SET 9901 TO INT. MODE
0119 002A 33C4           LDCR   TIMVAL,15                LOAD AND START 9901 TIMER
0120 002C 1E00           SBZ    0                        CONTROL BIT = 1, SET TO 0
0121                     *                                : TO TURN INT. ON
0122 002E 1000           NOP                             DELAY WHILE 9902 SETS BIT
0123 0030 1000           NOP                              : (FOR 1481) MUST WAIT
0124                     *                                : BEFORE CHANGING BASES.
0125 0032 C31A           MOV    *SP,CRUBAS               SET CRU BASE TO 9902 PORT
0126 0034 1F0F TIMELP    TB     RIN                      STILL SPACE?
0127 0036 16FE           JNE    TIMELP                   FALL OUT ON MARK
0128 0038 020C           LI     CRUBAS,CB9901+2          BASE OF 9901+<1 BIT>
     003A 0102
0129 003C 1DFF           SBO    -1                       SET 9901 TO CLOCK MODE
0130 003E 3782           STCR   COUNT,14                 STORE 9901 COUNT VALUE
0131 0040 1EFF           SBZ    -1                       SET 9901 TO INT. MODE
```

```
0133                    ************************************************************
0134                    *  RESET FOR 9902 PORT AND IGNORE THE AUTO-BAUD CHARACTER  *
0135                    ************************************************************
0136 0042 C31A                  MOV     *SP,CRUBAS              SET CRU BASE TO 9902 PORT
0137 0044 1F15  LOADLP  TB      RBRL                            CHARACTER FINISHED?
0138 0046 16FE                  JNE     LOADLP                  NO: CHECK AGAIN
0139 0048 1E12                  SBZ     RIENB                   RESET RBRL
0140                    ************************************************************
0141                    *               LOOK UP COUNT IN BAUD TABLE                *
0142                    ************************************************************
0143 004A 0203                  LI      TBLPTR,BAUDTB           GET ADDRESS OF BAUD TABLE
     004C 005E
0144 004E 8CC2  BAUDLP  C       COUNT,*TBLPTR+                  MATCH IF HIGH OR EQUAL
0145 0050 1402                  JHE     MATCH                   YES, SET BAUD RATE
0146 0052 05C3                  INCT    TBLPTR                  NO, UPDATE TABLE PTR
0147 0054 10FC                  JMP     BAUDLP
0148 0056 3313  MATCH   LDCR    *TBLPTR,12                      INITIALIZE RDR,XDR,
0149                    *                                         RESET LRDR,LXDR
0150 0058 C453                  MOV     *TBLPTR,*BAUDP          SET RETURNED BAUD RATE
0151                    ************************************************************
0152                    *                     ---- END OF CODE ----                  *
0153                    ************************************************************
0154 005A 0380                  RTWP
0155                    ************************************************************
0156                    *                   --- DATA SECTION ---                    *
0157                    ************************************************************
0158                    *
0159 005C   62  CONTRL  BYTE    >62             9902 CONTROL REGISTER VALUE
0160                    *                       SPECIFIES:      TWO STOP BITS
0161                    *                                       EVEN PARITY
0162                    *                                       7 BITS/CHAR
0163                    *
0164                    *     9901 TIMER VALUE (14 BITS)      9902 BAUD RATE FLAG
0165                    *                     |                       |
0166                    *                     |       +---------------------+
0167                    *                     |       |
0168                    *                     V       V
0169 005E 3FFD  BAUDTB  DATA    >3FFD,>01A              BAUD = 19200
     0060 001A
0170 0062 3FF8          DATA    >3FF8,>034              BAUD =  9600
     0064 0034
0171 0066 3FF0          DATA    >3FF0,>068              BAUD =  4800
     0068 0068
0172 006A 3FE1          DATA    >3FE1,>0D0              BAUD =  2400
     006C 00D0
0173 006E 3FC1          DATA    >3FC1,>1A0              BAUD =  1200
     0070 01A0
0174 0072 3F82          DATA    >3F82,>340              BAUD =   600
     0074 0340
0175 0076 3EC9          DATA    >3EC9,>4D0              BAUD =   300
     0078 04D0
0176 007A 0000          DATA    >0000,>638              BAUD =   110
```

007C 0638

