TEXAS INSTRUMENTS

9900

The Microprocessor Pascal System

User's Manual

MICROPROCESSOR SERIES™

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time to improve design and to supply the best possible 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. Application run-time packages generated from the Microprocessor Pascal system may, however, be reproduced for resale exclusively by the customer purchasing the Microprocessor Pascal system.

All manuals associated with Microprocessor Pascal (MP 351) 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 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 assumed neither for its use nor for any infringement of patents or rights of others 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.

PREFACE

Texas Instruments Microprocessor Pascal is a complete high level language development system specifically designed for microprocessor applications. TI's FS990 floppy based development system is the minimum hardware necessary to develop a software program written in Microprocessor Pascal. Pascal programs execute on a wide spectrum of TI's microprocessor based products from a 9900 microprocessor chip set to TM9900 microcomputer modules to 990 minicomputers. As a virtual superset of the proposed international Pascal standard, Microprocessor Pascal incorporates several language extensions such as concurrent task execution, expanded input/output capability, and specific library utilities for microprocessor applications. The Microprocessor Pascal system provides total software capability--from design to debug--resulting in a dramatic decrease in software costs and application development time.

This book introduces the user to the various features of the Microprocessor Pascal system. The manual content features introductory information (Sections 1 thru 3); and information on software development tools (Sections 4 thru 7), the Microprocessor Pascal language (Section 8), the Run-Time Support Libraries (Sections 9 thru 14), and operation of the development and target systems (Sections 15 thru 17). In addition, various topics that support the manual text are discussed in the appendices.

The Microprocessor Pascal User's Guide documents the Microprocessor Pascal system for the user. Future changes to the software making up the Microprocessor Pascal system will be indicated in revisions to this manual.

INTRODUCTION

SECTION I. MICROPROCESSOR PASCAL SYSTEM OVERVIEW

SECTION II. MICROPROCESSOR PASCAL SYSTEM CONCEPTS

SECTION III. MICROPROCCESSOR PASCAL SYSTEM EXAMPLE

SOFTWARE DEVELOPMENT TOOLS

SECTION IV. SOURCE EDITOR GUIDE

SECTION V. COMPILER AND NATIVE CODE GENERATOR

SECTION VI. HOST DEBUGGER GUIDE

SECTION VII. CONVENTIONAL PASCAL PROGRAM EXECUTION

THE MICROPROCESSOR PASCAL SYSTEM

SECTION VIII. THE MICROPROCESSOR PASCAL SYSTEM

RTS LIBRARY

SECTION IX. PROCESS SYNCHRONIZATION AND PROCESSOR MANAGEMENT

SECTION X. PROCESS COMMUNICATION

SECTION XI. PROCESS MANAGEMENT

SECTION XII. MEMORY MANAGEMENT

SECTION XIII. ERROR RECOVERY AND EXCEPTION HANDLING

SECTION XIV. IMPLEMENTATION OF DEVICE HANDLERS

OPERATIONS

SECTION XV. DEVELOPMENT SYSTEM OPERATION ON DX/10

APPENDICES

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

MICROPROCESSOR PASCAL SYSTEM OVERVIEW

1.1 WHAT IS MICROPROCESSOR PASCAL?

The Microprocessor Pascal System is a package of software development tools and run-time utilities which support the use of the Pascal programming language on the Texas Instruments 9900 Family of microprocessors.

Microprocessor Pascal is intended for applications that execute on small computers that do not have a general-purpose, disc-based operating system but do require multitasking executive support. Pascal, as extended to become the Microprocessor Pascal System, is well-suited for such applications. The Microprocessor Pascal System language supports structured programming, compiler-enforced compatibility checks that enhance program reliability, user-declared data structures, and reentrant code that can be shared among tasks. Microprocessor Pascal System language extensions allow multiple sites of execution (called processes), synchronization among processes via semaphores, and access to the communication register unit (CRU) of the 9900 Family. The execution of Microprocessor Pascal System-coded applications is supported by the Executive Run Time Support Libraries. Executive Run Time Support is a memory-resident executive that supports multiprogramming, Microprocessor Pascal System-coded interrupt handlers, process synchronization through semaphores, interprocess communication through Microprocessor Pascal System file I/O, dynamic creation and reclamation of processes, and process scheduling according to a multiple priority scheme.

A major benefit of the use of the Microprocessor Pascal System language and the Executive Run Time Support is that each designer of an application system does not have to specify system-wide standards and conventions for inter-module parameter passing, register usage, task control blocks, etc. These design decisions have been made by Texas Instruments in mapping the Microprocessor Pascal System onto the architecture of the 9900 family. Microprocessor Pascal System I/O provides a standard high-level module interface that permits the development of modular software component libraries from which users can select exactly those modules that are needed for a particular application. For example, Texas Instruments provides interface modules for common peripheral devices of the TM990 family of 9900-based microcomputer boards. The design of the Executive Run Time Support also emphasizes software modularity. The components of the Executive Run Time Support are provided in a form that permits the user to include in his applications only those features that are actually used.

Within the framework provided by the Microprocessor Pascal System, the production of a multi-tasking application is to a great degree reduced to construction of a collection of processes, each of which is basically a sequential Pascal program. Well-defined techniques and supportive executive components are provided to manage the time-dependent interactions among processes. The user of the Microprocessor Pascal System can concentrate on the algorithms that comprise his application without concern about the implementation of the executive under which they execute.

The user of the Microprocessor Pascal System develops software using one of the following host computer systems; either a single-user FS990/4 or FS990/10 floppy disc development system or a multi-user DS990/10 hard disc minicomputer. The Microprocessor Pascal System provides four major components which support software development on a host computer:

- An intelligent, interactive editor for source preparation

- A compiler to compile source into interpretive code

- A code generator to generate 9900 object code

- An interactive debugging interpreter.

The figure below illustrates these four major components, providing the user with an overview of the Microprocessor Pascal System.

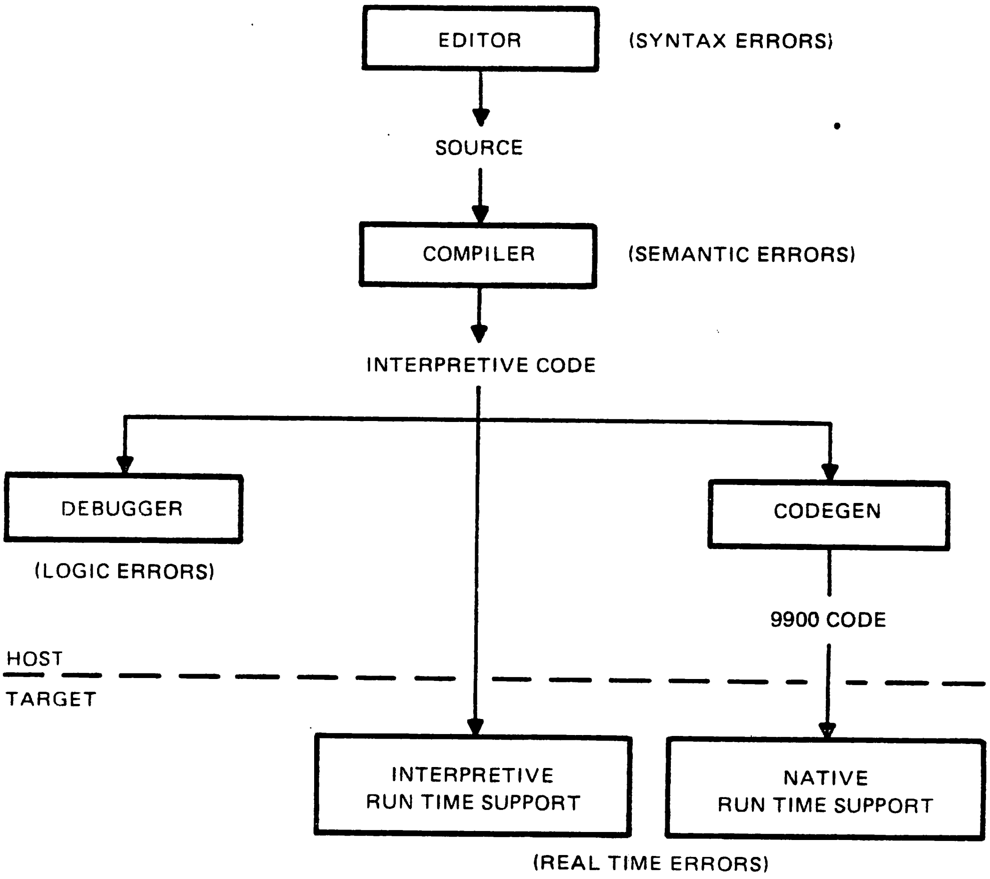


FIGURE 1-1 THE MICROPROCESSOR PASCAL SYSTEM

1.2 SOURCE EDITOR

The Microprocessor Pascal System provides an interactive source editor designed to help in the creation and modification of Microprocessor Pascal System source files. The editor interacts with the user at a video display terminal by displaying a desired portion of the file on the screen and allowing a cursor to be moved around within this display. Changes may be made to the file by simply typing over the old text with the new text, or by adding, moving or deleting complete lines or blocks of several lines. The editor helps the user input correct Pascal statements by syntax checking the complete file on command. If an error is detected, an appropriate error message is given to allow the user to correct the error before syntax checking continues. The editor also has features which help with the indentation of structured statements. Each time a new line is added, the editor positions the cursor to the current indentation level. The cursor location is only a suggested indentation level; the user can easily reposition the cursor to a different one.

1.3 COMPILER

The Microprocessor Pascal System provides a compiler that translates Microprocessor Pascal System source code into interpretive code for a hypothetical stack computer. This code may be executed interpretively using the Host Debugger or may be used as input to the Native Code Generator. The interpretive code is characterized by its small size, about half the size of 9900 native code. Another advantage of interpretive code is the minimal time required to produce an executable system that can be debugged at a functional level. The Microprocessor Pascal System Compiler processes the full Microprocessor Pascal System language and detects syntactic and semantic errors at the source level. The compiler also supports a "copy" statement which allows the user to partition a complete Microprocessor Pascal System system into separate source files which are easier to maintain and edit.

1.4 NATIVE CODE GENERATOR

A code generator will also be provided with the Microprocessor Pascal System in a future release; the Native Code Generator will translate the interpretive code from the compiler into 9900 object code.

1.5 HOST DEBUGGER

The Microprocessor Pascal System provides an interactive debugger which enables the user to debug systems at a functional level. The debugger supports symbolic references to module names, file names, and common names. A module is any unit of Pascal which may be invoked that is either a system, program, process, procedure or function. Statements can be referred to by Pascal statement numbers. Either a single process or a set of processes can be debugged at once. Breakpoints can be used to stop the execution at any point by specifying the Pascal statement number of a particular module. When execution is suspended, the status of the system can be examined. Examples include the status of each process in the system, as well as the values of module variables, common variables, or variables allocated from a dynamic heap. Data can also be modified if desired. The execution of the system can be traced at the process scheduling level, module entry and exit level, or module statement flow level. Target hardware interfaces such as CRU references and interrupts may be simulated in the debugging mode.

1.6 EXECUTIVE RUN TIME SUPPORT

The executive components of the Microprocessor Pascal System are provided in two versions that correspond to the two types of output that are available with the Microprocessor Pascal System compiler -- interpretive and native (object) code. The Interpretive Run Time Support Library supports interpretive execution. It is generally used for applications for which the program compactiion that is achieved with interpretation is more important than the associated decrease in execution speed. The Native Code Run Time Support Library supports the execution of 9900 Native code and is generally used for time-critical applications. Both versions of the executive provide the same capabilities to the user and will be referred to as the Executive Run Time Support Libraries.

SECTION II

MICROPROCESSOR PASCAL SYSTEM CONCEPTS

This section deals with the general concepts and structures which comprise the Microprocessor Pascal System. It is primarily designed to introduce the concepts which will be used in this document. The following topics include processes, programs, systems, memory management, reentrancy, and recursion.

2.1 PROCESS

A conventional Pascal program consists of a main program along with zero or more procedures or functions. Execution of such a program is done serially; that is, execution proceeds from one statement to the next so that execution exists at a single point in a single run-time environment at any time. However, the practice of having several sites of execution within one program at the same time, called multiprogramming, is often desirable. To permit multiprogramming, the concept of a process (a separately executing entity with its own run-time environment for its data) has been introduced into the Microprocessor Pascal System language. A conventional program is normally executed by one process which is created by the run-time support system before the program starts executing, and is destroyed when the program ends.

A process has resources which are allocated when the process is first started and deallocated when the process terminates. A process' stack holds variables which are statically declared in the process. A process' heap holds variables which are dynamically allocated and deallocated according to the NEW and DISPOSE statements, respectively. A process record is associated with each process by the Executive Run-Time Support and is used to hold implementation-dependent information.

A process is executed by a processor which is a CPU hardware device. The Microprocessor Pascal System is designed for the Texas Instruments model 990 computer and 9900 microprocessor. A process has a state which, if the process were snapshot at some instant, would indicate the next instruction the process is to execute as well as the current values of all data variables which it can address. If a process is temporarily suspended for any reason (such as the occurrence of an interrupt), the state of the process must be restored before the process can continue its computation. The state of a process includes at least the machine context (workspace pointer, program counter, and status register) which is saved in the process record. In the Microprocessor Pascal System, code produced by the compiler is not self modifying, so the state of a process does not include the instructions themselves. The instruction stream is invariant with respect to the execution of processes. The Microprocessor Pascal System Compiler also produces references to local data that are relative to the stack region. Invariant code and relative data references provide reentrancy, which allows one copy of code to be in simultaneous use by more than one process.

In a stand-alone environment, having multiple points (sites) of execution within a program and being able to support interrupt handling is convenient. As an example, a process control problem could be solved by preparing a process to control each type of device in the system and a particular instance of that process for each actual device. General multiprogramming is also possible using the process concept. The Executive Run Time Support allows the user to write programs and processes in Pascal to service interrupts and devices and realize general multiprogramming.

Multiprogramming (or multitasking) allows more than one process to be executing in the same system. Multiprogramming was originally introduced to achieve better utilization of large computers: rather than going to an idle state to await the occurrence of some external event such as an input/output operation, the processor could execute another program. Typically, each program was a step of a batch job that did not interact with other jobs. Improved processor utilization is not the primary reason the Executive Run Time Support supports multiprogramming; it is intended for applications (such as process control) that have a high degree of parallelism. Each concurrent activity is best managed by a separate software module that models its behavior. This one-to-one correspondence between external activity and software control programs provides a powerful technique for the decomposition of a complex problem into modular components. Such modularity is important for the simplification of software development and testing and for the application of previously developed modules to new problems.*

Processes can communicate among themselves and synchronize with each other using the Executive Run Time Support. Logically, they execute concurrently (at the same time) competing for the processor's response. Processes are scheduled (selected for execution) based on process readiness and process priority. Process readiness is an indication of whether a process may either proceed or must wait for some condition to occur. Process priority is an indication (a number) of the relative urgency of the process. The lower the number, the more time critical the associated process is.

* In the terminology of Executive Run Time Support, a program is a special case of a process, so "multiprocessing" would be a more appropriate term than "multiprogramming"; however, "multiprocessing" has been used to describe execution on multiple processors.

2.2 LANGUAGE EXTENSIONS TO SUPPORT PROCESSES

A conventional Pascal program is structured as a PROGRAM module that has zero or more procedures or functions declared within it. There is one site of execution per program. The Pascal language has been extended to form the Microprocessor Pascal System language by adding constructs to declare and concurrently start processes, each of which has a site of execution. Two types of processes have special properties and are given the names "system" and "program".

The extensions in the Microprocessor Pascal System language have been designed to aid the user in the following areas:

Process declaration is distinct from the declaration of a procedure or function

Process declarations may be nested, and the Pascal scope rules of global variables are enforced as usual

Process parameters may be declared, and the START statement allows the passing of process parameters with full type checking by the Microprocessor Pascal System System compiler

Variables within scope of a process are guaranteed to exist even if processes which are lexical ancestors have terminated

The PROGRAM construct is a special case of a process in that it has no variables global to it. Its resources are given a special treatment by the run-time support

Any process or program which is within scope can be concurrently executed with the START statement. To allow all program declarations (declared at level one) to be in scope, the SYSTEM construct at level zero contains all program declarations.

2.2.1 SYSTEM Declaration

The SYSTEM is the outermost level of declarations and executable statements in a Microprocessor Pascal System. All other modules are contained within the SYSTEM; programs are nested within the SYSTEM, and processes are nested within programs as well as within other processes. An example of the nesting of the system, programs, and processes is illustrated in Figure 2-1.

A SYSTEM is the process in which execution begins. It bootstraps the SYSTEM by initializing global parameters and starting the programs which comprise the SYSTEM. It may not have any variables except possibly variables in COMMONs.