```
0178            ****************************************************************
0179            *                                                              *
0180            *      TI$CIN:     INPUT A CHARACTER                           *
0181            *                                                              *
0182            *                                                              *
0183            *   PURPOSE:  THIS ROUTINE INPUTS A CHARACTER FROM A PORT      *
0184            *             OF A 9902. THE CHARACTER IS NOT ECHOED.          *
0185            *                                                              *
0186            *   CALLING_SEQUENCE:                                          *
0187            *             PUSH CRU ADDRESS OF PORT                         *
0188            *             PUSH ADDRESS OF WORD TO RECEIVE CHARACTER        *
0189            *             CALL TI$CIN                                      *
0190            *                                                              *
0191            *             MOV @<CRU ADDRESS>,*SP+                          *
0192            *             MOV @<POINTER TO WORD>,*SP+                      *
0193            *             DATA CALL$,TI$CIN                                *
0194            *                                                              *
0195            *   INPUTS:   THE CRU ADDRESS OF THE PORT IS USUALLY >80       *
0196            *             FOR PORT NUMBER ONE, AND >180 FOR PORT TWO.      *
0197            *                                                              *
0198            *   OUTPUTS:  THE CHARACTER IS STORED IN THE LOW 7 BITS         *
0199            *             OF THE WORD POINTED TO BY ARGUMENT TWO.          *
0200            *                                                              *
0201            *   EXCEPTIONS: NONE.                                          *
0202            *                                                              *
0203            *   CALLS:      NONE.                                          *
0204            *                                                              *
0205            *   REFERENCES: TMS 9902 ASYNCHRONOUS COMMUNICATIONS           *
0206            *                     CONTROLLER DATA MANUAL,                  *
0207            *                  COPYRIGHT 1978 BY TI INC, NUMBER MP004      *
0208            *                                                              *
0209            ****************************************************************
```

```
0211                    *
0212                    * EQUATES
0213                    *
0214        0001        CHARP   EQU     1
0215                    *
0216                    * PROGRAM SEGMENT
0217                    *
0218 007E               TI$CIN  PSEG
0219        007E'       TI$CIN  EQU $
0220 007E 0000                  DATA    0                       OPTIMIZED LINKAGE
0221                    ********************************************************
0222                    *                       GET ARGUMENTS                  *
0223                    ********************************************************
0224 0080 C2AD                  MOV     @SP*2(R13),SP           GET CALLER'S SP
     0082 0014
0225 0084 022A                  AI      SP,-4                   RESET SP
     0086 FFFC
0226 0088 CB4A                  MOV     SP,@SP*2(R13)           RESTORE CALLER'S SP
     008A 0014
0227 008C C31A                  MOV     *SP,R12                 SET CRU BASE OF PORT
0228 008E C06A                  MOV     @2(SP),CHARP            SET BAUD POINTER
     0090 0002
0229                    ********************************************************
0230                    *           WAIT FOR CHARACTER AND STORE IT            *
0231                    ********************************************************
0232 0092 1F15  WAITLP  TB      RBRL                    BUFFER FULL?
0233 0094 16FE          JNE     WAITLP                  NO:  WAIT
0234 0096 04D1          CLR     *CHARP                  YES: STORE IN LOW BYTE
0235 0098 35D1          STCR    *CHARP,7                 : OF WORD POINTED TO
0236 009A 06D1          SWPB    *CHARP                   : BY CHARP
0237 009C 1E12          SBZ     RIENB                   RESET RBRL
0238                    ********************************************************
0239                    *                   ---- END OF CODE ----              *
0240                    ********************************************************
0241 009E 0380          RTWP
```

```
0243            ************************************************************
0244            *                                                          *
0245            *      TI$COT:        OUTPUT A CHARACTER                   *
0246            *                                                          *
0247            *                                                          *
0248            *   PURPOSE:  THIS ROUTINE OUTPUTS A CHARACTER TO A PORT   *
0249            *             OF A 9902. IF THE TERMINAL IS A 1200 BAUD    *
0250            *             PORT THE CHARACTER IS PADDED WITH 3 NULLS.   *
0251            *             IF THE CHARACTER WAS A <CR>, IT IS PADDED    *
0252            *             WITH 3 TO 23 NULLS, DEPENDING ON THE BAUD    *
0253            *             RATE, TO GIVE A 200MS DELAY.                 *
0254            *                                                          *
0255            *   CALLING_SEQUENCE:                                      *
0256            *             PUSH CRU ADDRESS OF PORT                     *
0257            *             PUSH BAUD RATE FLAG                          *
0258            *             PUSH WORD, RIGHTMOST BYTE OF WHICH IS CHAR   *
0259            *             CALL TI$COT                                  *
0260            *                                                          *
0261            *             MOV @<CRU ADDRESS OF PORT>,*SP+              *
0262            *             MOV @<BAUD RATE FLAG>,*SP+                   *
0263            *             MOV @<WORD>,*SP+                             *
0264            *             DATA CALL$,TI$COT                            *
0265            *                                                          *
0266            *   INPUTS:   PORT:  CRU ADDRESS OF USER PORT, USUALLY     *
0267            *                    >80 FOR PORT 1, AND >180 FOR PORT 2.  *
0268            *             BAUD:  BAUD RATE FLAG SET BY TI$SET          *
0269            *             WORD:  LOW 7 BITS OF WORD ARE OUTPUT CHAR.   *
0270            *                                                          *
0271            *   OUTPUTS:  WRITES CHAR TO PORT, PADDED WITH 3 NULLS IF  *
0272            *             1200 BAUD. PADS <CR> WITH NULLS TO GIVE A    *
0273            *             200MS DELAY.                                 *
0274            *                                                          *
0275            *   EXCEPTIONS: NONE.                                      *
0276            *                                                          *
0277            *   CALLS:      TI$COT                                     *
0278            *                                                          *
0279            *   REFERENCES: TMS 9902 ASYNCHRONOUS COMMUNICATIONS       *
0280            *                     CONTROLLER DATA MANUAL,              *
0281            *                 COPYRIGHT 1978 BY TI INC, NUMBER MP004   *
0282            *                                                          *
0283            ************************************************************
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                        PAGE 000
 0285                    *
 0286                    * EQUATES
 0287                    *
 0288       0001   BAUDFL EQU    1
 0289       0002   TABLEP EQU    2
 0290       0003   CHR    EQU    3
 0291       0004   PADNLS EQU    4
 0292       0005   CRNULS EQU    5
 0293       0006   NUMNUL EQU    6
 0294                    *
 0295                    * PROGRAM SEGMENT
 0296                    *
 0297 00A0                      PSEG
 0298       00A0'  TI$COT EQU $
 0299 00A0 0000                 DATA   0                        OPTIMIZED LINKAGE
 0300                    ********************************************************
 0301                    *                  ---- GET ARGUMENTS ----
 0302                    ********************************************************
 0303 00A2 C2AD                 MOV    @SP*2(R13),SP            GET CALLER'S SP
      00A4 0014
 0304 00A6 022A                 AI     SP,-6                    RESET SP
      00A8 FFFA
 0305 00AA CB4A                 MOV    SP,@SP*2(R13)            RESTORE CALLER'S SP
      00AC 0014
 0306 00AE C31A                 MOV    *SP,R12                  SET CRU BASE OF PORT
 0307 00B0 C06A                 MOV    @2(SP),BAUDFL            BAUDFL := SECOND ARG
      00B2 0002
 0308 00B4 C0EA                 MOV    @4(SP),CHR               CHR := THIRD ARG
      00B6 0004
 0309 00B8 06C3                 SWPB   CHR                      PUT CHR INTO TOP BYTE
 0310                    ********************************************************
 0311                    *                  --- SEND CHARACTER ---
 0312                    ********************************************************
 0313 00BA 1D10                 SBO    RTSON                    SET REQUEST TO SEND
 0314 00BC 1F16  WTLP$1 TB     XBRE                     TRANSMIT BUFFER EMPTY?
 0315 00BE 16FE                 JNE    WTLP$1                   NO:  CHECK AGAIN
 0316 00C0 3203                 LDCR   CHR,8                    YES: SEND CHARACTER
 0317                    ********************************************************
 0318                    *          --- LOOK UP NULLS IN BAUDFL RATE TABLE ---
 0319                    ********************************************************
 0320 00C2 0202                 LI     TABLEP,TABLE             LOAD TABLE POINTER
      00C4 00F6'
 0321 00C6 8C81  TBLLP  C      BAUDFL,*TABLEP+          MATCH?
 0322 00C8 1203                 JLE    FOUND                    YES: GET NUMBER OF NULLS
 0323 00CA 0222                 AI     TABLEP,4                 NO:  UPDATE TABLE POINTE
      00CC 0004
 0324 00CE 10FB                 JMP    TBLLP                    LOOP
 0325 00D0 C132  FOUND  MOV    *TABLEP+,PADNLS          NUMBER OF NULLS PADDED
 0326 00D2 C152                 MOV    *TABLEP,CRNULS           NUMBER OF NULLS AFTER CR
```