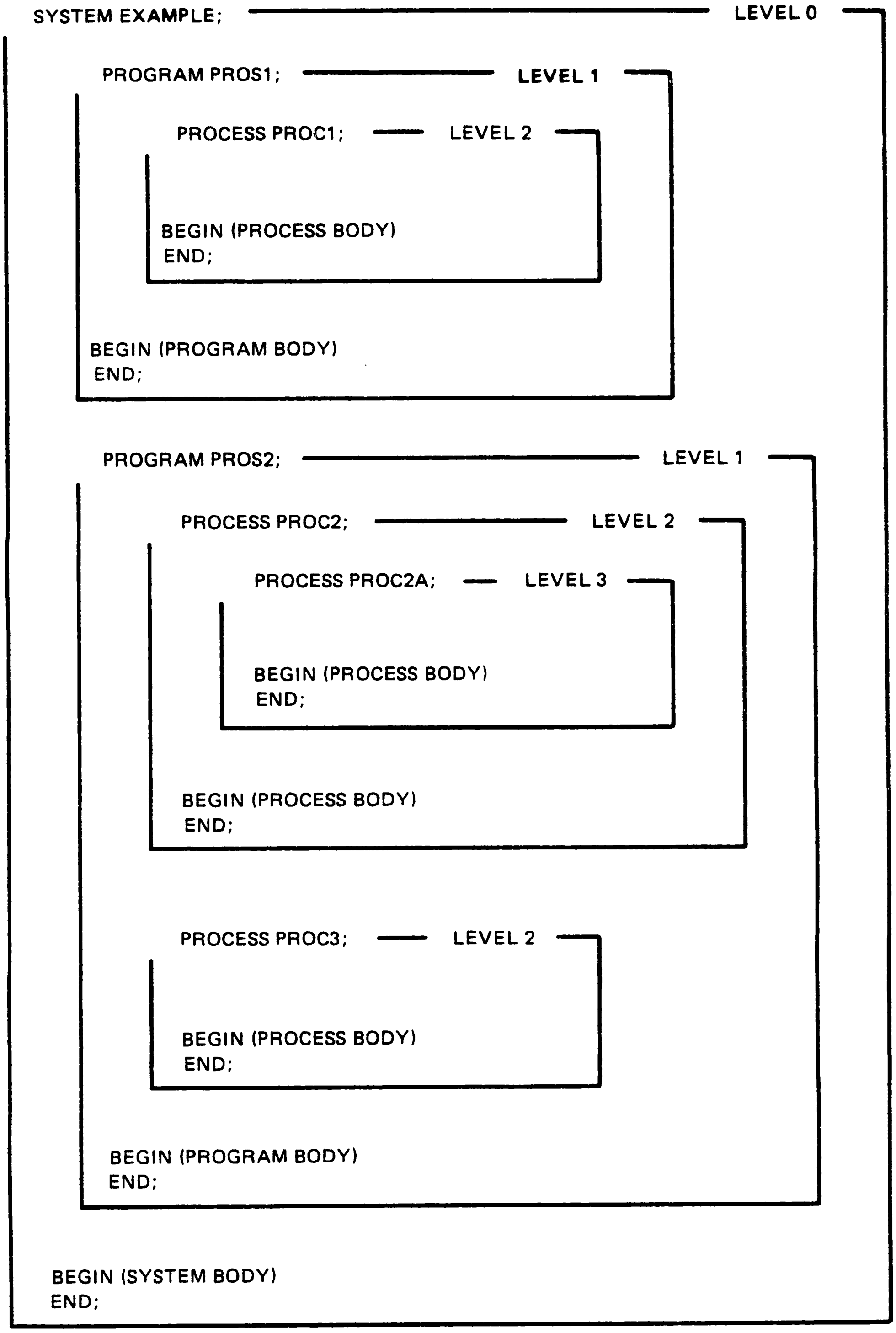


FIGURE 2-1. NESTING OF SYSTEM, PROGRAM, AND PROCESS DECLARATIONS

2.2.2 PROGRAM Declaration

Using the Microprocessor Pascal System, multitasking is possible. Because of this feature more than one program may be declared within the same SYSTEM. Processes and routines (procedures and functions) may be declared in a PROGRAM within the SYSTEM. In addition, the Microprocessor Pascal System also supports conventional Pascal which allows only single program environments.

A program is a process that is self-contained with respect to accessing data via scope of variables or pointers. It corresponds to the PROGRAM construct of the Pascal language and has no external data available to it except possibly through COMMONs.

2.2.3 PROCESS Declaration

A process may only be declared within a program, or within another process; within a process, procedures and functions may be declared along with other processes. A process may have value parameters associated with it, and may also have access to all variables which are declared global to it.

2.3 MEMORY

Each program or process has two data structures associated with it to manage memory. One of these is called the stack and the other is called the heap. The stack is an area allocated to the declared variables of the program or process and its procedures. The heap holds dynamically allocated variables, which are not declared but are created and destroyed by the procedures NEW and DISPOSE.

2.3.1 System Memory

System memory comprises all the data space which is possibly available for use. It must, however, be memory which the Executive Run Time Support system knows to use. System memory is a resource from which the program data structures are constructed.

2.3.2 Stack

A stack is implemented by using a block of storage called a stack region, out of which stack frames are allocated upon routine entry and deallocated upon routine exit. These stack frames are managed on a last-in, first-out basis. Each frame or activation record corresponds to a particular call of a program, process, procedure, or function, and includes space for variables, for temporaries, and for an administration area. (Refer to Figure 2-2.)

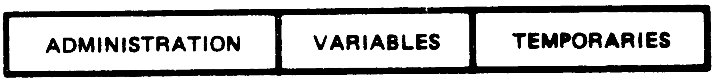


FIGURE 2-2. TYPICAL STACK FRAME

2.3.3 Heap. A heap is an area of memory which may be allocated in arbitrary sized packets which may then be returned and reused. These packets are used to hold dynamically allocated variables. Heaps may be one of two types: program or nested. Programs have heaps which are created from system memory. A nested heap is allocated out of another heap, called the parent, so that a hierarchy of heaps may be created. When a process is started, it is specified either to have its own heap (nested) or to share that of its lexical parent. NEW and DISPOSE use the heap associated with the process from which they are called. Hence, each procedure, function, process, or program may use only one heap using NEW and DISPOSE.

A heap is implemented as a heap region with an administration packet and allocated and unallocated packets. All dynamically allocated variables are allocated from the heap and returned to the heap in program-dependent order (using NEW and DISPOSE). (See Figure 2-3.)

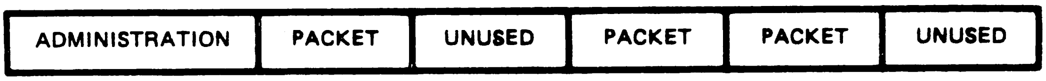


FIGURE 2-3. HEAP STRUCTURE

2.4 REENTRANCY

Reentrancy is a property of code (of which Microprocessor Pascal System stem c System example) which allows multiple activated copies or calls of a code module to be executing at the same time. These activations execute independently of each other, causing modifications of separate areas of data though using physically the same code. This is made possible by initializing all variables by executable code, not using self-modifying code, and keeping local variables and temporaries in an unshared data space. This allows, for example, many users to execute the same copy of a text editor, though working on different text. The controller for a device can be implemented by a routine that has as a parameter the identification of the specific instance of that device that must be controlled. If the code is reentrant, then the same handler can be invoked to control a number of devices.

2.5 RECURSION

Recursion is a property whereby an algorithm (solution) is expressed in terms of itself. This occurs whenever a routine calls itself directly (direct recursion), as well as when a calling routine is called (through a series of calls) without having first returned (indirect recursion). Implementing recursion requires that data references be relative to unshared data spaces for each activation of a routine. The reentrant nature of Pascal code easily supports the implementation of recursion. An example of direct recursion is that of factorials of positive integers: the factorial of N is N times the factorial of N-1 (the factorial of zero is 1). This is expressed as:

```
FACTORIAL(n) := n*FACTORIAL(n-1)

FACTORIAL(0) := 1
```

In Pascal, this could be coded as:

```
function factorial (n: integer): integer;
begin
  assert n >= 0;
  if n = 0 then factorial := 1
  else factorial := n * factorial(n - 1)
end;
```

SECTION III

MICROPROCESSOR PASCAL SYSTEM EXAMPLE

3.1 OVERVIEW

Much attention has recently been given to methods for producing good quality software. However, the solution to this problem remains a highly subjective one. Two software designers may use very different methods to achieve the same result, namely, a reliable, maintainable software product that performs the desired function.

Though methodological views may differ, one point of agreement emerges. The software design process must become more and more disciplined. Software systems must be simple, adaptable, and reliable if they are to achieve a long lifetime of use.

We cannot hope to present even a cursory overview of modern software design principles and techniques in this short section. The purpose here is to present a small view of the software development cycle using the Microprocessor Pascal System. A simple software system is presented with a step-by-step example showing how the system can be implemented using the Microprocessor Pascal System.

3.2 PROBLEM DEFINITION AND STRUCTURING

An early design problem is the decision regarding how the system is to be structured. The system can usually be divided into fairly independent functional units. Each functional unit should be defined so that it can be understood in terms of the inputs it can receive and the outputs it is expected to produce. In this way, the interfaces between the units form a nearly complete definition of the system. Each functional unit can then be designed and implemented one at a time. Moreover, a single unit can be systematically tested in isolation from all other units in order to verify that it performs the required function. It is possible to construct even the most complex systems in this incremental fashion.

In terms of a Microprocessor Pascal System implementation, a functional unit can be considered to be a process. A Microprocessor Pascal System can be divided into separate processes, each of which accepts a set of inputs and produces a set of outputs. A single process can be viewed in isolation from other processes. The behavior of each process can be verified one at a time, before the process is placed into the system in its normal context.

3.3 AN EXAMPLE

The example chosen for this section is a simple system containing a producer process and a consumer process. The two processes communicate with each other through a message buffer. A message in this system could be any kind of data structure. However, in the example, a message is simply a single character value from A to Z. The producing process may send a message to the message buffer without waiting for the message to be copied by the consuming process. The producing process is suspended only if the required buffer space is not available. The consuming process copies the character out of the message buffer. This process is suspended only if the buffer space is empty or not available for exclusive access. (Refer to Figure 3-1.)

In Figure 3-2 on the following page, a message buffer is declared as a COMMON variable which is a record. The "slots" field is the circular buffer into which messages are deposited by the producer and fetched by the consumer. The "next_in" field indicates where the next incoming message is to be deposited. The "next_out" field indicates from where the next outgoing message is to be fetched The "exclusive_access" field is a semaphore used to guarantee that only one process has access to the message buffer at a given instant. The "not_empty" field is a semaphore used to ensure that the buffer is not empty when removing messages from it. The "not_full" field is a semaphore used to ensure that there is an available space in the buffer when depositing a message.

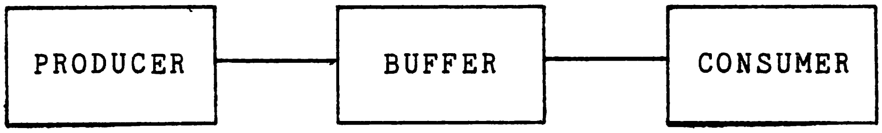


FIGURE 3-1. DIAGRAM OF INPUT/OUTPUT

```
DX Microprocessor Pascal System Compiler     1.0   05/24/79 13:51:41
  0 {$ DEBUG, MAP}
  0 SYSTEM tutorial;
  0
  0 CONST  number_of_slots = 10; .maximum number of slots in a buffer
  0
  0 TYPE   slot_index = 1..number_of_slots;
  0        alpha_character = 'A'..'Z';
  0
  0 COMMON message_buffer: RECORD             "circular message buffer
  0            slots: ARRAY [slot_index] OF alpha_character;
  0            next_in, next_out: slot_index;
  0            not_empty, not_full: SEMAPHORE;
  0            exclusive_access: SEMAPHORE;
  0            END;
  0
  0 ACCESS message_buffer;
  0
  0 PROCEDURE initsemaphore( var sema: SEMAPHORE; count: INTEGER);
  4     EXTERNAL;
  0 PROCEDURE signal(sema:SEMAPHORE); EXTERNAL;
  0 PROCEDURE wait(sema: SEMAPHORE); EXTERNAL;
  0 PROCEDURE swap; EXTERNAL;
  0
  0   PROGRAM producer;
  0   VAR item: alpha_character;
  2       line: PACKED ARRAY [1..16] OF CHAR;
 18       status: INTEGER;
 20   ACCESS message_buffer;
  1   BEGIN    {#priority = 20; stacksize = 100}
  1     item := 'A';
  2     line := 'item produced:  ';
  3     WITH m = message_buffer DO
  4       WHILE TRUE DO BEGIN
  5         wait(m.not_full);               "wait for buffer space
  6         wait(m.exclusive_access);       "get acces to the buffer
  7         m.slots [m.next_in]  := item;   "deposit item into buffer
  8         ENCODE(line, 16, status, item);
 .9         MESSAGE(line);
 10         IF item = 'Z' THEN item := 'A'  "generate the next item
 12         ELSE item := SUCC(item);
 13         m.next_in := SUCC(m.next_in MOD number_of_slots);
 14         signal(m.exclusive_access);     "release access to buffer
 15         signal(m.not_empty);       "indicate presence of another item
 16         swap;                      "give the consumer a chance
 17         END;
 17   END;
 17
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES (1 OF 3)

```
  0    PROGRAM consumer;
  0    VAR item: alpha_character;
  2        line: PACKED ARRAY [1..16] OF CHAR;
 18        status: INTEGER;
 20    ACCESS message_buffer;
  1    BEGIN {#priority = 20; stacksize = 100}
  1      line := 'item consumed:  ';
  2      WITH m = message_buffer DO
  3        WHILE TRUE DO BEGIN
  4          wait(m.not_empty);      "wait for available item to appear
  5          wait(m.exclusive_access);        "wait for access to buffer
  6          item := m.slots m.next_out ;   "extract item from buffer
  7          ENCODE(line, 16, status, item);
  8          MESSAGE(line);
  9          m.next_out := SUCC(m.next_out MOD number_of_slots);
 10          signal(m.exclusive_access);       "release access to buffer
 11          signal(m.not_full);"indiacte an available space in buffer
 12          swap;                  "give the producer a chance
 13          END;
 13    END;
  1  BEGIN {#stacksize = 300; heapsize = 500}
  1    WITH m = message_buffer DO BEGIN "initialize the message buffer
  2      m.next_in := 1;                  "index of first in-coming item
  3      m.next_out := 1;                 "index of first out-going item
  4      initsemaphore(m.exclusive_access, 1); "one access at a time
  5      initsemaphore(m.not_empty, 0); "of full slots in the buffer
  6      initsemaphore(m.not_full, number_of_slots);"#of empty slots
  7      END;
  7    START producer;
  8    START consumer;
  9  END.

SYSTEM TUTORIAL;
    STACK SIZE = 0000

          COMMON    TYPE      SIZE
          MESSAGE_  RECORD    30

          FIELD     DISP      TYPE      SIZE
          SLOTS     0000      ARRAY     20
          NEXT_IN   0014      SUBRANGE  2
          NEXT_OUT  0016      SUBRANGE  2
          NOT_EMPT  0018      SEMAPHORE 2
          NOT_FULL  001A      SEMAPHORE 2
          EXCLUSIV  001C      SEMAPHORE 2
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES (2 OF 3)

```
PROCEDURE INITSEMA ( VAR SEMA    :SEMAPHORE; COUNT   :INTEGER); EXTERNAL;

PROCEDURE SIGNAL   ( SEMA    :SEMAPHORE); EXTERNAL;

PROCEDURE WAIT     ( SEMA    :SEMAPHORE); EXTERNAL;

PROCEDURE SWAP    ; EXTERNAL;

PROGRAM PRODUCER;
    STACK SIZE = 0014

          VARIABLE  DISP      TYPE      SIZE
          ITEM      0000      SUBRANGE  2
          LINE      0002      STRING    16
          STATUS    0012      INTEGER   2

PROGRAM CONSUMER;
    STACK SIZE = 0014

          VARIABLE  DISP      TYPE      SIZE
          ITEM      0000      SUBRANGE  2
          LINE      0002      STRING    16
          STATUS    0012      INTEGER   2
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES (3 OF 3)

The text on the following page is a sample debugging session for the producer/consumer example; an explanation follows.