```
0328                    ********************************************************************
0329                    *                --- SET NUMBER OF NULLS TO PAD ---                 *
0330                    ********************************************************************
0331 00D4 9803          CB      CHR,@CR                 CARRIAGE RETURN?
     00D6 00F5'
0332 00D8 1302          JEQ     PADCR                   YES: PAD CR
0333 00DA C184          MOV     PADNLS,NUMNUL           NO:  PAD REGULAR CHR
0334 00DC 1001          JMP     PAD                     JMP TO PAD CODE
0335 00DE C185  PADCR   MOV     CRNULS,NUMNUL           PAD REGULAR CHR
0336                    ********************************************************************
0337                    *                --- PAD WITH NUMNUL NULLS ---                      *
0338                    ********************************************************************
0339 00E0 C186  PAD     MOV     NUMNUL,NUMNUL           IS NUMNUL ZERO?
0340 00E2 1306          JEQ     SENDOF                  YES: EXIT
0341 00E4 1F16  WTLP$2  TB      XBRE                    TRANSMIT BUFFER EMPTY?
0342 00E6 16FE          JNE     WTLP$2                  NO: CHECK AGAIN
0343 00E8 3220          LDCR    @NULLCH,8               SEND CHARACTER
     00EA 00F4'
0344 00EC 0606          DEC     NUMNUL                  DECREMENT NUMBER NULLS
0345 00EE 10F8          JMP     PAD                     PAD AGAIN
0346                    ********************************************************************
0347                    *           --- DONE, TURN OFF REQUEST TO SEND ---                   *
0348                    ********************************************************************
0349 00F0 1E10  SENDOF  SBZ     RTSON                   BRING RTS HIGH AFTER -
0350                    *                                : CHARACTER HAS FINISHED
0351                    ********************************************************************
0352                    *                      ---- END OF CODE ----                        *
0353                    ********************************************************************
0354 00F2 0380          RTWP
```

```
0356                ************************************************************
0357                *                        --- DATA ---                          *
0358                ************************************************************
0359                *
0360 00F4    00     NULLCH BYTE   >00
0361 00F5    0D     CR     BYTE   >0D
0362                *
0363       00F6'    TABLE  EQU    $
0364                *
0365                *       BAUDFL RATE FLAG,  NULLS AFTER CHR, NULLS AFTER CR
0366                *                  |                  |                |
0367                *                  |    +-------------+                |
0368                *                  |    |  +----------------------------+
0369                *                  |    |  |
0370                *                  V    V  V
0371 00F6 001A             DATA   >01A,0,0          BAUDFL = 19200
     00F8 0000
     00FA 0000
0372 00FC 0034             DATA   >034,0,0          BAUDFL =  9600
     00FE 0000
     0100 0000
0373 0102 0068             DATA   >068,0,0          BAUDFL =  4800
     0104 0000
     0106 0000
0374 0108 00D0             DATA   >0D0,0,0          BAUDFL =  2400
     010A 0000
     010C 0000
0375 010E 01A0             DATA   >1A0,3,23         BAUDFL =  1200
     0110 0003
     0112 0017
0376 0114 0340             DATA   >340,0,11         BAUDFL =   600
     0116 0000
     0118 000B
0377 011A 04D0             DATA   >4D0,0,7          BAUDFL =   300
     011C 0000
     011E 0007
0378 0120 0638             DATA   >638,0,3          BAUDFL =   110
     0122 0000
     0124 0003
```

```
0380            ********************************************************
0381            *                                                      *
0382            *      TI$MSG:  OUTPUT A STRING, DELIMITED BY A NULL   *
0383            *                                                      *
0384            *                                                      *
0385            *   PURPOSE:  OUTPUT A STRING TO A 9902 PORT. THE STRING *
0386            *             IS COMPOSED OF CONSECUTIVE BYTES POINTED  *
0387            *             TO BY THE THIRD ARGUMENT, AND DELIMITED BY *
0388            *             A ZERO BYTE.                             *
0389            *                                                      *
0390            *   CALLING_SEQUENCE:                                  *
0391            *             PUSH CRU ADDRESS OF PORT                 *
0392            *             PUSH BAUD RATE FLAG FROM TI$SET          *
0393            *             PUSH POINTER TO MESSAGE STRING           *
0394            *                                                      *
0395            *             MOV @<CRU ADDRESS>,*SP+                  *
0396            *             MOV @<BAUD RATE FLAG>,*SP+               *
0397            *             MOV @<PTR TO MSG>,*SP+                   *
0398            *             DATA CALL$,TI$MSG                        *
0399            *                                                      *
0400            *   INPUTS:   PORT: CRU BASE OF OUTPUT PORT            *
0401            *             BAUD: BAUD RATE FLAG FROM TI$SET         *
0402            *             MSGP: POINTER TO MESSAGE STRING          *
0403            *                                                      *
0404            *   OUTPUTS:  OUTPUT IS SENT TO PORT.                  *
0405            *                                                      *
0406            *   EXCEPTIONS: NONE.                                  *
0407            *                                                      *
0408            *   CALLS:    TI$COT                                   *
0409            *                                                      *
0410            ********************************************************
0411            *
0412            * REFERENCES
0413            *
0414                    REF     CALL$
0415                    REF     EXIT$P
0416            *
0417            * EQUATES
0418            *
0419    0000    PORTOF  EQU     >0000           ADDRESS 2
0420    0002    BAUDOF  EQU     >0002           INTEGER 2
0421    0004    MSGP    EQU     >0004           POINTER 2
0422            *
0423    0001    MSGPTR  EQU     1
0424    0002    WORD    EQU     2
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                        PAGE 001
 0426 0126                PSEG
 0427      0126' TI$MSG   EQU $
 0428 0126 0008           DATA   MSGENT-TI$MSG          OFFSET TO EXECUTABLE COD
 0429 0128 0024           DATA   MSGEXI-TI$MSG          OFFSET TO TERMINATION CO
 0430 012A 0000           DATA   0                      LOCAL VARIABLE SIZE
 0431 012C 0006           DATA   6+0                    LOCAL FRAME SIZE
```