```
 HOST DEBUGGER     1.0     05/24/79  13:52:12

     Enter system heap size in (K)bytes: 5
     Do you wish to debug the most recently compiled system?
     Please answer YES or NO:YES

  debug(producer)

  debug(consumer)

  go
     run-time support now initialized

  go
     *** Process Created *** PRODUC(2)

  sdp(producer)

  ab(producer, 10)

  go
     *** Process Created *** CONSUM(3)

  dap
Status Summary of All Existing Processes

                      Site of                              Enabled    Stmt
     Process Name   Execution        Status          Pri Traces    Bkpts

  0  IDLE$P         IDLE$P    0     Ready           32767          no
  1  TUTORI         runtime code    Active           1             no
  2   PRODUC        PRODUC    0     Ready           20             yes
  3   CONSUM        CONSUM    0     Wait            20             no

  ab(consumer.1, 9)

  go
     *** Breakpoint ***       PRODUC(2).PRODUC    Statement 10

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 0041 0000 0000 0000 0000 0000 0000 0000  (.A..............)
 C96C (0010) 0000 0000 0001 0001                      (........        )

  go
     *** Breakpoint ***       CONSUM(3).CONSUM    Statement 9

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 0041 0000 0000 0000 0000 0000 0000 0000  (.A..............)
 C96C (0010) 0000 0000 0002 0001                      (........        )

  go
```

```
    *** Breakpoint ***      PRODUC(2).PRODUC    Statement 10

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 0041 0042 0000 0000 0000 0000 0000 0000  (.A.B............)
 C96C (0010) 0000 0000 0002 0002                      (........        )

  go
    *** Breakpoint ***      CONSUM(3).CONSUM    Statement 9

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 0041 0042 0000 0000 0000 0000 0000 0000  (.A.B............)
 C96C (0010) 0000 0000 0003 0002                      (........        )

  go
    *** Breakpoint ***      PRODUC(2).PRODUC    Statement 10

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 0041 0042 0043 0000 0000 0000 0000 0000  (.A.B.C..........)
 C96C (0010) 0000 0000 0003 0003                      (........        )

  hp(consumer)

  dab(producer)

  go
     Idle Instruction

  dap
Status Summary of All Existing Processes

                       Site of                               Enabled    Stmt
    Process Name   Execution         Status          Pri Traces     Bkpts

  0  IDLE$P         IDLE$P    0      Active          32767           no
  2   PRODUC        runtime code     Wait Sema       20              no
  3   CONSUM        runtime code     Hold            20              yes

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 004B 004C 0043 0044 0045 0046 0047 0048  (.K.L.C.D.E.F.G.H)
 C96C (0010) 0049 004A 0003 0003                      (.I.J....        )

  hp(producer)

  rp(consumer)

  dap
Status Summary of All Existing Processes

                       Site of                               Enabled    Stmt
    Process Name   Execution         Status          Pri Traces     Bkpts
  0  IDLE$P          IDLE$P    0     Ready          32767          no
  2   PRODUC         runtime  code   Wait Sema (h)  20             no
  3   CONSUM         runtime  code   Active         20             yes

  go
     *** Breakpoint ***         CONSUM(3).CONSUM    Statement 9

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 004B 004C 0043 0044 0045 0046 0047 0048  (.K.L.C.D.E.F.G.H)
 C96C (0010) 0049 004A 0003 0003                      (.I.J....        )

  go
     *** Breakpoint ***         CONSUM(3).CONSUM    Statement 9

  sc(message, 0, 24)go
     extra characters will be ignored
     common MESSAG
 C95C (0000) 004B 004C 0043 0044 0045 0046 0047 0048  (.K.L.C.D.E.F.G.H)
 C96C (0010) 0049 004A 0003 0004                      (.I.J....        )

  sc(message, 0, 24)
     common MESSAG
 C95C (0000) 004B 004C 0043 0044 0045 0046 0047 0048  (.K.L.C.D.E.F.G.H)
 C96C (0010) 0049 004A 0003 0004                      (.I.J....        )

  dab(consumer)

  go
     Idle Instruction

  dap
Status Summary of All Existing Processes

                     Site of                          Enabled    Stmt
     Process Name    Execution        Status      Pri Traces     Bkpts

  0  IDLE$P          IDLE$P    0     Active         32767          no
  2   PRODUC         runtime  code   Hold           20             no
  3.  CONSUM         runtime  code   Wait Sema      20             no

  sc(message, 0 , 24)
     common MESSAG
 C95C (0000) 004B 004C 0043 0044 0045 0046 0047 0048  (.K.L.C.D.E.F.G.H)
 C96C (0010) 0049 004A 0003 0003                      (.I.J....        )

  quit

Execution Terminated
Memory Used (bytes)  Maximum = 4082  Current = 2184
```

In the sample debugging session, the message buffer is displayed at various points using a "show common" (sc) command. Notice that a length of 24 bytes starting at displacement 0 is specified each time the common is displayed. From the compiler map, it can be seen that the first 20 bytes of the common comprise the message buffer slots, followed by the "next_in" and the "next_out" field. The semaphore values are not important to understanding this example.

Once the producer and consumer processes have been created, breakpoints are set just beyond the point where a message is produced or consumed. Since the producer and the consumer both perform a "swap", they manage to keep up with one another, i.e. as soon as an item is produced it is generally consumed immediately. To make the example more interesting, the consumer is held using a "hold process" (hp) command. The consumer becomes ineligible for execution until an explicit "release process" (rp) command is given. This causes the producer to completely fill the buffer until no more slots are available. When this happens, the producer is suspended on the "not_full" semaphore. At this point, the producer is held (using an hp command) and the consumer is released (using an rp command). This causes the consumer to consume all messages in the buffer. When all messages are consumed, the consumer is suspended on the "not_empty" semaphore. The example debugging session would continue forever if both processes were allowed to continue. The session can be terminated with a quit command.

Notice that a dangerous problem can occur if the wait(exclusive_access) precedes wait (not_empty) in the consumer process. Suppose the consumer process is started and becomes suspended on "not_empty" because no messages have yet been deposited into the buffer. A producer process then cannot get exclusive access to the buffer to deposit a message. The consumer will be waiting forever for an item to appear. In fact, all processes sharing the message buffer become suspended forever. Semaphores are low-level synchronization tools that must be used with great care. Users are therefore encouraged to use the mechanism of interprocess files for interprocess communication whenever possible, since this is a much safer, higher-level interface mechanism.

SECTION IV

SOURCE EDITOR GUIDE

4.1 SOURCE EDITOR OVERVIEW

The Source Editor allows the user interactively to create and modify Microprocessor Pascal source files which may be input to the Microprocessor Pascal Compiler. The Source Editor permits the user to enter and modify data only in the first 72 columns of a line. This protects the user from entering data which he had intended to be read by the compiler as part of his source program in columns 73 through 80. The Source Editor is invoked and operated from a TI 911(TI 913) VDT.

The Source Editor allows the user either to create a new source file, or to edit an existing source file. To create a new source file using the Source Editor, a blank line should be entered by pressing the RETURN(NEW LINE) key when prompted by the editor for an input file. In order to edit an existing source file, the user should respond to the prompt by typing the pathname of the file to be edited.

4.1.1 The Video Display

Editing occurs on a page basis, with a page being 23 lines on the 911 VDT and 11 lines on the 913 VDT. Any line displayed on the screen may be edited by positioning the cursor anywhere within the line to be edited. Lines may be inserted between any two lines, and may be inserted or deleted in any order. In addition, characters within a line may be inserted, deleted, or modified. Positioning of the file for display is accomplished by the use of the Roll Up, Roll Down, Cursor Up, and Cursor Down functions as well as the Relative Positioning, Top, Bottom, and Find commands.

4.1.2 Microprocessor Pascal Source File Definition

A Microprocessor Pascal source file is a file which is determined to be syntactically complete by the CHECK command. A Microprocessor Pascal source file may be:

> A module header with any portion of its declaration section and its associated body,
>
> The declaration section of a module with one or more of the declarations in the order Const, Type, Var, Common, Access, and Submodules.

4.1.3 Command Summary

The commands and functions of the Source Editor are conveniently divided into five separate classes. The following table is a summary of the editor commands and functions. A detailed description of each command and function is given in Section 4.3.

TABLE 4-1. SOURCE EDITOR COMMANDS AND FUNCTIONS

Command/Function	911 VDT Key	913 VDT Key
Setup and Termination		
Help	CMD/"HELP"	HELP/"HELP"
Edit/Compose Toggle	F7	F7
Syntax Check	CMD/"HELP"	HELP/"CHECK"
Quit	CMD/"QUIT"	HELP/"QUIT"
Abort	CMD/"ABORT"	HELP/"ABORT"
Save	CMD/"SAVE"	HELP/"SAVE"
Input	CMD/"INPUT"	HELP/"INPUT"
Cursor Positioning		
Roll Up	F1	ROLL UP
ROLL DOWN	F2	ROLL DOWN
NEW LINE	RETURN	NEW LINE
TAB	SHIFT TAB SKIP	TAB
BACK TAB	LEFT FIELD	BACK TAB
SET TAB INCREMENT	CMD/"TAB(increment)"	HELP/"TAB(increment)"
CURSOR UP	up-arrow	up-arrow
CURSOR DOWN	down-arrow	down-arrow
CURSOR RIGHT	right-arrow	right-arrow
CURSOR LEFT	left-arrow	left-arrow
Home	HOME	HOME
Find	CMD/"FIND(parms)"	HELP/"FIND(parms)"
Relative Positioning	CMD/number	HELP/number
Top	CMD/"TOP"	HELP/"TOP"
Bottom	CMD/"BOTTOM"	HELP/"BOTTOM"
Program Modification		
Insert Line	unlabeled gray key	INSERT LINE
Duplicate Line	F4	F4
Delete Line	ERASE INPUT	DELETE LINE
Skip	TAB SKIP	SKIP
Insert Character	INS CHAR	INS CHAR
Delete Character	DEL CHAR	DEL CHAR
Clear Line	ERASE FIELD	CLEAR
Replace	CMD/"REPLACE(parms)"	HELP/"REPLACE(parms)"
Split Line	F8	F0
Insert	CMD/"INSERT"	HELP/"INSERT"
Block Commands		
Start Block	F5	F5
End Block	F6	F6
Copy	CMD/"COPY"	HELP/"COPY"
Move	CMD/"MOVE"	HELP/"MOVE"
Delete	CMD/"DELETE"	HELP/"DELETE"
Put	CMD/"PUT"	HELP/"PUT"
Show Command		
Show	CMD/"SHOW"	HELP/"SHOW"

4.2 EXAMPLE EDIT SESSIONS

The following edit sessions provide examples depicting the creation of a new source file, modification of an existing source file, and saving the results of an edit session.

4.2.1 Creating a File

The following procedure applies to creating a new file using the Source Editor. The example assumes that the Source Editor has been invoked as described in Section XV covering the DX10, and Section XVI covering TXDS.

Upon invocation of the Source Editor, the user is prompted as follows:

```
    INPUT FILE ACCESS NAME:
```

Press the RETURN(NEW LINE) key to indicate that a new source file is to be created. The screen is cleared, *EOF is displayed in the upper left hand corner of the screen, COMPOSE MODE is displayed in the lower right hand corner of the screen, and the cursor is positioned at row one, column one of the screen. The display indicates that the only record in the file is the end-of-file record and that the editor is in COMPOSE mode.

To begin entering a program, press either the RETURN(NEW LINE) key or the unlabeled gray key(INSERT LINE). Note that row one is now a blank line, the end-of-file marker is on row two, and the cursor is positioned at row one, column one. You may now begin entering a source file by simply typing the source file and pressing the RETURN(NEW LINE) key whenever you wish to enter another line.

After the program has been typed in, the file may be saved by one of two methods:

> If the user does not want to edit another source file then he should press the CMD(HELP) key and type the word QUIT. The user will then be prompted as follows:
>
> OUTPUT FILE ACCESS NAME:
>
> The user should respond by typing the pathname of the file to which the edited source file is to be written and press the RETURN(NEW LINE) key. The user will then be prompted as follows:
>
> REPLACE?:
>
> The user should respond by typing the letter N to

specify that an existing file with the pathname entered in response to the previous prompt should not be replaced, otherwise the user should respond by typing the letter Y. The file is then saved in the file specified by the pathname entered and the editor is exited.

If the user wants to edit another file after saving the file just edited, the CMD(HELP) should be pressed, and the word SAVE typed. The user will then be prompted as follows:

OUTPUT FILE ACCESS NAME:

The user should respond by typing the pathname of the file to which the edited source file is to be written and press the RETURN(NEW LINE) key. The user will then be prompted as follows:

REPLACE?:

The user should respond by typing the letter N to specify that an existing file with the pathname entered in response to the previous prompt should not be replaced, otherwise the user should respond by typing the letter Y. The file is then saved in the file specified by the pathname entered and the user is prompted for the pathname of the next file to be edited as follows:

INPUT FILE ACCESS NAME: pathname

The user may then respond by pressing the RETURN(NEW LINE) key to re-edit the file just saved, clearing the line and pressing RETURN(NEW LINE) to create a new file, or entering the pathname of an existing file as described in the next section.

4.2.2 Editing an Existing File

The example described in this section gives the general procedures for editing an existing file by using the Source Editor. It is assumed that the editor has already been invoked. For this example session, the file in Figure 4-1 will be used as the input file.

Upon invocation the following is displayed:

INPUT FILE ACCESS NAME:

Enter the pathname of the file to be edited and press the RETURN(NEW LINE) key.

The screen is cleared and the file is displayed, beginning in row one of the screen. EDIT MODE is displayed in the lower right hand corner. *EOF is displayed on row 14, after the last line of the source file. The cursor is in column one, row one.

It may be noticed that the keyword "BEGIN" has been omitted. In order to insert the keyword, the user should press the down-arrow key three times and then press the unlabeled gray key(INSERT LINE), which inserts a blank line in front of line 4. The cursor is in column one of the blank line. The user then types "BEGIN".

At this point the user may think that the program is syntactically complete. To verify this he presses the CMD(HELP) key, types the word CHECK, and presses the RETURN(NEW LINE) key.

Shortly afterwards the message "SEMICOLON MAY NOT PRECEDE AN ELSE" is displayed on the bottom line of the screen and the cursor is positioned at the keyword "ELSE". The user presses the up-arrow key, presses the HOME key, and then presses the LEFT FIELD(BACK TAB) key three times to position the cursor on the semicolon following the THEN clause. To remove this semicolon, press the space bar, causing the semicolon to be replaced by a blank.

The user again wants to verify that his program is syntactically correct, so he again presses the CMD(HELP) key, reenters CHECK, and presses the RETURN(NEW LINE) key.

The error message "END-OF-FILE EXPECTED" is displayed on the bottom line of the screen and the cursor is positioned at the keyword "END" which immediately precedes the *EOF.

The user presses the ERASE INPUT(DELETE LINE) key to delete the line which contains the extra "END;" statement. He then presses the CMD(HELP) key, types the word CHECK, and presses the RETURN(NEW LINE) key.

After a short period of time, the message "NO SYNTAX ERRORS FOUND" is displayed on the bottom line of the screen.

The user then saves the file as described in the previous section.

Notice that the syntax checker did not detect that the variable "second" was not declared. This is an example of a semantic error, which is not detected by the syntax checker.

```
PROGRAM example;
VAR
  first : INTEGER;
  reset(INPUT);
  WHILE NOT eof DO
    BEGIN
      readln(first,second);
      IF first = second
        THEN writeln('EQUAL',first);
        ELSE writeln(first,' NOT EQUAL',second);
    END
END;
END.
```

FIGURE 4-1. INPUT FILE EXAMPLE

4.3 EDITOR COMMANDS AND FUNCTIONS

The following sections describe the commands and functions of the Source Editor. Commands may be entered in either upper-case or lower-case letters. They are divided into five groups:

1) Setup and Termination,

2) Cursor Positioning, and Program Modification

3) Block Commands

4) Show Command.

4.3.1 Kinds of Parameters

There are four basic kinds of parameters recognized by the editor. These are:

Integer Constant - An integer constant parameter is a non-negative number less than or equal to 32767

Identifier - An identifier parameter is either a Microprocessor Pascal identifier or Microprocessor Pascal System reserved word

String - A string parameter is a character string enclosed in double quotes (a double quote is represented by two double quotes inside a string)

Pathname - A pathname parameter is a valid DX10 or TXDS pathname, depending on which operating system is being used.

4.3.2 Optional Parameters

If a parameter is optional, it can simply be omitted, forcing the default value to be assumed.

Extra commas for optional parameters at the end of a command need not appear. For example, the command "FIND(Identifier)" is equivalent to "FIND(Identifier,1)".

4.3.3 Current Line Marker

The line on which the cursor is currently positioned is often significant while editing a file. The editor automatically marks the current line location when the CMD(HELP) key is pressed during an edit session by placing a "--------" in columns 73 through 80.

If the current line should be a line which already contains a start or end block marker in columns 73 through 80 (section 4.3.8.1), the current line marker becomes a "<------ " or a "------>", respectively. Should the current line already contain a start and end block marker there will be no change; a "<----->" will remain in columns 73 through 80.

4.3.4 CMD(HELP) Key

The CMD(HELP) key is used for several purposes within the editor. Generally, it will cause the editor to prompt the user for a command. It can also be used after an error has occurred to erase the error message generated from the screen and to prompt the user for the next command.

If the user is prompted for information after having entered a command, pressing the CMD(HELP) key will cause the editor to return to command mode. However, if the user presses the CMD(HELP) key in respond to the prompt INPUT FILE ACCESS NAME, the editor will abort.

4.3.5 Setup and Termination Commands

The functions and commands described in this section are used to set up the input file for the editor, specify the output pathname for the file after it has been modified, and set the mode of editing (i.e. compose or edit).

4.3.5.1 HELP Command. Pressing CMD(HELP) and typing "HELP" causes a list of available commands and short descriptions of each to be displayed. This command also displays the tab increment amount.

4.3.5.2 Edit/Compose Mode. The editor operates in either compose mode or edit mode. Compose mode is generally used to enter large blocks of new program text. By contrast, edit mode is most useful when modifying portions of existing program text.

The major difference between compose mode and edit mode is the function of the RETURN(NEW LINE) key. When operating in edit mode, pressing RETURN(NEW LINE) causes the cursor to move to the next line; in compose mode, pressing RETURN(NEW LINE) causes a blank line to be inserted after the current line. When creating a new file the editor is invoked in compose mode. If editing an existing file, the editor is invoked in edit mode. The user may switch back and forth between edit and compose mode by pressing a special function key, F7, which acts as a toggle switch. The mode of the editor is displayed in the lower right hand corner of the screen.

4.3.5.3 CHECK Command. This command instructs the editor to perform a syntax check of the module being edited and allows errors to be corrected interactively as they are found. This feature has the advantage of helping the user to detect common syntax errors before a (possibly) time-consuming compile step is attempted. The typical use of this command is when the edit session is nearly complete. If a syntax error is found, the editor positions the window and cursor to the point of the error, and allowing the user to correct it.

4.3.5.4 QUIT Command. The QUIT command is used to save the results of the most recent edit session and exit the editor. This command is entered by pressing CMD(HELP) followed by typing "QUIT", the following prompt is then displayed:

```
    OUTPUT FILE ACCESS NAME: pathname
```

The pathname displayed is the value entered when the current edit session was invoked. If no pathname is displayed, the user must enter the pathname of the output file. The pathname must be entered if the user wishes to save the results of the edit session on a different file (possibly a new file). To replace the file that is displayed in the prompt, the user simply presses the RETURN(NEW LINE) key. The user will then be prompted as follows:

```
    REPLACE?:
```

The user should respond by typing the letter N to specify that an existing file with the displayed pathname should not be replaced, otherwise the user should respond by typing the letter Y.

If a legal pathname is given and a valid replacement option is specified, the file will be "saved" in its final edited form. During the "saving", the following message will be displayed:

```
     "QUIT" COMMAND EXECUTING
```

Should either of these responses be incorrect, a file I/O error will occur and an appropriate error message will be generated.

If no pathname is specified when the RETURN(NEW LINE) key is pressed, no action is performed and the editor prompts for another command.

4.3.5.5 ABORT Command. The ABORT command is used to gracefully exit the editor without saving the results of the current edit session. This command is entered by pressing the CMD(HELP) key and typing the word ABORT.

4.3.5.6 SAVE Command. The SAVE command is used to save the results of the most recent edit session on and start editing a new file. This command is entered by pressing the CMD(HELP) key and typing "SAVE", the following prompt is then displayed:

```
      OUTPUT FILE ACCESS NAME: pathname
```

The pathname displayed is the value entered when the most recent edit session was invoked. If no pathname is displayed, the user must enter the name of the output file. The name must be entered if the user wishes to save the results of the edit session on a different file (possibly a new file). To replace the file that is displayed in the prompt, the user simply presses the RETURN(NEW LINE) key. The user will then be prompted as follows:

```
     REPLACE?:
```

The user should respond by typing the letter N to specify that an existing file with the displayed pathname should not be replaced, otherwise the user should respond by typing the letter Y.

If a legal pathname is given and a valid replacement option is specified, the file will be "saved" in its final edited form. During the "saving", the following message will be displayed:

```
     "SAVE" COMMAND EXECUTING
```

Should either of these responses be incorrect, a file I/O error will occur and an appropriate error message will be displayed.

If no pathname is specified when the RETURN(NEW LINE) key is pressed, no action is performed and the editor prompts for another command.

The user is then prompted for the pathname of the next file to be edited with the following prompt:

```
INPUT FILE ACCESS NAME:
```

The response to the prompt is the pathname of the file which is to be edited. If no pathname is specified when the RETURN(NEW LINE) key is pressed, a new file is created. The cursor is positioned at the first column of the first line of the file.

4.3.5.7 INPUT Command. The INPUT command is used to stop the editing of the current file without saving the results of the most recent edit session and begin the editing of another file. This command is entered by pressing CMD(HELP) followed by typing "INPUT". When the command is entered, the following prompt is displayed:

```
INPUT FILE ACCESS NAME:
```

The response to the prompt is the pathname of the file which is to be edited. If no pathname is specified when the RETURN(NEW LINE) key is pressed, a new file is created. The cursor is positioned at the first column of the first line of the file.

4.3.6 Cursor Positioning

The functions and commands described in this section are used to position the window and the location of the cursor within the window.

4.3.6.1 Roll-Up Function. The Roll-Up function is called by pressing the F1(ROLL UP) key. This function causes the cursor to be repositioned to column one of the line which is 23 (11 on the 913 VDT) lines toward the end of file from the current line. The row on which the cursor is positioned is not changed.

4.3.6.2 Roll- Down Function. The Roll Down function is called by pressing the F2(ROLL DOWN) key. This function causes the cursor to be repositioned to column one of the line which is 23 (11 on the 913 VDT) lines toward the top of file from the current line. The row on which the cursor is positioned is not changed unless the new line on which the cursor is positioned is the first line of the file, in which case the cursor is at row one.

4.3.6.3 New Line Function. The New Line function is called by pressing the RETURN(NEW LINE) key. In edit mode, this function causes the cursor to move to the first character of the first token on the next line. In compose mode, a blank line is inserted after the current line and the cursor is moved to the new line. The cursor is positioned on a new line at the same indentation level as the first token on the previous line. If the program text to be entered should start at a different nesting level, the tab and back tab functions can be used to indent or un-indent the cursor to the proper place.

4.3.6.4 Tab Function. The Tab function is called by pressing the SHIFT key and TAB SKIP key at the same time on the 911 VDT or the TAB key on the 913 VDT. If the cursor is positioned on a blank line, the cursor moves to the right one indentation level. Otherwise, the cursor moves to the start of the next token. If the cursor is at the last token, the cursor moves to the right one indentation level. If the cursor is at column 72, the cursor is positioned to the beginning of the line.

4.3.6.5 Back Tab Function. The Back Tab function is called by pressing the LEFT FIELD(BACK TAB) key. If there are no characters to the left of the cursor, the cursor moves to the left one indentation level. If the cursor is positioned to the right of the space following the last token on a line, the cursor moves to the space following the last token. Otherwise, the cursor moves to the start of the previous token. If the cursor is at the first token, the cursor moves to the left one indentation level. If the cursor is at the beginning of a line, the cursor is positioned to the end of the line.

4.3.6.6 Set Tab Increment Command. The Set Tab Increment command is used to change the increment amount used for tabs and back tabs. The syntax of this command is:

```
TAB(increment)
```

The increment value must be a positive integer value less than 72. The default increment value used by the editor for tabs is two.

4.3.6.7 Cursor Up Function. The Cursor Up function is called by pressing the up-arrow key. This function causes the cursor to move to the previous line. The cursor remains at the same position within the line.

4.3.6.8 Cursor Down Function. The Cursor Down function is called by pressing the down-arrow key. This function causes the cursor to be moved to the next line. The cursor remains at the same position within the line.

4.3.6.9 Cursor Right Function. The Cursor Right function is called by pressing the right-arrow key. This function causes the cursor to be moved one position to the right. If the cursor is in column 72 on a line when the cursor right function is called, the cursor remains in its current position.

4.3.6.10 Cursor Left Function. The Cursor Left function is called by pressing the left-arrow key. This function causes the cursor to be moved one position to the left. If the cursor is in column one of the line when the cursor left function is called, the cursor remains in its current position.

4.3.6.11 Home Function. The Home function is called by pressing the HOME key. This function causes the cursor to be moved to column one of the current line.

4.3.6.12 FIND Command. The FIND command is used to position the cursor to the next (or nth) occurrence of a specific identifier or string after the current cursor position. The command is entered by pressing the CMD(HELP) key followed by typing a command with the syntax:

```
    FIND (identifier or string, occurrence number)
```

If the occurrence number is not specified, it is assumed to be one (i.e. the next occurrence is searched for). The search begins at the first character following the cursor. If the specified number of occurrences of the identifier or string is found the cursor is positioned so that it is on the first character of the last occurrence of the identifier or string, and the line in which the identifier or string is contained is in the center row of the screen. If the specified number of matches for the name or string is not found, the cursor position remains unchanged and a message is displayed which indicates that the identifier or string was not found the specified number of times.

4.3.6.13 Relative Positioning. The file may be positioned an arbitrary number of lines relative to the current position by first pressing the CMD(HELP) key followed by the integral number of lines to be skipped, either forward or backward. If the jump is to go forward the specified integer is preceded by an optional + (plus sign); for a backward skip, the integer must be preceded by a - (minus sign). If the specified jump is outside the file boundaries, the cursor is positioned to the file's beginning or end, depending on the direction of the jump. The cursor is positioned at column one of the destination line. If the line is the end-of-file then the line is displayed as the last line on the screen. Otherwise, the line is displayed at the center of the screen.

4.3.6.14 TOP Command. The TOP command is used to position the cursor to the first column of the first line of the file being edited. The command is entered by pressing CMD(HELP) followed by typing "TOP". The cursor is positioned at row one, column one.

4.3.6.15 BOTTOM Command. The BOTTOM command is used to position the cursor to the end-of-file marker of the file being edited. The command is entered by pressing CMD(HELP) followed by typing "BOTTOM". The cursor is positioned at column one of the end-of-file marker which is displayed as the last line on the screen.

4.3.7 Program Modification

The functions and commands described in this section are used to modify source files. Any time that a line is modified, any data which may be in columns 73 through 80 is replaced by blanks to indicate to the user that the line has been modified. This deletion of characters from columns 73 through 80 does not effect the program being entered, since only columns 1 through 72 are used by the compiler.

4.3.7.1 Insert Line Function. The Insert Line function is called by pressing the unlabeled gray key(INSERT LINE). When this function is called, a blank line is inserted just before the line the cursor is on.

4.3.7.2 Duplicate Line Function. The Duplicate Line function is called by pressing the F4 key. This function causes a copy of the characters from the cursor position to the end of the line to be duplicated and placed after the current line. The cursor is moved to the new line and remains in the same column.

4.3.7.3 Delete Line Function. The Delete Line function is called by pressing the ERASE INPUT(DELETE LINE) key. This function causes the line on which the cursor is positioned to be deleted. The cursor is positioned at the first character of the first token of the line which followed the deleted line.

4.3.7.4 Skip Function. The Skip function is called by pressing the TAB SKIP(SKIP) key. This function clears all of the characters on the current line from the cursor position to the right margin. The cursor position is not changed.

4.3.7.5 Insert Character Function. The Insert Character function is called by pressing the INS CHAR key and typing the new character(s) which are inserted into the file. Characters are never lost at the right margin. If a non-blank character is present at the right margin, no additional characters can be inserted on the line. The beeper is sounded if this is attempted. Note that a "split line" command exists for breaking up long lines.

4.3.7.6 Delete Character Function. The Delete Chracter function is called by pressing the DEL CHAR key. This function causes the character at the current cursor position to be deleted and the characters to the right of the cursor position to be shifted to the left one character position with a blank being inserted in column 72.

4.3.7.7 Clear Line Function. The Clear Line function is called by pressing the ERASE FIELD(CLEAR) key. This function causes the line containing the cursor to be cleared. The cursor is repositioned to the beginning of the line.

4.3.7.8 REPLACE Command. The REPLACE command searches for the next n occurrences of an identifier (or string) and when found, substitutes a different identifier (or string). The syntax of this command is:

```
    REPLACE (pattern1, pattern2, repeat count)
```

where pattern1 and pattern2 are either an identifier or string enclosed in double quotes.

If the specified command is executed without an error, the cursor is positioned at the first character of the last occurrence of pattern2 which was replaced. If pattern1 is not found the specified number of times, the cursor is returned to the line and column at which it was positioned prior to the execution of the command and a message is displayed which indicates the number of occurrences of pattern1 which were not replaced. If the replacing of an occurrence of pattern1 by pattern2 would result in characters being lost off the end of a line, the command is halted and the cursor is positioned at the beginning of the occurrence of pattern1 which caused the halt.

4.3.7.9 Split Line Function. The Split Line function is called by pressing the F8(FO) key. This function causes the current line to be split into two lines such that the cursor position indicates the first character of the first token of the new line. The first character of the new line is positioned at the same indentation level as the first token of the line which was split. The cursor position is not changed.

4.3.7.10 INSERT Command. The INSERT command copies a sequential file, other than the file being edited, to the position after the line at which the cursor is positioned. After the INSERT command has been entered, the user will be prompted for a file name:

```
      INSERT FILE ACCESS NAME:
```

Enter the pathname of the file to be inserted and press the *RETURN*(NEW LINE) key. The entire file specified will be copied into the file being edited.

4.3.8 Block Commands

The functions and commands described in this section are used to modify files by using designated blocks of lines instead of single characters, or single lines.

4.3.8.1 Start and End Block Functions. These two functions are used to bracket sections of the file to be manipulated by the COPY, MOVE, DELETE, and PUT commands by placing markers in the file. A start block marker is set by pressing the F5 key and results in a beep and the placing of "<-------" in columns 73 through 80. An end block marker is set by pressing the F6 key and results in a beep and the placing of "------->" in columns 73 through 80. If both the start and end block markers are set on the same line of the file "<------>" is placed in columns 73 through 80 of that line. For either function, the cursor position is used as the location for the marker.

4.3.8.2 COPY Command. The COPY command causes a copy of the block designated by the start and end block markers to be made after the line on which the cursor is positioned. When this command is completed, the markers are not modified and the cursor is placed in column 1 of the first line of the copied block. If the end block marker precedes the start block marker, or either of the markers does not exist, a message is displayed and no action is taken.

4.3.8.3 MOVE Command. The MOVE command causes the block designated by the start and end block markers to be moved after the line on which the cursor is positioned. When this command is completed, the markers are removed from the file and the cursor is placed in column 1 of the first line of the moved block. The designated block can not be moved to a location which is contained within that block. If the end block marker precedes the start block marker, or either of the markers does not exist, a message is displayed and no action is taken.

4.3.8.4 DELETE Command. The DELETE command causes the block designated by the start and end block markers to be deleted. When this command is completed, the markers are removed from the file and the cursor is placed in column 1 of the line following the deleted block. If the end block marker precedes the start block marker, or either of the markers does not exist, a message is displayed and no action is taken.

4.3.8.5 PUT Command. The PUT command causes a copy of the block designated by the start and end block markers to be copied to the file specified. After the PUT command has been entered, the user will be prompted for the file pathname to which the block is to be copied.

```
OUTPUT FILE ACCESS NAME:
```

The user should respond by typing the pathname of the file to which the block is to be written and press the RETURN(NEW LINE) key. The user will then be prompted as follows:

```
REPLACE?:
```

The user should respond by typing the letter N to specify that an existing file with the entered pathname should not be replaced, otherwise the user should respond by typing the letter Y.

If a legal pathname is given and a valid replacement option is specified, the designated block will be "put" into the file in its current form. The following message will be displayed while the command is completing its execution:

"PUT" COMMAND EXECUTING

Should either of these responses be incorrect, a file I/O error will occur and an appropriate error message will be displayed.

If no pathname is specified when the RETURN(NEW LINE) key is pressed, no action is performed and the editor prompts for another command.

4.3.9 SHOW Command

The SHOW command allows the user to "look" at a sequential file, other than the one being edited, during an edit session. After the SHOW command has been entered, the following prompt will be displayed requesting the pathname of the file to be "shown":

SHOW FILE ACCESS NAME:

The file may be positioned an arbitrary number of lines relative to the current position by first pressing the CMD(HELP) key followed by the integral number of lines to be skipped, either forward or backward. If the jump is to go forward the specified integer is preceded by an optional "+"; for a backward skip, the integer must be preceded by a "-". If the specified jump is outside the file boundaries, the cursor is positioned to file's beginning or end, depending on the direction of the jump.

4.4 ERROR MESSAGES

This section describes each error message generated by the command processor and syntax checker.

4.4.1 Command Syntax Errors

When a command or a parameter of a command is improperly formed or recognized by the Source Editor, one of the following error messages is given.

BAD PARAMETER
An illegal parameter was found within a command. Parameters can only be one of the following: integer constant, identifier, string (delimited by double quotes), or pathname.

INCOMPLETE COMMAND SYNTAX
A command is improperly terminated. If a command has parameters, the parameter list must be enclosed in parentheses.

INVALID COMMAND NAME
The command name is not valid. Use the HELP command to find the proper command name to use.

EXTRANEOUS CHARACTERS
The command contains extra non-blank characters to the right of an otherwise well-formed command.

TOO MANY PARAMETERS
The command contains too many parameters. Use the HELP command to check the number and meaning of parameters for the command.

4.4.2 Command Processing Errors

The following error messages may be generated by a command which is being executed.

n OCCURENCE(S) NOT FOUND
The identifier or string specified in a FIND or REPLACE command was not found the specified number of times between the current cursor position and the end-of-file marker.

REPLACEMENT STRING TOO LONG
If replacement of a string or name in a line would cause characters to be lost off the right hand side of a line, this message is displayed.

RESPONSE MUST BE "YES" OR "NO"
The response given to the REPLACE?: prompt must be a yes (y) or a no (n).

START BLOCK NOT SPECIFIED
A COPY, MOVE, DELETE, or PUT command was entered but the designated block was not completely bracketed.

END BLOCK NOT SPECIFIED
A COPY, MOVE, DELETE, or PUT command was entered but the designated block was not completely bracketed.

END BLOCK PRECEDES START BLOCK
Within the file being edited, the start block marker must precede the end block marker for a MOVE, COPY, DELETE, or PUT command to be executed.

ILLEGAL DESTINATION SPECIFIED
The designated block in a MOVE command can not be moved to any location which is within that block of the file.

4.4.3 File I/O Errors

The following errors may be generated when responding to a prompt from the editor for an input, or output file access name.

SVC ERROR NO. n

This error may be generated when responding to the editor's prompt for an input file access name at the beginning of an edit session, or following a SAVE or INPUT command. This error occurs if the specified file can not be accessed. The SVC status code is given, in hexadecimal, to further clarify the error encountered. The meanings associated with each of these codes can be found in either the TX 990 Operating System Programmer's Manual, or in the DX 10 Operating System Reference Manual - VOL.6 Error Reporting and Recovery.

BAD DISK NAME/DISK VOLUME NOT INSTALLED

This error occurs when the disk name (within a file access name) given as a response to an editor prompt does not exist..

NO FILE DEFINED BY NAME SPECIFIED

This error occurs when the file access name given by the user in response to an editor prompt does not exist.

FILE IS DELETE PROTECTED

This error will occur if the user attempts to save a file used in an edit session in an existing file which is delete protected.

FILE EXISTS AND REPLACE NOT SPECIFIED

This error will occur when the user requests that a file be saved, but not replaced and a file of that name already exists.

BAD PATHNAME SYNTAX

This error occurs when the syntax of the file access name entered is invalid.

UNABLE TO GRANT REQUESTED ACCESS PRIVILEGES

This error will occur if the user has requested a file in response to a prompt which can not be accessed by the editor, e.g. it is already in use.

4.4.4 Syntax Checker Error Messages

The following is a list of error messages which will be generated by the syntax checking routine of the editor. The list includes the actual error message and a brief description of what could have caused the error.

1 STATEMENT SEPARATOR EXPECTED

A statement must be separated by ";", "END", or "UNTIL".

2 MISMATCHED PARENTHESES

Parentheses do not match in an expression, declaration, or parameter list.

3 " " EXPECTED

A " " was expected following a set reference, or an array subscript.

4 INVALID OPERAND IN EXPRESSION

An invalid term was encountered in an expression.

5 ERROR IN QUALIFIED VARIABLE

An identifier must follow the "." of a qualified variable.

6 ERROR IN TYPE TRANSFER VARIABLE

A TYPE identifier must follow the "::" of a type transfer variable.

7 CASE ALTERNATIVE ERROR

A CASE label, ";", "END", or "OTHERWISE" was expected.

8 "OF" EXPECTED IN CASE STATEMENT

Incomplete CASE statement found; "OF" must precede the included list of case alternatives.

9 MISMATCHED REPEAT/UNTIL PAIR

An "UNTIL" was not expected to occur at this point in the system.

10 SEMICOLON MAY NOT PRECEDE AN "ELSE"

The THEN and the ELSE clauses of an IF statement may not be separated by a semicolon.

11 THEN EXPECTED

An IF statement is incomplete without a THEN clause.

12 ":" EXPECTED AFTER LABEL

All statement labels must be followed by a ":".

13 STRUCTURED STATEMENT MUST FOLLOW ESCAPE LABEL

A REPEAT, WHILE, WITH, FOR, IF, CASE, or compound statement must follow all escape labels.

14 ":=" EXPECTED IN ASSIGNMENT STATEMENT

An invalid operator, or operand was encountered in an assignment statement.

15 ERROR IN WRITE PARAMETER LIST

A "," was expected or the keyword "HEX" was expected in a write parameter list.

16 ESCAPE IDENTIFIER EXPECTED

The keyword "ESCAPE" must be followed by an escape label.

17 STATEMENT LABEL EXPECTED

The keyword "GOTO" must be followed by a statement label.

18 PROGRAM OR PROCESS NAME MUST FOLLOW START
A START statement must include a PROCESS, or PROGRAM identifier following the keyword "START".

19 CONTROL VARIABLE EXPECTED
The control variable of a FOR statement was expected following the keyword "FOR".

20 ":=" EXPECTED IN FOR STATEMENT
A FOR statement control variable must be followed by a ":=".

21 "TO" OR "DOWNTO" EXPECTED IN FOR STATEMENT
A "TO" or "DOWNTO" must separate the initial and final expressions of a FOR statement.

22 "DO" EXPECTED IN FOR, WITH, OR WHILE STATEMENT
A "DO" must be included in all FOR, WITH, and WHILE statements.

23 INVALID TAGFIELD IN WITH STATEMENT
A record variable or an identifier was expected in the tagfield of a WITH statement.

24 STATEMENT EXPECTED
An unknown keyword, or statement beginning was encountered.

25 ":" EXPECTED AFTER CASE LABEL LIST
A ":" must follow all CASE label lists.

26 INVALID CASE LABEL
An enumeration constant was expected as a CASE label.

27 DECLARATION SEPARATOR EXPECTED (";")
All declarations must be separated by ";".

40 ERROR IN LABEL LIST
A statement label was expected in a LABEL declaration.

41 "=" EXPECTED IN TYPE OR CONST DECLARATION
An "=" must follow all TYPE and CONST identifiers that are being declared.

42 CONST IDENTIFIER EXPECTED
An identifier was expected in a CONST declaration.

43 TYPE IDENTIFIER EXPECTED
An identifier was expected in a TYPE declaration.

44 ":" EXPECTED IN VAR OR COMMON DECLARATION
A ":" must follow all VAR and COMMON identifiers that are being declared.

45 VAR IDENTIFIER EXPECTED

An identifier was expected in a VAR declaration.

46 COMMON IDENTIFIER EXPECTED

An identifier was expected in a COMMON declaration.

47 INVALID OPERAND IN CONST DECLARATION

An integer term was expected in a CONST expression.

48 " " EXPECTED IN ARRAY DECLARATION

A " " must precede the index type(s) of all ARRAY declarations.

49 "OF" EXPECTED IN DECLARATION

An "OF" was expected in an ARRAY, FILE, SET, or RECORD variant declaration.

50 "END" EXPECTED FOLLOWING RECORD DEFINITION

An "END" was expected to terminate a RECORD declaration.

51 "ARRAY" OR "RECORD" MUST FOLLOW "PACKED"

PACKED structures only include ARRAYs and RECORDs.

52 "FILE" MUST FOLLOW "RANDOM"

A RANDOM file declaration must include the keyword "FILE" following the "RANDOM" specification.

53 ":" EXPECTED IN RECORD FIELD LIST

A ":" must separate all identifiers from the TYPE identifier with which they are associated.

54 INVALID TAGFIELD IN RECORD

A tagfield type was expected in the variant portion of a RECORD declaration.

55 "(" EXPECTED PRECEDING FIELD LIST

A "(" was expected in the variant portion of a RECORD declaration.

56 ".." EXPECTED IN DECLARATION

A ".." was expected in a subrange declaration.

57 ENUMERATION CONSTANT EXPECTED

An enumeration constant was expected in the declaration section.

58 INDEX TYPE EXPECTED IN DECLARATION

An index type was expected in an ARRAY declaration.

59 SIMPLE TYPE EXPECTED IN DECLARATION

A simple type was expected in a TYPE declaration, or in a SET declaration.

60 ERROR IN IDENTIFIER LIST
An identifier was expected in an identifier list.

61 PARAMETER LIST EXPECTED
A "(" was expected following a WRITE, ENCODE, or DECODE procedure call.

70 FILE MUST BEGIN WITH MODULE OR DECLARATIONS
The file being edited does not begin with an acceptable keyword.

71 MODULE DECLARATION SECTION EXPECTED
The module header has been encountered and parsed; declarations are expected next. Possibly a "FORWARD" or "EXTERNAL" is expected.

72 SYSTEM MUST BE OUTERMOST MODULE
A SYSTEM may not occur within any module.

73 MODULE HEADER MISSING
A body has been encountered, but the corresponding module header was missing.

74 MODULE EXPECTED
The end-of-file, or a module header is expected.

75 "END" NOT EXPECTED
An "END" was encountered, but not expected in a REPEAT statement.

76 END-OF-FILE EXPECTED
The parser has completed an entire system, but the file has not been exhausted.

77 MODULE IDENTIFIER EXPECTED
The name of the module must immediately follow the keyword "SYSTEM", "PROGRAM", "PROCESS", "PROCEDURE", or "FUNCTION" in a module header.

78 FUNCTION RESULT TYPE EXPECTED
The FUNCTION header is not complete without the result type of the FUNCTION included.

79 ":" EXPECTED IN PARAMETER LIST
A ":" must separate all parameters from the TYPE identifier with which they are associated in all parameter lists.

80 "BEGIN" EXPECTED
A "BEGIN" is expected to precede a module body section.

81 INVALID MODULE TERMINATOR (";" or ".")
The terminator following a module is missing, or an incorrect terminator was encountered.

82 SYSTEM MAY NOT HAVE PARAMETERS
A "(" was encountered following a SYSTEM identifier; parameter lists are not allowed at the system level.

90 SYSTEM NESTING LEVEL TOO DEEP FOR PARSER
The nesting within the file being edited is too deep to be handled by the CHECK command.

91 INVALID ?COPY STATEMENT
A ?COPY statement was encountered, but it is syntactically incorrect.

92 END OF STRING EXPECTED
AN "'" was expected to terminate a string within file.

93 END OF COMMENT EXPECTED
A "}" or "*)" was expected to terminate a comment.

94 NESTED COMMENTS ENCOUNTERED
A nested comment was encountered; comments should not be nested.

95 INVALID NUMBER
A symbol was encountered within a number which is not allowed in the type of number found. It could be a "." within an integer, or a hexadecimal digit within a real number.

1001 - 1006 INTERNAL PARSER ERROR
These errors should never be generated by the editor during its syntax check. If one should occur, recheck your system using the CHECK command. If the problem persists, send a listing of the file, along with some indication as to the location of the cursor when the error was generated, to your nearest TI representative.

SECTION V

COMPILER AND NATIVE CODE GENERATOR

5.1 COMPILER AND NATIVE CODE GENERATOR OVERVIEW

The Microprocessor Pascal Compiler takes, as input, the source for a Microprocessor Pascal system and produces Interpretive Code for a hypothetical stack computer. Normally only the first 72 columns of each input line are scanned and columns 73 through 80 are assumed to be the sequence field. This may be changed via a compiler option so that the entire source line will be scanned. The compiler checks for syntax and semantic errors while it is producing code, and it generates error messages for any errors it detects. The code generated by the compiler may be executed interpretively using the Host Debugger or it may be input to the Native Code Generator to generate 9900 native code. The interpretive code is characterized by its small size, about half the size of 9900 native code. Another advantage of interpretive code is the minimal time required to produce code which can be executed.

The Native Code Generator generates 9900 native code from the interpretive code produced by the compiler. If native code is to be produced, the OBJECT option must be specified so that additional information can be included in the interpretive code for use by the Native Code Generator. This same interpretive code may be executed by the Host Debugger, but it will be larger than it needs to be.

5.2 LISTING EXAMPLES

This section describes the listings generated by the compiler. An example of a normal error free listing is given, and also an example of a program with errors.

5.2.1 Compiler Execution Messages

As the compiler executes, messages are output to a file which is normally the user's interactive display. These messages indicate how much of the system has been compiled. An example of this file follows:

```
COMPILER  EXECUTION BEGINS
SCANNER IS FINISHED
FACTORIA
EXAMPLE1
NO ERRORS IN COMPILATION
NORMAL TERMINATION
STACK USED =   2706  HEAP USED =  2672
```

The first line and the last two lines are generated by the DX run-time support system and not by the compiler. They indicate that execution has begun, that the compiler terminated normally, and how much memory in bytes was used for the compilation.

The other lines are generated by the compiler. Since the compiler executes in two passes, a scanner and a parser , the first message indicates that the first pass is completed. As the compilation of each module is completed, the first eight characters of the name module is output. The last line indicates that no errors were found during compilation.

If the compiler finds errors in the system being compiled, this is indicated in the message file. For each module which has errors, the message "ERRORS IN MODULE" is output before the name of the module. Also the last line indicates that there were fatal errors in the compilation. If only non-fatal errors were found in the compilation, the last line output will be "ERRORS IN COMPILATION". An example of this type of message file is given below:

```
COMPILER  EXECUTION BEGINS
SCANNER IS FINISHED
ERRORS IN MODULE
ERROR
ERRORS IN MODULE
EXAMPLE2
FATAL ERRORS IN COMPILATION
NORMAL TERMINATION
STACK USED =    2532  HEAP USED =   2722
```

5.2.2 Compiler Listing

The compiler listing is produced by the second pass of the compiler. An example of this listing is shown on the following page.

```
DX Microprocessor Pascal Compiler   1.0     08/24/79 08:21:30

   0 PROGRAM EXAMPLE1;
   0 VAR N : INTEGER;
   2     M : INTEGER;
   0 FUNCTION FACTORIAL(I:INTEGER) : LONGINT;
   2 BEGIN
   1   IF I  = 1
   2     THEN FACTORIAL := 1
   3     ELSE FACTORIAL := I * FACTORIAL(I-1);
   4 END;
   1 BEGIN (* EXAMPLE1 *)
   1   N := 5;
   2
   2   M := FACTORIAL(N);
   3
   3   WRITELN(N:2,' FACTORIAL = ',M);
   4   WRITELN('NORMAL PROGRAM TERMINATION');
   5 END. (* EXAMPLE1 *)
```

The first line identifies the compiler, version of compiler, and date and time that the compilation occurred. The listing of the source is given next. If the "LIST" option is on (the default), the source is listed. Any lines between a "NO LIST" option and the next "LIST" option are not listed unless they contain errors.

Each line listed contains a number first, followed by the first 72 columns of the source line. In the declaration section, the number on the left-hand side is the displacement, in bytes, of the first variable declared on that line. In the body section, this number is the statement number of the first statement which appears on that line. Both of these numbers are helpful when debugging because variable displacements and statement numbers are required in many of the commands.

Syntax and semantic errors are detected by the compiler. The numeric values of these errors are included in the output listing directly below the point in the source listing where they were encountered. The meanings associated with these numeric values are listed in Section 5.7.1. A series of four asterisks are placed in the left hand margin below the line with the error so that errors can be found quickly. Then a "!" is placed below the token which was found to be incorrect followed by the number associated with the error. One error may cause several error messages which are separated by commas, but generally the first error is the correct one. Sometimes a token was expected to terminate the previous line but was not found, such as a semicolon, in which case the error will appear on the first token on the next line.

An example of a listing with error messages is shown below:

```
 DX Microprocessor Pascal Compiler  1.0   05/09/79 08:53:26

   0 PROGRAM EXAMPLE2;
   0 CONST TWO = 2;
   0       TEN = 10;
   0 VAR   X,XTWO,XTEN : REAL;
  12       RESULT      : REAL;
   0 PROCEDURE ERROR(ERR:INTEGER);
   1 BEGIN
   1   IF ERR    0
   2     THEN WRITELN('ERROR ##',ERR:2);
   3     ELSE WRITELN('NO ERRORS - NORMAL TERMINATION');
****     !41
   3 END;
   1 BEGIN
   1   X := 133.726;
   2   XTWO := X * TWO
   3   XTEN := X * TEN;
****   !14
   3
   3   RESULT := X/((X/XTWO) * XTEN/5.0);
   4
   4   IF RESULT  0
   5     THEN ERROR(1)
   6     ELSE IF RESULT = 0
   7            THENERROR(2)
****            !52      !104
   8            ELSE ERROR(0);
   9 END. (* EXAMPLE2 *)
```

5.2.3 Variable Map

This section describes the listing generated by the MAP compiler option. This option must be turned on or off for the entire compilation. This listing is produced after the complete system has been compiled, and it appears after the source listing. The map is produced in the order in which the declarations appear, that is, the outer blocks are listed before inner blocks. In all cases only the first eight characters of the each name are listed, all displacements are given in hexadecimal bytes, and all sizes unless otherwise stated are given in decimal bytes.

An example of the listing produced by the compiler when the MAP option was specified is shown below:

```
 DX Microprocessor Pascal Compiler      1.0    06/11/79 12:54:17

   0 (*$MAP *)
   0 SYSTEM MAP_EXAMPLE;
   0
   0 TYPE PTR = @ REC;
   0
   0      REC = RECORD
   0          A : INTEGER;
   0          S : SEMAPHORE;
   0          NEXT : PTR;
   0          END;
   0
   0      PREC = PACKED RECORD
   0          A : 0..255;
   0          B : BOOLEAN;
   0          C : CHAR;
   0          D : -128..127;
   0          END;
   0
   0 COMMON COM1 : REC;
   0     COM2 : PREC;
   0
   0 PROCEDURE INITSEMAPHORE(VAR SEMA : SEMAPHORE; COUNT : INTEGER);
   4     EXTERNAL;
   0 PROCEDURE SIGNAL(SEMA : SEMAPHORE); EXTERNAL;
   0 PROCEDURE WAIT(SEMA : SEMAPHORE); EXTERNAL;
   2
   0 PROGRAM PROG_EXAMPLE(OUTPUT, INPUT : TEXT);
   4 VAR P : @ REC;
   6     R : REC;
  12     S : PREC;
  16     I : INTEGER;
  18
   0 PROCEDURE PRINT_ERROR(N : INTEGER);
   1 BEGIN                      " body for PRINT_ERROR
   1 END;
   2
   1 BEGIN                      " body for PROG_EXAMPLE
   1 END;
   2
   1 BEGIN                      " body for MAP_EXAMPLE
   1 END.

SYSTEM MAP_EXAM;
    STACK SIZE = 0000

          COMMON     TYPE      SIZE
          COM1       RECORD    6
          COM2       RECORD    4
          FIELD     DISP      TYPE      SIZE
          A         0000      INTEGER   2
          S         0002      SEMAPHORE 2
          NEXT      0004      POINTER   2

          FIELD     DISP      TYPE      SIZE
          A         0000      SUBRANGE  8 BITS    (XXXXXXXX........)
          B         0000      BOOLEAN   1 BIT     (...............X)
          C         0002      CHAR      8 BITS    (XXXXXXXX........)
          D         0002      SUBRANGE  8 BITS    (........XXXXXXXX)
PROCEDURE INITSEMA ( VAR SEMA    :SEMAPHORE; COUNT   :INTEGER); EXTERNAL;

PROCEDURE SIGNAL   (     SEMA    :SEMAPHORE); EXTERNAL;

PROCEDURE WAIT     (     SEMA    :SEMAPHORE); EXTERNAL;

PROGRAM PROG_EXA ( OUTPUT  :FILE; INPUT   :FILE);
    STACK SIZE = 0012

          VARIABLE  DISP      TYPE      SIZE
          OUTPUT    0000      FILE      2
          INPUT     0002      FILE      2
          P         0004      POINTER   2
          R         0006      RECORD    6
          S         000C      RECORD    4
          I         0010      INTEGER   2

PROCEDURE PRINT_ER ( N         :INTEGER);
    STACK SIZE = 0002

          VARIABLE  DISP      TYPE      SIZE
          N         0000      INTEGER   2
```

Each section starts with a module header which both identifies the name and indicates whether it is a system, program, process, procedure, or function. If the module has any parameters, they are listed after the name surrounded by parenthesis. For each parameter, the name and type classification are given. If the parameter is a reference parameter, then the parameter is preceded by the keyword VAR. If the module is a function, the result type is given. If the module is external, EXTERNAL follows the module header.

The stack frame size in bytes is given following the header of a non external module.

The variable section is listed next. Parameters are also included in the variable section. For each variable, its name, displacement in bytes in the stack frame, type, and size in bytes is given. If the module has no variables, this section is missing.

The common section is listed next. For each common, its name, type, and size in bytes is given. Again if the module does not declare any commons, this section is missing.

The record section is listed last. The information presented for each record field includes its name, displacement in bytes from the beginning of the record, type, size in bytes or bits if packed, and the bit map which the packed field occupies. If the field is not packed, the last column is empty. Each record is separated from the others by a header. If no records are declared in the module, this section is missing.

5.2.4 Native Code Generator Listing

This section will describe the listings produced by the Native Code Generator. This will include the statement map cross reference which gives the displacement in the object module of the beginning of each statement in the module. The Native Code Generator is not supported in the current release, but will be added in a later release.

5.3 OPTIONS

Options are Boolean objects, each of which may have the value TRUE or FALSE independent of the values of all other options. Options are specified in a special form of a comment shown below:

```
                (*$ option list *)
or
                {$ option list}
```

Upon textual entry to a new Pascal routine, the values of all options are saved, but not changed. Since blocks may be nested, these values are stacked. Within a block options may be changed, subject to certain restrictions. Upon textual exit of the Pascal block, the values of all options are restored to the value they had upon block entry.

Option names may be preceded by NO or RESUME. The presence of an option name, without a prefix of NO or RESUME, in an option control comment causes the value of that option to become TRUE (subject to restrictions discussed below). If the option identifier appears with the prefix NO, the option's value becomes FALSE. If the option name is prefixed with RESUME, the value is set to the value the option had upon entry of the smallest enclosing block's scope. Notice that RESUME is not the same as "pop" because resume doesn't "pop" the stack (e.g. (*$RESUME LIST, RESUME LIST*) has exactly the same effect as (*$RESUME LIST*)).

Although option control comments may appear anywhere that a comment may appear, not all of the options may be controlled at any point in a system. "System sensitive" options must have a single value for the entire compilation. Since default values exist for all options in the imaginary scope in which the system is embedded, control of these options must be done before the system's text is entered. Thus, these options must appear only in option control comments located before the keyword SYSTEM, or before the keyword PROGRAM for a conventional Pascal program. The only options in this class are MAP and OBJECT.

"Routine sensitive" options have a single value for the entire statement part of any routine. Options in this class may be changed at three different places in a routine: before the beginning of the system, between the semicolon ending the routine header and the next keyword or symbol, and immediately after the BEGIN which starts the routine's statement part and before any portion of the first statement following the BEGIN. The options in this class are DEBUG, NULLBODY, STATMAP, and TRACEBACK. The remaining class of options may legally be set to new values at any point in a system where a comment could occur. (Refer to Table 5-1 below.)

TABLE 5-1. LISTING CONTROL OPTIONS

OPTION	DEFAULT	MEANING
COL72	TRUE	When this option is turned off, the entire source line is scanned, otherwise only 72 columns of the source are scanned. This option does not obey the normal scope rules so it must be explicitly turn on and off when desired. This option only applies to the line on which the option appears. (INSENSITIVE)
LIST	TRUE	This option controls the source listing. Lines with errors are always listed with informative error messages. (INSENSITIVE)
MAP	FALSE	This option indicates that a map of the system modules and variables are desired. The map listing is described in section 5.2.3. (SYSTEM SENSITIVE)
PAGE	FALSE	This option has the immediate effect of causing the next line to be printed at the top of the next page. The option is turned off immediately following the line. (INSENSITIVE)
STATMAP	FALSE	This option indicates that a map of the displacements for each statement in the object module is to be generated by the Native Code Generator. (ROUTINE SENSITIVE)

TABLE 5-1. OPTIONS (CONTINUED)

OPTION	DEFAULT	MEANING
		Native Code Options
OBJECT	FALSE	This option indicates whether native code is to be generated by the Native Code Generator. If native code is to be generated, this option must be turned on. If only interpretive code is to be generated, this option should be turned off because the code produced will be smaller in size. (SYSTEM SENSITIVE)
DEBUG	FALSE	This option should be used when the Host Debugger is to be used. The interpretive code is instrumented with statement numbers so that the module may be debugged. (ROUTINE SENSITIVE)
TRACEBACK	FALSE	This option is used when native code is being generated to include debugging information. (ROUTINE SENSITIVE)
NULLBODY	FALSE	This option is used between the BEGIN / END of an empty module body. The body must be empty which means that statements may not occur between the BEGIN and the END. This option indicates that no code is to be generated for the empty body. (ROUTINE SENSITIVE)
ASSERTS	TRUE	This option directs the compiler to generate code for ASSERT statements. (INSENSITIVE)
Run-Time Checks		
CKINDEX	FALSE	This option is used to enable run-time checks for array indices out of bound. (INSENSITIVE)
CKPTR	FALSE	This option turns on (off) run-time checks for pointers equal to NIL. (INSENSITIVE)
CKSET	FALSE	This option is used to enable run-time checks for set element expressions out of bounds. (INSENSITIVE)
CKSUB	FALSE	This option directs the compiler to produce run-time checks for subrange assignments to assure that they are in bounds. (INSENSITIVE)

5.4 Copy Statement

A copy statement is provided so that source files can be separated into individual files. A copy statement is specified as follows:

```
?COPY file-access-name
```

where "?COPY" must begin in column one of a source line and the rest of the line after the "file-access-name" is treated as a remark. Copy files may have embedded copy statements, but the nesting is limited to 8 levels.

Use of copy files has the advantage of making editor sessions more efficient because files are smaller. One typical use of copy files is a set of commonly used declarations which can be included in separately compiled systems. Another example is a set of declarations for the Native Code RTS Library.

The example given in Section III is used in the following figures to illustrate possible uses of the ?COPY statement:

```
{$ DEBUG, MAP}
SYSTEM tutorial;

?COPY EXAMPLE.SYSDECL

?COPY EXAMPLE.PRODUCER

?COPY EXAMPLE.CONSUMER

BEGIN {#stacksize = 300; heapsize = 500}
  WITH m = message_buffer DO BEGIN    "initialize the message buffer
    m.next_in := 1;                   "index of first in-coming item
    m.next_out := 1;                  "index of first out-going item
    initsemaphore(m.exclusive_access, 1); "allow 1 access at a time
    initsemaphore(m.not_empty, 0);    "of full slots in the buffer
    initsemaphore(m.not_full, number_of_slots); "of empty slots
    END;
  START producer;
  START consumer;
END.
```

System Body

```
CONST   number_of_slots = 10;  "maximum number of slots in a buffer

TYPE    slot_index = 1..number_of_slots;
        alpha_character = 'A'..'Z';

COMMON message_buffer: RECORD                "circular message buffer
          slots: ARRAY  slot_index  OF alpha_character;
          next_in, next_out: slot_index;
          not_empty, not_full: SEMAPHORE;
          exclusive_access: SEMAPHORE;
          END;

ACCESS message_buffer;

PROCEDURE initsemaphore(var sema: SEMAPHORE; count: INTEGER);
    EXTERNAL;
PROCEDURE signal(sema: SEMAPHORE); EXTERNAL;
PROCEDURE wait(sema: SEMAPHORE); EXTERNAL;
PROCEDURE swap; EXTERNAL;
```

FIGURE 5-1. EXAMPLE.SYSDECL

```
PROGRAM producer;
VAR item: alpha_character;
    line: PACKED ARRAY  1..16  OF CHAR;
    status: INTEGER;
ACCESS message_buffer;
BEGIN   {#priority = 20; stacksize = 100}
  item := 'A';
  line := 'item produced:  ';
  WITH m = message_buffer DO
    WHILE TRUE DO BEGIN
      wait(m.not_full);                    "wait for buffer space
      wait(m.exclusive_access);            "get access to the buffer
      m.slots  m.next_in   := item;        "deposit item into buffer
      ENCODE(line, 16, status, item);
      MESSAGE(line);
      IF item = 'Z' THEN item := 'A'       "generate the next item
      ELSE item := SUCC(item);
      m.next_in := SUCC(m.next_in MOD number_of_slots);
      signal(m.exclusive_access);          "release access to buffer
      signal(m.not_empty);        "indicate presence of another item
      swap;                       "give the consumer a chance
      END;
END;
```

FIGURE 5-2. EXAMPLE.PRODUCER

```
PROGRAM consumer;
VAR item: alpha_character;
    line: PACKED ARRAY [1..16] OF CHAR;
    status: INTEGER;
ACCESS message_buffer;
BEGIN {#priority = 20; stacksize = 100}
  line := 'item consumed:  ';
  WITH m = message_buffer DO
    WHILE TRUE DO BEGIN
      wait(m.not_empty);        "wait for available item to appear
      wait(m.exclusive_access);        "wait for access to buffer
      item := m.slots m.next_out ;     "extract item from buffer
      ENCODE(line, 16, status, item);
      MESSAGE(line);
      m.next_out := SUCC(m.next_out MOD number_of_slots);
      signal(m.exclusive_access);       "release access to buffer
      signal(m.not_full);  "indicate an available space in buffer
      swap;                "give the producer a chance
      END;
END;
```

FIGURE 5-3. EXAMPLE.CONSUMER

5.5 SEPARATE COMPILATION

The Microprocessor Pascal System supports separate compilation of system segments. A segment is simply a group of modules, typically a program or process and all inner modules, which are to be compiled together. This segment may then be saved in the form of a standard 9900 object module for later use in debugging of the complete system. All separately compiled segments must be compiled with the same global declaration environment so they access the same global variables. Any modules which are referenced by a segment but are not included in the segment must be declared EXTERNAL, as are any global modules which are required only because of their declarations must have null bodys. Any module declared as having a null body in any separate compilation of system segments must have a body in another system segment. Otherwise the module having the null body is an unresolved external reference when the system is constructed (by the debugger or link editor).

The example given in Section III could be divided into segments as follows:

```
{$DEBUG}
SYSTEM tutorial;

?COPY EXAMPLE.SYSDECL

  PROGRAM producer; EXTERNAL;

  PROGRAM consumer; EXTERNAL;

BEGIN  {#stacksize = 300; heapsize = 500}
  WITH m = message_buffer DO BEGIN   "initialize the message buffer
    m.next_in := 1;                  "index of first in-coming item
    m.next_out := 1;                 "index of first out-going item
    initsemaphore(m.exclusive_access, 1); .allow 1 access at a time
    initsemaphore(m.not_empty, 0); {#of full slots in the buffer}
    initsemaphore(m.not_full, number_of_slots);{#of empty slots}
    END;
  START producer;
  START consumer;
END.
```

FIGURE 5-4. SEGMENT 1 - SYSTEM BODY

```
{$DEBUG}
SYSTEM tutorial;

?COPY EXAMPLE.SYSDECL

  PROGRAM producer;
  VAR item: alpha_character;
      line: PACKED ARRAY [1..16] OF CHAR;
      status: INTEGER;
  ACCESS message_buffer;
  BEGIN   {#priority = 20; stacksize = 100}
    item := 'A';
    line := 'item produced:  ';
    WITH m = message_buffer DO
      WHILE TRUE DO BEGIN
        wait(m.not_full);                   "wait for buffer space
        wait(m.exclusive_access);           "get access to the buffer
        m.slots  m.next_in   := item;       "deposit item into buffer
        ENCODE(line, 16, status, item);
        MESSAGE(line);
        IF item = 'Z' THEN item := 'A'      "generate the next item
        ELSE item := SUCC(item);
        m.next_in := SUCC(m.next_in MOD number_of_slots);
        signal(m.exclusive_access);          "release access to buffer
        signal(m.not_empty);     "indicate presence of another item
        swap;                    "give the consumer a chance
        END;
  END;

BEGIN  {$Nullbody}
END.
```

FIGURE 5-5. SEGMENT 2 - PRODUCER

```
{$DEBUG}
SYSTEM tutorial;

?COPY EXAMPLE.SYSDECL

  PROGRAM consumer;
  VAR item: alpha_character;
      line: PACKED ARRAY [1..16] OF CHAR;
      status: INTEGER;
  ACCESS message_buffer;
  BEGIN  {#priority = 20; stacksize = 100}
    line := 'item consumed:   ';
    WITH m = message_buffer DO
      WHILE TRUE DO BEGIN
        wait(m.not_empty);          "wait for available item to appear
        wait(m.exclusive_access);          "wait for access to buffer
        item := m.slots m.next_out ;       "extract item from buffer
        ENCODE(line, 16, status, item);
        MESSAGE(line);
        m.next_out := SUCC(m.next_out MOD number_of_slots);
        signal(m.exclusive_access);         "release access to buffer
        signal(m.not_full);  "indicate an available space in buffer
        swap;                "give the producer a chance
        END;
  END;

BEGIN  {$Nullbody}
END.
```

FIGURE 5-6. SEGMENT 3 - CONSUMER

In the example given above, each one of the segments would be compiled separately and saved. When the segments are loaded for execution, segment 1 should be loaded first.

5.6 Saving Segments

The Microprocessor Pascal System provides a utility which takes the interpretive code generated by the compiler and puts it in the form of a standard 9900 object module so that it can be saved for later use. This object module could be included in a debug session or included in an Interpretive RTS target system.

The utility to produce the segment will ask for the segment number to be assigned to this segment, and whether debug information is to be placed into the object module. The prompt file is shown on the following page with the responses shown proceeded with "-->":

```
ENTER THE SEGMENT NUMBER:
-->2
INSERT DEBUG INFO? (YES/NO):
-->YES
```

The segment number is needed for Interpretive RTS system construction, and it may be any number between 1 and 50. If an invalid segment number is specified, the following message is generated: ERROR: BAD SEGMENT NUMBER. The debug information is for debugging and must be present if this segment is to be debugged. The object modules created by the SAVE command include only the modules which are referenced. After the segment has been created, a map of the modules in the segment and those referenced by the segment is generated. The listing produced for segment 2 is shown below:

```
                    MAP FOR SEGMENT 2     LENGTH = 0098

                    ASSEMBLED WITH DEBUG INFORMATION

   0   NAME = TUTORI   KIND = EXTERNAL

   1   NAME = PRODUC   KIND = ROUTINE   DISP = 001A

   2   NAME = MESSAG   KIND = COMMON   LEN = 30

   3   NAME = WAIT     KIND = EXTERNAL

   4   NAME = SIGNAL   KIND = EXTERNAL

   5   NAME = SWAP     KIND = EXTERNAL
```

The first two lines of the listing shows the segment number, segment length, and debug information status. Then, for each module in the system, its number and name are given as well as whether it isan external module, common variable, or internal module. For common variables, the length of the common area is given in bytes. For internal modules, the hex displacement within the object module is given, followed by an indication of whether or not it is externally defined.

Saved segments are generally smaller than their unsaved counterparts. Additionally, segments saved without debug information are smaller than those saved with debug information. Once a segment of routines for a system has been throroughly tested, it may be saved without debug information to conserve space, but still may be used to test other parts of the system.

5.7 ERROR MESSAGES

This section describes the error messages generated by the compiler. The first section describes the syntax error numbers generated by the compiler when it finds errors. The next section describes all other error messages generated by the compiler.

5.7.1 Syntax Error Number Descriptions

This section describes each error number generated by the compiler by giving a brief description of the error, a more detailed description of the error, and what action to take to correct the error.

1 ERROR IN SIMPLE TYPE - This occurs when a simple type was expected but not found, or when a scalar type specification was incomplete.
Action: Make sure the simple type is specified correctly.

2 IDENTIFIER EXPECTED - This occurs when an identifier is expected but not found.
Action: Make sure the identifier is correct.

3 "SYSTEM" EXPECTED - The keyword SYSTEM was expected but not found.
Action: Make sure your system starts with SYSTEM or PROGRAM for a conventional Pascal program.

4 ")" EXPECTED - A) was expected to match a (in an expression, parameter list, record variant specification, or scalar declaration.
Action: Make sure that the parentheses are balanced.

5 ":" EXPECTED - A : was expected to follow a statement label, variable declaration, parameter list declaration, case label list, or record variant label list.
Action: Make sure statement label or declaration is specified correctly.

7 ERROR IN PARAMETER LIST - An invalid symbol was found in a parameter list or the parameter list was formed incorrectly.
Action: Correct parameter list.

8 "OF" EXPECTED - The keyword OF was expected in an array, file, or set declaration or case statement.
Action: Insert the keyword OF in the declaration.

9 "(" EXPECTED - A (was expected to begin a record variant specification.
Action: Insert the (in the specification.

10 ERROR IN TYPE - A type definition was expected but not found or incorrectly specified.
Action: Specify type correctly.

11 " " EXPECTED - A was expected in an array definition but was not found.
Action: Insert a in the array definition.

12 " " EXPECTED - A was expected in an array definition, array variable reference, or set constant but was not found.
Action: Insert the where needed.

13 "END" EXPECTED - An END was expected to terminate a record definition, case statement, compound statement, or routine body but was not found.
Action: Insert the END where needed.

14 ";" EXPECTED - A ; was expected to terminate a declaration or separate a list of statements.
Action: Insert the ; where needed.

15 INTEGER CONSTANT EXPECTED - An integer constant was expected but not found.
Action: Insert the integer constant where needed.

16 "=" EXPECTED - A = was expected in a constant declaration or type declaration but was not found.
Action: Insert the = where needed.

17 "BEGIN" EXPECTED - A BEGIN was expected to begin a module body. An error in the declaration section may cause this error.
Action: Correct the declaration section error.

18 ERROR IN DECLARATION PART - An error was found in the declaration section and recovery will begin at the next declaration.
Action: Fix declaration which had the error.

19 ERROR IN FIELD LIST - The field list does not begin with an identifier.
Action: Fix the error in the field list.

20 "," EXPECTED - A , was expected to separate a list of identifiers or labels.
Action: Use a , to separate a list of items.

22 ".." EXPECTED - A .. was expected to separate a subrange definition.
Action: Use .. to separate subrange constants.

40 ERROR IN COPY STATEMENT - This error is caused when a syntax error is found in a copy statement, when an I/O error occurs while trying to open a copy file, or when more than 8 levels of nested files are copied.
Action: Correct the copy statement.

41 STATEMENT EXPECTED - This error is caused when a statement in a list of statements does not begin with a valid token.
Action: Fix the statement.

43 FORWARD OR EXTERNAL EXPECTED - This occurs when one of the keywords, FORWARD or EXTERNAL is expected in a routine declaration, but is not found.
Action: Make sure the routine declaration is correct and that FORWARD and EXTERNAL are spelled correctly, if present.

50 ERROR IN CONSTANT - An error was found in the kind of constant or integer constant expression.
Action: Fix the constant specification.

51 ":=" EXPECTED - The assignment operator is expected in an assignment statement or for statement. This error occurs when a = is used instead of a := .
Action: Fix the assignment statement operator.

52 "THEN" EXPECTED - THEN is expected after the boolean expression in a if statement.
Action: Fix the if statement.

53 "UNTIL" EXPECTED - UNTIL is expected to terminate a repeat statement.
Action: Fix the repeat statement.

54 "DO" EXPECTED - DO is expected in a for, while, or with statement.
Action: Fix the statement.

55 "TO" OR "DOWNTO" EXPECTED - TO or DOWNTO is expected to separate the initial and final expression of the for statement.
Action: Fix the for statement.

57 "FILE" EXPECTED - FILE is expected after the keyword RANDOM.
Action: Fix the file definition.

58 ERROR IN FACTOR - An error was found while processing the operand of an expression. The operand was expected but was not found.
Action: Fix the expression.

59 ERROR IN VARIABLE - A variable was expected but an invalid variable identifier was found.
Action: Fix the variable.

60 "HEX" EXPECTED - HEX was expected to follow a write statement parameter but was not found. This may be caused by a missing comma in the write statement.
Action: Fix the write statement.

80 OPTION IDENTIFIER EXPECTED - An identifier was expected in an option comment but was not found.
Action: Fix the option comment.

81 UNKNOWN OPTION IDENTIFIER - The option name is unknown to the option processor. This may be caused by an unsupported option.
Action: Fix the option comment.

82 SYSTEM SENSITIVE OPTION NOT ALLOWED HERE - A system sensitive option may only be specified before the first symbol of a system.
Action: Place the option comment at the beginning.

83 MODULE SENSITIVE OPTION NOT ALLOWED HERE - A module sensitive option may only be specified between the module header and the first declaration, or after the begin statement of the body and the first statement.
Action: Place the option comment at the correct place.

84 NULL BODY EXPECTED - The null body option was specified but an empty body was not found. Null body may only be used within a empty begin / end body.
Action: Fix the null body specification.

85 ERROR IN CONCURRENT CHARACTERISTIC SPECIFICATION - The concurrent characteristic identifier is not PRIORITY, HEAPSIZE, or STACKSIZE.
Action: Fix the concurrent characteristic specification.

101 IDENTIFIER DECLARED TWICE - The identifier has already been declared at this level and cannot be redeclared.
Action: Use another identifier.

102 LOWER BOUND EXCEEDS UPPER BOUND - The lower bound of a subrange specification exceeds the upper bound.
Action: Fix the subrange definition.

103 WRONG KIND OF IDENTIFIER - The identifier found is not the correct kind. A procedure identifier within an expression is an example of this type of error.
Action: Use the identifier correctly.

104 IDENTIFIER NOT DECLARED - All identifiers must be declared before they are referenced. This is most often caused by a misspelled identifier.
Action: Declare the identifier.

105 SIGN NOT ALLOWED - The constant operand was not a binary constant so a sign is not allowed.
Action: Correct the constant.

106 NUMBER EXPECTED - A binary constant was expected but was not found.
Action: Correct the constant.

107 INCOMPATIBLE SUBRANGE TYPES - The type of the lower bound and upper bound do not agree.
Action: Correct the subrange specification.

108 FILE NOT ALLOWED HERE - A pointer may not point to a file variable, and a file may not be the component type of an array or be a field type within a record.
Action: Fix the pointer specification.

110 TAGFIELD TYPE MUST BE SCALAR OR SUBRANGE - The record variant selector type must be scalar or a subrange.
Action: Correct the tagfield type specification.

111 INCOMPATIABLE VARIANT LABEL - Record variant label is incompatible with the type of the record variant selector type.
Action: Correct the label specification.

113 INDEX TYPE MUST BE SCALAR OR SUBRANGE - The array index type must be a scalar or subrange type. This also applies to the array variable index expression.
Action: Fix the array specification.

115 SET BASE TYPE MUST BE SCALAR OR SUBRANGE - The set base type must be a scalar or subrange type.
Action: Fix the set specification.

116 ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER - The type of the parameter is not the type which was expected for the particular standard procedure parameter.
Action: Correct the standard procedure call.

119 REPETITION OF PARAMETER LIST NOT ALLOWED - When the full declaration of a forward routine is given, the parameter list must not be repeated.
Action: Fix the routine header.

120 FUNCTION RESULT TYPE MUST BE SCALAR, SUBRANGE, OR POINTER - The type of the result returned by a function must be scalar, subrange, or a pointer.
Action: Fix the function specification.

121 FILE VALUE PARAMETER NOT ALLOWED - A file must be passed by reference to a procedure or function.
Action: Fix the file parameter specification.

122 REPETITION OF THE RESULT TYPE NOT ALLOWED - When the actual declaration of a function declared forward is given, the result type must not be repeated.
Action: Fix the function specification.

123 MISSING RESULT TYPE IN FUNCTION DECLARATION - The type of the function was expected but was not found.
Action: Fix the function specification.

125 ERROR IN TYPE OF STANDARD FUNCTION PARAMETER - The type of a standard function parameter is incompatible with what was expected.
Action: Fix the parameter of the standard function call.

126 NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION - The number of parameter in the call does not agree with the declaration of the routine.
Action: Fix the call parameters.

127 ACTUAL PARAMETER MUST NOT BE PACKED - The actual reference parameter must not be packed.
Action: Pass an unpacked variable by reference and assign its returned value to the packed variable.

129 TYPE CONFLICT IN ASSIGNMENT - The type of the expression is not compatible with the variable on the left hand side of the assignment.
Action: Fix the assignment statement.

130 EXPRESSION IS NOT A SET - The second operand of an IN operator must be a set but it is not.
Action: Fix the expression.

131 TESTS FOR POINTER EQUALITY ONLY - The only operators valid on pointer types are equal to and not equal to.
Action: Fix the expression.

132 ILLEGAL OPERATOR - The operator is not valid given the types of the operands or the expression is misformed.
Action: Fix the expression.

134 ILLEGAL TYPE OF OPERAND(S) - The type of the operands are incompatible.
Action: Fix the expression operands.

135 TYPE OF EXPRESSION MUST BE BOOLEAN - The type of the expression was expected to be boolean but it was not.
Action: Supply a boolean expression.

136 SET ELEMENT TYPE MUST BE SCALAR OR SUBRANGE - The type of a set constant element must be scalar or subrange but it was not.
Action: Fix the set constant.

137 SET ELEMENT TYPES NOT COMPATIBLE - The type of the set constant element is not compatible with previous set elements.
Action: Fix the set constant.

138 TYPE OF VARIABLE IS NOT ARRAY - Array subscripts are allowed only on array variables.
Action: Fix the variable specification.

139 INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION - The type of the array subscript expression is not compatible with the declaration of the array.
Action: Fix the variable specification.

140 TYPE OF VARIABLE IS NOT RECORD - A field designator is valid only after a variable of type record.
Action: Fix the variable specification.

141 TYPE OF VARIABLE MUST BE POINTER - A pointer reference is only valid on a variable of type pointer.
Action: Fix the variable specification.

142 ILLEGAL PARAMETER SUBSTITUTION - The type of the actual parameter is not compatible with the declaration of the parameter.
Action: Fix the call parameter.

143 ILLEGAL TYPE OF FOR EXPRESSION - The type of the initial and final for expressions are not compatible or they are not scalar types.
Action: Fix the for statement.

144 ILLEGAL TYPE OF CASE SELECTOR - The type of the case selector must be scalar or subrange.
Action: Fix the case selector expression.

146 ASSIGNMENT OF FILES OR SEMAPHORES NOT ALLOWED - Files and semaphores may not be assigned to other variables.
Action: Delete file assignment.

147 INCOMPATIBLE CASE LABEL - The type of the case label is not compatible with the case selector expression.
Action: Fix the case label.

148 SUBRANGE BOUNDS MUST BE SCALAR - The type of the subrange constants must be scalar.
Action: Fix the subrange constants.

149 INDEX TYPE MUST NOT BE "INTEGER" - The index type must be bounded, and INTEGER does not have fixed lower or upper bounds.
Action: Change array specification.

152 NO SUCH FIELD IN THIS RECORD - The identifier specified was not declared to be a field of the record variable. The identifier may be misspelled.
Action: Fix the variable specification.

154 ACTUAL PARAMETER MUST BE A VARIABLE - Only variables may be passed by reference to routines and they may not be components of packed structures.
Action: Pass an unpacked variable instead of an expression.

156 MULTIDEFINED CASE LABEL - The case label was already defined in another alternative. This may be caused by overlapping subranges.
Action: Fix the case label specification.

157 CASE LABEL RANGE TOO LARGE - The total range of all labels in a case statement must be no larger than 256.
Action: Use a different method for ranges greater than 256.

158 MISSING CORRESPONDING VARIANT DECLARATION - The record specified in NEW or SIZE was not declared to have variants.
Action: Fix the NEW or SIZE constant parameter.

160 PREVIOUS DECLARATION WAS NOT FORWARD - A module which was previously declared is being redeclared at the same level.
Action: Correct the routine specification.

161 MODULE DECLARED FORWARD AGAIN - Two forward declarations for a module are not allowed.
Action: Correct the module specification.

162 PARAMETER MUST BE A CONSTANT - A constant parameter is expected for NEW or SIZE but was not found.
Action: Correct the parameter specification.

163 MISSING VARIANT IN DECLARATION - The constant value specified in a NEW or SIZE was not found in the record variant list.
Action: Fix the constant specification.

165 MULTIDEFINED LABEL - A statement label must appear only once within a module.
Action: Fix the statement label specification.

166 MULTIDECLARED LABEL - A statement label must appear only once in the label declaration list.
Action: Fix the statement label declaration.

167 UNDECLARED LABEL - A statement label must be declared in the label declaration section of the module where it is defined and referenced.
Action: Declare the statement label.

177 ASSIGNMENT TO NON-LOCAL FUNCTION NOT ALLOWED - A value may only be assigned to the local function identifier.
Action: Fix the assignment statement.

178 MULTIDEFINED RECORD VARIANT LABEL - The record variant label was defined in another record variant list. This may be caused by overlapping subranges.
Action: Fix the record variant label specification.

179 ILLEGAL ESCAPE - An escape statement may not reference another routine at the same lexical level as the current routine.
Action: Fix the escape statement.

180 UNACCESSED COMMON VARIABLE - The common variable referenced was not in the access list of the current routine.
Action: Declare access to the common variable.

181 ASSIGNMENT TO "FOR" VARIABLE IS NOT ALLOWED - The for variable may not be modified within the for statement.
Action: Delete the assignment statement.

182 ACTUAL REFERENCE PARAMETER MUST NOT BE A FOR VARIABLE - A for variable may not be passed by referenced to a routine.
Action: Fix the call statement.

183 ILLEGAL TYPE TRANSFER - A type transfer was applied to a packed element which was larger than the original element.
Action: Fix the type transfer specification.

184 TYPE OF COMMON MUST NOT BE A FILE - A common variable may not be a file.
Action: Make the common a global variable.

185 FILE ELEMENT TYPE MUST NOT BE FILE OR POINTER - A file of files or file of pointers is not allowed.
Action: Fix the file specification.

186 SET BOUND OUT OF RANGE - The lower bound of a set must not be less than 0 and the upper bound of a set must not be greater than 1023.
Action: Fix the set specification.

188 DIVISION BY ZERO - Division by zero is not allowed in an integer constant expression.
Action: Fix the integer constant expression.

189 STATEMENT MUST BE A STRUCTURED STATEMENT - Escape labels are allowed only on structured statements.
Action: Fix escape label specification.

190 STATEMENT LABEL IN FOR OR WITH STATEMENT NOT ALLOWED - This is a warning message which indicates that a goto statement could possibly jump into a for or with statement with undefined results.
Action: Check for jumps into the for or with statement.

191 VARIABLE DECLARATIONS NOT ALLOWED AT SYSTEM LEVEL - Global variables may not be declared at the system level.
Action: Make system variables commons.

192 INVALID NESTING OF SYSTEM, PROGRAM, OR PROCESS - Programs may only be declared within a system, and processes may only be declared within programs or other processes. Systems may not be declared within anything.
Action: Fix declaration.

193 REFERENCE PARAMETERS NOT ALLOWED FOR PROGRAM OR PROCESS - Only value parameters are allowed for programs or processes.
Action: Change parameters to value parameters.

194 POINTER PARAMETERS NOT ALLOWED FOR PROGRAM - Pointers may not be passed as parameters to programs because heaps are local to programs.
Action: Fix parameter specification.

195 "INPUT" AND "OUTPUT" MUST BE DECLARED "TEXT" - When specifying INPUT or OUTPUT as parameters to programs or processes, they must be declared to be text.
Action: Declare files to be text.

196 "INPUT" OR "OUTPUT" NOT DECLARED - Implicit use of INPUT or OUTPUT was encountered in a standard I/O routine without a declaration.
Action: Declare the needed text file, INPUT or OUTPUT; or remove the reference to the I/O routine.

201 FRACTION EXPECTED - The fractional portion of a real number was expected but was not found.
Action: Specify real constant correctly.

202 STRING CONSTANT MUST NOT EXCEED SOURCE LINE - A string constant must not extend across a source line boundary. This error may be caused by an unclosed string constant.
Action: Correct string constant.

203 INTEGER CONSTANT EXCEEDS RANGE - The integer constant can not be represented as a long integer.
Action: Correct integer constant.

206 EXPONENT EXPECTED - A real constant was followed by an E but no exponent was found.
Action: Correct real constant.

207 HEX DIGIT EXPECTED - A character other than a hex digit was found. Only digits 0 through 9 and letters A through F are hex digits.
Action: Correct hex constant.

208 ILLEGAL LONG INTEGER CONSTANT - A real constant was suffixed with the letter L which indicates a long integer constant.
Action: Correct real constant.

209 NESTED COMMENTS - This is a warning message that indicates that a comment was found within another comment. This may indicate a previously unclosed comment.
Action: Check for unclosed comments.

251 TOO MANY NESTED MODULES - Modules may only be nested to a level of 10 or less.
Action: Correct routine nesting.

252 TOO MANY MODULES DECLARED - Only 256 modules may be declared in one compilation.
Action: Your system is too large to be compiled. A possible action is to split the system into separate segments and compile the segments separately.

255 TOO MANY ERRORS IN THIS SOURCE LINE - If more than 9 errors are found on any one line, this error message is generated.
Action: Fix all errors on line.

258 TOO MANY IDENTIFIERS DECLARED IN LIST - Only 8 identifiers may be declared in one identifier list.
Action: Break declaration up into multiple lists.

304 SET ELEMENT OUT OF RANGE - A set constant element is less than 0 or greater than 1023.
Action: Correct set constant.

399 INTERNAL COMPILER ERROR - An inconsistency was found in the compiler which may be caused by previous errors.
Action: Fix all errors and try again.

5.7.2 Other Compiler Error Messages

The following error messages are generated by the compiler in message form rather than error number.

**** UNRESOLVED TYPE - name -- This message is generated when the type referenced by a pointer is not declared. Since this is the only time forward type references are allowed, a error message cannot be generated until the end of the declaration section. The type identifier name is given.
Action: Define the type identifier.

**** LABEL UNDEFINED - number -- This message is generated at the end of the body when a label is declared and referenced but not defined. Each label declared and referenced must precede one and only one statement within the body where it is declared.
Action: Define the statement label.

**** END OF SYSTEM EXPECTED **** -- This message is generated at the end of a compilation when the end of the system or program is expected but more source is found. This may be caused by mismatched begin/end pairs or some other mismatched statement. Only one system or program may be compiled at one time.
Action: Correct syntax errors.

5.7.3 Native Code Generator Error Messages

This section will describe the error messages generated by the Native Code Generator when it is producing 9900 machine code. The Native Code Generator is not supported in the current release, but will be added in a later release.

SECTION VI

HOST DEBUGGER GUIDE

6.1 HOST DEBUGGER OVERVIEW

A flaw in software is commonly called a bug. The act of removing bugs from software is called debugging. The Source Editor helps the user to construct a syntactically correct program, whereas the Microprocessor Pascal System language and its compiler help the user to discover and to correct semantic errors. Because the language itself prohibits the use of semantically inconsistent operations on data, the user is freed from many traditional debugging chores. However, an interactive debugging tool such as the Host Debugger can be very useful for observing the behavior of Microprocessor Pascal Systems as they evolve; i.e. the debugger can be a useful design tool.

It must be recognized that the design, implementation, and testing of large, complex systems is a difficult task. The modern approach to this problem is to break up large pieces of software into small, independent units. The smaller modules can be examined one at a time to ensure that they perform the desired function; intuitively, this task is more intellectually manageable. If the interfaces between the modules are well-defined and the modules work correctly by themselves, it is reasonable to assume that the entire system will perform correctly when the modules are combined. The Host Debugger can be used to ensure that modules perform correctly by themselves; it is also useful for monitoring and possibly altering the interfaces between modules and concurrent processes.

The debugger's user interface is designed to be as helpful and friendly as possible. Whenever The debugger expects a response from the user, a prompting message is displayed, which usually consists of the characters "<>". The HELP command can be used at any time to determine the correct syntax for a command.

A complete history of a debugging session can be obtained on a hard copy device if desired. User input commands, debugger responses, trace information, and all status displays are sent to a log file, stored on disc at some user determined pathname. It is sometimes helpful to be able to track the steps which led to a certain state, so that state can be recreated. HELP displays are not echoed on the log file. Note also that the log file does not contain any user input, output, or messages.

The following table is a summary of the debugger command names. A detailed description of each command is given in section 6.3.

Command Name	Meaning
Getting Started/Finished	
GO	Resume execution
QUIT	Quit debugging session
HELP	Help command
DEBUG	Debug process
LOAD	Load saved segment
SE	Show unresolved Externals
COPY	Copy commands from file
Status Displays	
DP	Display Process
DAP	Display All Processes
Breakpoints/Single Step	
AB	Assign Breakpoint
DB	Delete Breakpoint
DAB	Delete All Breakpoints
LB	List Breakpoints
SS	Single-Step execution mode
Showing/Modifying Data	
SF	Show Frame
SH	Show Heap
SC	Show Common
SI	Show Indirect
SM	Show Memory
MF	Modify Frame
MH	Modify Heap
MC	Modify Common
MI	Modify Indirect
MM	Modify Memory
Tracing Execution	
TP	Trace Process scheduling
TR	Trace Routine entry/exit
TS	Trace Statement flow
TOFF	Trace echo OFF
TON	Trace echo ON
Monitor Process Scheduling	
SDP	Select Default Process
ABP	Assign Breakpoint to Process
DBP	Delete Breakpoint from Process
HP	Hold Process
RP	Release Process

Interprocess File Simulation	
CIF	Connect Input File
COF	Connect Output File
Interrupt Simulation	
SIMI	SIMulate Interrupt
Selection of CRU Mode	
CRU	select CRU mode

6.2 DEBUGGING EXAMPLES

A system to be monitored using the Host Debugger must be compiled with the DEBUG option set. This is done by inserting a .$DEBUG option comment into the source code before it is compiled -- see Section 5.3 of this manual for the available compiler options. The compiler listing contains useful information for debugging as follows: (1) in the body section of each module (between the BEGIN-END pair), statement numbers are listed in the left margin, (2) in the declaration section of a module, stack frame displacements for variables are listed. Consider the following example compiler listing:

```
   DX Microprocessor Pascal Compiler     1.0   06/11/79 09:32:44

     0   {$ DEBUG, MAP}
     0 PROGRAM example;
     0 VAR
     0     n: INTEGER;
     2     m: INTEGER;
     0 FUNCTION factorial (i: INTEGER): INTEGER;
     1 BEGIN
     1   IF i <= 1 THEN
     2     factorial := 1
     3   ELSE
     3     factorial := i * factorial (i - 1)
     4 END;
     1 BEGIN  {# stacksize = 200}
     1   n := 5;
     2   m := factorial (n);
     3 END.