```
WAITIO     SDSMAC 3.3.0 79.312   09:59:39 THURSDAY, MAY 07, 1981.
TI$LIB -- WAIT LOOP DRIVEN I/O -- 6/25/80                        PAGE 001
 0433      012E' MSGENT EQU    $                      MSGENT POINT OF PROCESS
 0434            ************************************************************
 0435            *                  ---- MAIN BODY OF CODE ----
 0436            ************************************************************
 0437 012E C069           MOV    @MSGP(LF),MSGPTR      GET POINTER TO MESSAGE
      0130 0004
 0438 0132 04C2 CHARLP    CLR    WORD                  ZERO BOTH BYTES
 0439 0134 D0B1           MOVB   *MSGPTR+,WORD         CHR IN HIGH BYTE OF WORD
 0440 0136 1309           JEQ    MSGEXI                YES: QUIT
 0441 0138 CEA9           MOV    @PORTOF(LF),*SP+      PUSH PORTOF
      013A 0000
 0442 013C CEA9           MOV    @BAUDOF(LF),*SP+      PUSH BAUDOF RATE FLAG
      013E 0002
 0443 0140 06C2           SWPB   WORD                  CHR IN LOW BYTE OF WORD
 0444 0142 CE82           MOV    WORD,*SP+             PUSH WORD ON STACK
 0445 0144 0000           DATA   CALL$,TI$COT          SEND CHR
      0146 00A0'
 0446 0148 10F4           JMP    CHARLP
 0447            ************************************************************
 0448            *                      ---- END OF CODE ----
 0449            ************************************************************
 0450      014A' MSGEXI   EQU    $                     EXIT CODE
 0451 014A 0460           B      @EXIT$P
      014C 0000
 0452                     END
NO ERRORS,     NO WARNINGS
```

```
0001                    IDT  'CONFIG'            SPECIFY CONFIGURATION
0002                * REVISION: 08/01/80  1.00    ORIGINAL FOR RX 2.0
0003                * ROUTINE LIST: CONFIG, IWP$0 .. IWP$15, BAD$WP,
0004                *                $RAMTB, $RESTA, $LREX, $SYSCR,
0005                *                $DEFAU, $FILL, $STKSZ, $BOOTP,
0006                *                $IODIR, DB$WP
0007                * COPY MODULES:
0008                *      NONE.
0009                * MACRO DEFINITIONS:
0010                *      NONE.
0011                * EXTERNAL ROUTINES:
0012                *      NONE.
0013                * EXTERNAL DATA:
0014 0000              PSEG
0015                * MODULE CONSTANTS:
0016       0018     IWPSZ  EQU  24                SIZE OF AN INTERRUPT
0017                *                               WORKSPACE (R4-R15)
0018       5000     LOWRAM EQU  >5000             LOW BOUNDARY OF RAM
0019                * MODULE VARIABLES:
0020                *
0021 5000              DORG LOWRAM
0022                *
0023                   DEF  IWP$0,IWP$1,IWP$2,IWP$3
0024                   DEF  IWP$4,IWP$5,IWP$6,IWP$7
0025                   DEF  IWP$8,IWP$9,IWP$10,IWP$11
0026                   DEF  IWP$12,IWP$13,IWP$14,IWP$15
0027                   DEF  BAD$WP,DB$WP
0028 5000         IWP$0  BSS  32
0029 5020         IWP$1  BSS  32
0030       5020   DB$WP  EQU  IWP$1
0031       5038   IWP$2  EQU  $-32+IWPSZ
0032 5040                BSS  IWPSZ
0033       5050   IWP$3  EQU  $-32+IWPSZ
0034 5058                BSS  IWPSZ
0035       5068   IWP$4  EQU  $-32+IWPSZ
0036 5070                BSS  IWPSZ
0037       5080   IWP$5  EQU  $-32+IWPSZ
0038 5088                BSS  IWPSZ
0039       5098   IWP$6  EQU  $-32+IWPSZ
0040 50A0                BSS  IWPSZ
0041       50B0   IWP$7  EQU  $-32+IWPSZ
0042 50B8                BSS  IWPSZ
0043       50C8   IWP$8  EQU  $-32+IWPSZ
0044 50D0                BSS  IWPSZ
0045       50E0   IWP$9  EQU  $-32+IWPSZ
0046 50E8                BSS  IWPSZ
0047       50F8   IWP$10 EQU  $-32+IWPSZ
0048 5100                BSS  IWPSZ
0049       5110   IWP$11 EQU  $-32+IWPSZ
0050 5118                BSS  IWPSZ
0051       5128   IWP$12 EQU  $-32+IWPSZ
0052 5130                BSS  IWPSZ
```

```
0053       5140   IWP$13 EQU   $-32+IWPSZ
0054 5148                BSS   IWPSZ
0055       5158   IWP$14 EQU   $-32+IWPSZ
0056 5160                BSS   IWPSZ
0057       5170   IWP$15 EQU   $-32+IWPSZ
0058 5178                BSS   IWPSZ
0059 5190         BAD$WP BSS   32
0060               *
```



```
0061       51B0   LOWHP  EQU   $
0062               *
0063 0000                RORG
```

```
CONFIG     SDSMAC 3.3.0 79.312   16:57:27 WEDNESDAY, MAY 06, 1981.
CONFIG:          SPECIFY CONFIGURATION                          PAGE 000
0066             * ABSTRACT:
0067             *       SPECIFY CERTAIN SYSTEM PARAMETERS, THE RAM
0068             *       CONFIGURATION, AND THE I/O SUBSYSTEM
0069             *       DIRECTORY.
0070             * CALLING SEQUENCE:
0071             *       NONE.
0072             * EXCEPTIONS AND CONDITIONS:
0073             *       NONE.
0074             * LOCAL DATA:
0075             *       NONE.
0076             * ENTRY POINT:
0077             *       NONE.
0078             ************************************************************
0079             * ADDRESS OF THE "BLWP" VECTOR FOR RESTARTS; USE "0" FOR
0080             * LEVEL 0 INTERRUPT, ">FFFC" FOR THE "LREX" VECTOR, OR
0081             * THE ADDRESS OF A USER-DEFINED VECTOR.
0082             ************************************************************
0083                     DEF  $RESTA
0084 0000 0000   $RESTA DATA 0
0085             ************************************************************
0086             * ADDRESS OF THE "BLWP" VECTOR FOR THE "LREX" INSTRUCTION;
0087             * USE "0" IF THERE IS TO BE NO "LREX" VECTOR OR IF HIGH
0088             * MEMORY IS ROM.
0089             ************************************************************
0090                     DEF  $LREX
0091 0002 0000   $LREX  DATA 0
0092             ************************************************************
0093             * ADDRESS OF THE USER-DEFINED ROUTINE TO BE INVOKED IN CASE
0094             * OF A SYSTEM CRASH; USE "0" FOR THE SYSTEM DEFAULT WHICH
0095             * IS TO MASK INTERRUPTS AND IDLE THE PROCESSOR.
0096             ************************************************************
0097                     DEF  $SYSCR
0098 0004 0000   $SYSCR DATA 0
0099             ************************************************************
0100             * ADDRESS OF THE MPP ROUTINE TO BE INVOKED IF AN EXCEPTION
0101             * OCCURS BUT NO EXCEPTION HANDLER HAS BEEN SPECIFIED; USE
0102             * "0" FOR THE SYSTEM DEFAULT WHICH IS A "NO EXCEPTION
0103             * HANDLER" SYSTEM CRASH.
0104             ************************************************************
0105                     DEF  $DEFAU
0106 0006 0000   $DEFAU DATA 0
0107             ************************************************************
0108             * THIS IS THE VALUE WITH WHICH THE HEAP WILL BE
0109             * INITIALIZED AT POWER-UP.
0110             ************************************************************
0111                     DEF  $FILL
0112 0008 10FF   $FILL  JMP  $
0113             ************************************************************
0114             * THIS IS THE DEFAULT STACK SIZE (IN WORDS) THAT IS USED
0115             * IF A "STACKSIZE" CONCURRENT PARAMETER IS NOT SPECIFIED.
0116             ************************************************************
0117                     DEF  $STKSZ
```

```
0118 000A 0100   $STKSZ DATA >10
0119             ********************************************************
0120             * THE PARAMETER LIST FOR THE CALL TO "$$PRCS" TO START THE
0121             * "BOOT" PROGRAM.
0122             ********************************************************
0123                    DEF  $BOOTP
0124 000C 0000   $BOOTP DATA >0000              FRAME SIZE
0125 000E 0000          DATA >0000              LEXICAL NESTING LEVEL
```

```
CONFIG      SDSMAC 3.3.0 79.312   16:57:27 WEDNESDAY, MAY 06, 1981.
CONFIG:!          SPECIFY CONFIGURATION                          PAGE 000
 0126 0010 0000           DATA >0000              PRIORITY
 0127 0012 0100           DATA >0'00              STACK SIZE
 0128 0014 0000           DATA >0000              HEAP SIZE
 0129                     ********************************************************
 0130                     * ADDRESS OF THE "RAM TABLE," THE TABLE THAT DESCRIBES THE
 0131                     * REGIONS OF READ-WRITE MEMORY TO BE COLLECTED INTO THE
 0132                     * HEAP.
 0133                     ********************************************************
 0134                           DEF  $RAMTB
 0135 0016 001A'  $RAMTB DATA RAMTB
 0136                     ********************************************************
 0137                     * ADDRESS OF THE DIRECTORY OF I/O SUBSYSTEMS.
 0138                     ********************************************************
 0139                           DEF  $IODIR
 0140 0018 0020'  $IODIR DATA IODIR
 0141                     ********************************************************
 0142                     * THE FOLLOWING TABLE IS A LIST OF "LENGTH_IN_BYTES,
 0143                     * STARTING_ADDRESS" PAIRS THAT DEFINE THE RAM TO BE USED
 0144                     * BY THE EXECUTIVE; A WORD OF "0" TERMINATES THE LIST.
 0145                     * THE RAM REGIONS MUST BE IN ASCENDING ORDER AND MUST NOT
 0146                     * OVERLAP.
 0147                     ********************************************************
 0148 001A 4E50   RAMTB   DATA >A000-LOWHP,LOWHP
      001C 51B0
 0149 001E 0000           DATA 0                  LIST TERMINATOR
 0150                     ********************************************************
 0151                     * THE FOLLOWING TABLE IS A LIST OF "SERVICE_DIRECTORY,
 0152                     * PORT_CONSTANTS" PAIRS THAT DEFINE THE I/O SUBSYSTEM TO
 0153                     * BE INITIALIZED WHEN ROUTINE "D$INIT" IS CALLED;
 0154                     * A WORD OF "0" TERMINATES THE LIST.
 0155                     ********************************************************
 0156          0020'  IODIR  EQU  $
 0157                     *
 0158                     *      INSERT LIST ENTRIES HERE.
 0159                     *
 0160 0020 0000           DATA 0                  LIST TERMINATOR
 0161                     *
 0162                     END
NO ERRORS,    0001 WARNINGS
```

```
TXSLNK        2.3.0  78.244       01/00/00  00:01:29                PAGE 001
COMMAND LIST
; Rx Link Edit Control File for Rx 2.0 Demonstration Program
;
SYMT                    ; Specify a symbol table
TASK RXDEMO             ; Name of Load Module
INCLUDE DSC2.RXKERN     ; Kernel - RX (standard) - DB (debug)
;INCLUDE DSC2.EPRCS0    ; Optional E$PRCS Module
;                         (Use only with Rxkern, not with Dbkern)
;
INCLUDE DSC2.CONFIG     ; Standard Config
INCLUDE DSC2.RXDEMO     ; System Process
INCLUDE DSC2.PRODUC     ; Producer Process
INCLUDE DSC2.CONSUM     ; Consumer Process
INCLUDE DSC2.WAITIO     ; Wait Loop Driven I/O
;
FIND DSC2.RX1OBJ        ; Standard Rx Routines
FIND DSC2.CHNOBJ        ; Channel Routines
FIND DSC2.CLKOBJ        ; Clock Management
FIND DSC2.RX2OBJ        ; Optional Rx Routines
END
```