  PROGRAM EXAMPLE ;
      STACK SIZE = 0004

            VARIABLE  DISP      TYPE      SIZE
            N         0000      INTEGER   2
            M         0002      INTEGER   2

  FUNCTION FACTORIA ( I         :INTEGER):INTEGER;
      STACK SIZE = 0004

            VARIABLE  DISP      TYPE      SIZE
            I         0000      INTEGER   2
```

In the program "example", the variable "n" is at displacement 0 in the stack frame for "example"; the variable "m" is at displacement 2. The value parameter "i" in the function "factorial" is stored at displacement 0 in its stack frame. The function result is stored at displacement 2 in the stack frame for "factorial". The statement numbers are listed in the body section.

The following "walk-through" is an example of an interactive debugger session. All user input commands are preceeded by the characters <>. Output messages from the debugger are shown following many of the commands, although some commands do not evoke a response from the debugger. Explanatory comments about the example walk-through is given between the comment symbols { and } .

```
                        {start of debugging session}

 HOST DEBUGGER       1.0      03/27/79   09:07:51

     Enter system heap size in (K)bytes: 5
     Do you wish to debug the most recently compiled system?
     Please answer YES or NO: YES

<> DEBUG(example)   {breakpoint when "example" process is created}

<> GO
     run-time support now initialized

<> GO

     *** Process Created *** EXAMPL(1)

<> AB(factorial, 1) {assign breakpoint to statement 1 of factorial}

<> AB(example, 1)   {assign breakpoint to statement 1 of example}

<> LB

     Breakpoints for Process EXAMPL(1)
       EXAMPL          1
       FACTOR          1

<> DB(example, 1)   {delete breakpoint from statement 1 of example}

<> LB

     Breakpoints for Process EXAMPL(1)
       FACTOR          1

<> AB(example, 3)

<> AB(example, 2)

<> AB(example, 1)

<> LB

     Breakpoints for Process EXAMPL(1)
       EXAMPL          1
       EXAMPL          2
```

```
          EXAMPL          3
          FACTOR          1

 <> GO

        *** Breakpoint ***       EXAMPL(1).EXAMPL    Statement 1

 <>DP

 Static/Dynamic Calling Order for Process EXAMPL(1)

        Stack Size (bytes) = 464
        Stack Used (bytes) Maximum = 4  Current = 4

        Call Order           Name                Statement
             1             EXAMPL                    1

<> SF(example)

        stack frame for EXAMPL(1).EXAMPL
  CD9A (0000) 0000 0000                          (....              )

<> SF(factorial)   {note, factorial has not been called}

        stack frame not found

. <> GO

        *** Breakpoint ***       EXAMPL(1).EXAMPL    Statement 2

<> DP

 Static/Dynamic Calling Order for Process EXAMPL(1)

        Stack Size (bytes) = 464
        Stack Used (bytes) Maximum = 4  Current = 4

        Call Order           Name                Statement
             1             EXAMPL                    2

<>  SF                 { note the value of "n" is 5}

        stack frame for EXAMPL(1).EXAMPL
  CD9A (0000) 0005 0000                          (....              )

<> GO
        *** Breakpoint ***       EXAMPL(1).FACTOR    Statement 1

 <> DP

 Static/Dynamic Calling Order for Process EXAMPL(1)

        Stack Size (bytes) = 464
        Stack Used (bytes) Maximum = 36  Current = 36
```

```
     Call Order           Name                 Statement
          1             EXAMPL                     2
          2               FACTOR                   1

< > SF

     stack frame for EXAMPL(1).FACTOR
 CB42 (0000) 0005 0000                         (....         )

< >GO

     *** Breakpoint ***       EXAMPL(1).FACTOR    Statement 1

<> DP             {note that factorial is called recursively}

Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (bytes) = 464
     Stack Used (bytes) Maximum = 70  Current = 70

     Call Order           Name                 Statement
          1             EXAMPL                     2
          2               FACTOR                   3
          3               FACTOR                   1

<>SF

     stack frame for EXAMPL(1).FACTOR
 CB64 (0000) 0004 0001                         (....         )

<>GO

     *** Breakpoint ***       EXAMPL(1).FACTOR    Statement 1

< > DP            {note, factorial again called recursively}

Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (bytes) = 464
     Stack Used (bytes) Maximum = 104  Current = 104

     Call Order           Name                 Statement
          1             EXAMPL                     2
          2               FACTOR                   3
          3               FACTOR                   3
          4               FACTOR                   1

<> SF

     stack frame for EXAMPL(1).FACTOR
 CB86 (0000) 0003 0001                         (....         )

<>GO
```

```
     *** Breakpoint ***       EXAMPL(1).FACTOR    Statement 1

<> DP                 {and again, factorial is called}

Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (bytes) = 464
     Stack Used (bytes) Maximum = 138  Current = 138

     Call Order           Name                  Statement
          1             EXAMPL                      2
          2               FACTOR                    3
          3               FACTOR                    3
          4               FACTOR                    3
          5               FACTOR                    1

<> SF

     stack frame for EXAMPL(1).FACTOR
 CBA8 (0000) 0002 0001                           (....            )

<>GO

     *** Breakpoint ***       EXAMPL(1).FACTOR    Statement 1

<>DP

Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (bytes) = 464
     Stack Used (bytes) Maximum = 172  Current = 172

     Call Order           Name                  Statement
          1             EXAMPL                      2
          2               FACTOR                    3
          3               FACTOR                    3
          4               FACTOR                    3
          5               FACTOR                    3
          6               FACTOR                    1

<> SF                 {note, the value of the parameter is 1}

     stack frame for EXAMPL(1).FACTOR
 CBCA (0000) 0001 0001                           (....            )

<> GO

     *** Breakpoint ***       EXAMPL(1).EXAMPL    Statement 3
```

```
<> DP

Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (bytes) = 464
     Stack Used (bytes) Maximum = 172  Current = 4

     Call Order           Name                     Statement
          1             EXAMPL                         3

<> SF                {note, the value of "m" is 78 (hex) = 5 factorial}

     stack frame for EXAMPL(1).EXAMPL
 CD9A (0000) 0005 0078                                (...x            )

<> GO

     *** Process Terminated *** EXAMPL(1)
         Stack Used (bytes) = 172

<> DAP

Status Summary of All Existing Processes

                         Site of                             Enabled   Stmt
     Process Name       Execution       Status          Pri Traces    Bkpts

   0  IDLE$P            IDLE$P    0     Active          32767          no
   1   EXAMPL                          >Terminated                     yes

<> QUIT

Execution Terminated
Memory Used (bytes)  Maximum = 3064  Current =  1266
```

6.3 DEBUGGER COMMANDS

A Host Debugger command is similar in appearance to a Microprocessor Pascal System or AMPL procedure call. AMPL stands for "A Microprocessor Prototyping Lab", which is a useful tool for debugging at the the target machine level. A debugger command name is followed by a (possibly empty) list of parameters enclosed in parentheses. Parameters are separated by commas, as in a normal Microprocessor Pascal System procedure call. Only one debugger command can appear on a single input line. Continuation of a command across an input line boundary is not allowed.

The debugger commands are described using a BNF-like notation. Command names are written using upper-case characters, although the debugger treats every lower-case character as if it were its upper-case equivalent (the command "SF" is the same as "sf"). Parameters are written using lower-case characters. The brackets and are used to indicate that the enclosed symbol is optional.

Examples:

```
        SF (  routine ,  displacement ,  length  )
```

This is the syntax for the SF (Show Frame) command. All three parameters as described in a later section, are optional.

```
        SM ( address,  length  )
```

This is the syntax for the SM (Show Memory) command. The first parameter must be present, whereas the second parameter is optional.

6.3.1 Kinds of Parameters

There are four basic types of parameters recognized by the debugger. These are:

(1) integer constant - An integer constant parameter can be either in hex or decimal format. By default, all numerals (strings of digits) are interpreted in decimal. Any numeral beginning with the character > or #, however, is interpreted in hex. For example, the number twenty-two may be represented by 22, >16, or #16. Hex integer values may contain from 1 to 4 hex digits.

(2) name - A name parameter has the same syntax as a Microprocessor Pascal identifier. The debugger maintains routine names and common names which must be unique within the first six characters. A name parameter may be longer than six characters, but only the first six characters are significant.

(3) string - A string parameter is a character string enclosed in double quotes (a double quote is represented by two double quotes inside a string).

(4) qualified routine - A qualified routine parameter consists of an integer or name, followed by a period, followed by an integer or name.

6.3.2 Process and Routine Parameters

To define the debugger commands, a consistent terminology is adopted. A process is considered to be an abstraction which may be represented at the source code level by a SYSTEM, PROGRAM, or PROCESS. The term process is then used to refer to a specific entity which owns a set of resources and performs some succession of computations. A routine is considered to be any executable sequence of source statements which is delineated by a BEGIN-END pair. The term routine can then be used to refer to the body of a SYSTEM, PROGRAM, PROCESS, PROCEDURE, or FUNCTION.

For example, one could refer to the process CARDREADER which is represented at the source code level as a PROCESS. CARDREADER can also be referred to as a routine when talking about the individual statements which comprise its body.

Keeping these terms in mind, the following command descriptions make extensive use of two special kinds of parameters: process parameters and routine parameters.

A process parameter specifies a process, either by name or by a unique positive integer which is assigned by the debugger. A process name, when given as a parameter, always refers to the most recently created process of that name. To refer to an older instance of a process, the process number must be given. Process numbers are displayed in the far left column of the DAP (Display All Processes) display.

A routine parameter must not only specify a routine, but also the process which caused it to be invoked. A routine parameter can be written in one of six different ways as follows:

Forms for Routine Parameters	Examples
i. process-name.routine-name	EXAMPLE.DUMMY
ii. process-name.routine-number	EXAMPLE.3
iii. process-number.routine-name	4.DUMMY
iv. process-number.routine-number	4.3
v. routine name	DUMMY
vi. routine number	3

Each of the example routine parameters given in the table refer to the same routine. It is assumed that EXAMPLE is process number 4 and has been selected as the default process. EXAMPLE contains a procedure DUMMY which is number 3 in the dynamic calling sequence. If a routine name or number is simply given, as in form (v) and (vi), the default process is implicitly specified. Therefore, for "DUMMY" to be equivalent to "EXAMPLE.DUMMY", EXAMPLE must be the default process. A routine name, when given as a parameter, always refers to the most recent activation of the routine. To refer to an older activation of a routine, its dynamic calling number must be given (see the DP command).

In the first four forms presented above, the process is specified explicitly either by name or by number, according to the rules for process parameters, stated previously.

The syntax of a "routine" parameter is then:

```
process . routine
```

where the process parameter (along with the period seperator) is optional.

6.3.3 Optional Parameters

If a parameter is optional, it can simply be omitted, forcing the default value to be assumed. However, since parameters are positional, extra commas are sometimes required if parameters are omitted. For example, the length parameter in the show frame command (SF) can be given without specifying the displacement, as in "SF(TEST,,12)". This command says to show 12 bytes of the stack frame for the routine TEST, starting at the default displacement (zero). Note that the consecutive commas are required in this case.

If a command has no parameters or all of its parameters are optional, the command name may be given by itself or the command name may be given followed by a set of (matching) parentheses. For example, the command "SF" is equivalent to "SF ( )". Also, extra commas for optional parameters at the end of a command need not appear. For example, the command "SF(MYROUTINE,,)" is equivalent to "SF(MYROUTINE)".

6.3.4 Getting Started/Finished

When a debugging session is started, the debugger asks the question: "Do you wish to debug the most recently compiled system?". If the response is YES, the object code for the most recently compiled system is loaded. An error occurs if no object code is found, i.e. there was nothing previously compiled. If the response is NO, the debugger assumes that the object code to be debugged is to be loaded explicitly by the user via LOAD commands. Therefore, it is not necessary to recompile a system each time it is to be debugged, provided the object is saved.

For relatively large and complex systems it is advantageous to divide the system into segments which can be separately compiled and saved (see Section 5.5). A considerable amount of time can be saved if a change to a large system only involves editing and compiling a relatively small portion of it. The Host Debugger supports this mode of development by allowing code segments to be loaded explicitly by the user. The debugger automatically "links" together references from one code segment to another. The diagram below illustrates the alternative debugging strategies:

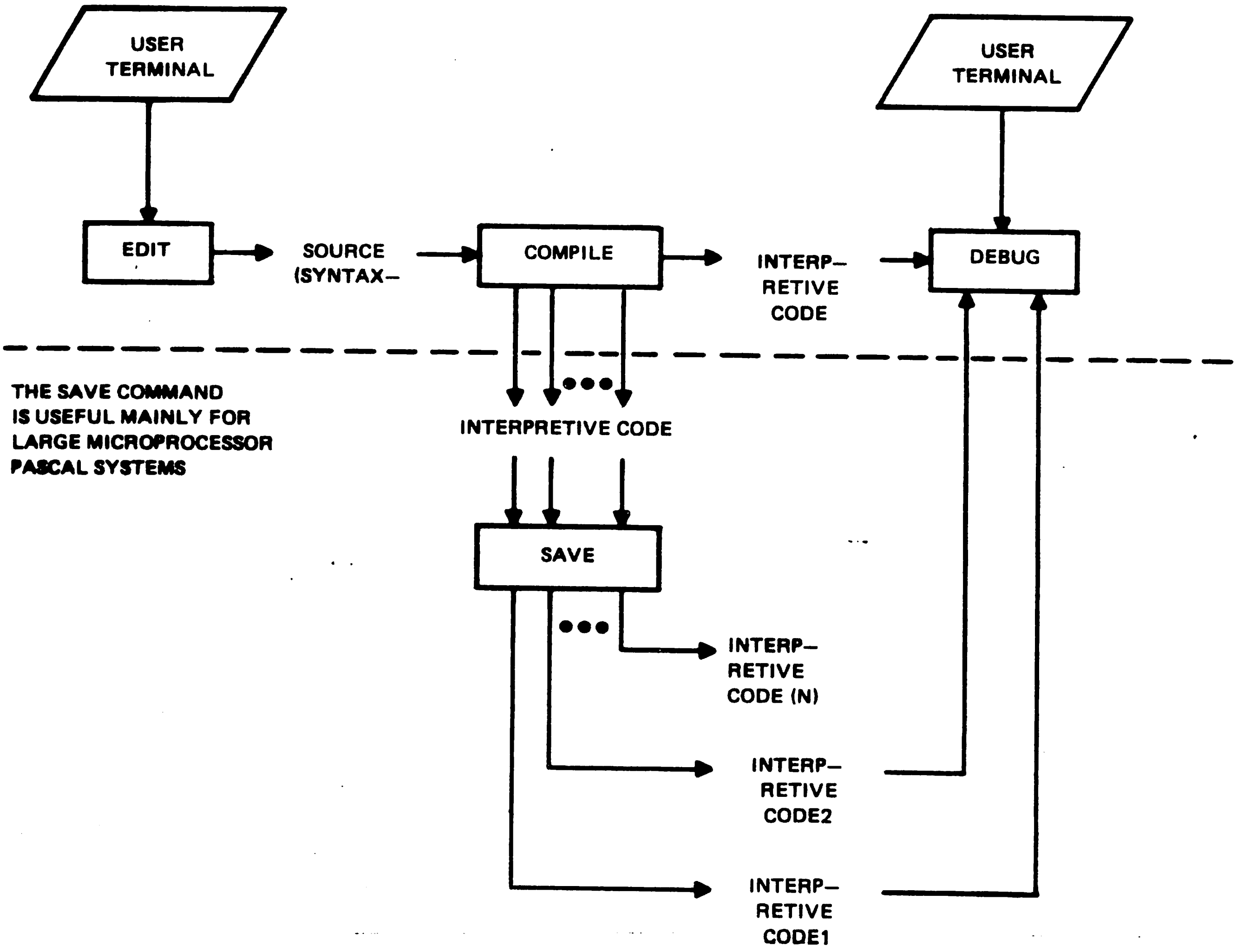


FIGURE 6-1. DEBUGGING STRATEGIES

The square boxes in the diagram represent Microprocessor Pascal Development System commands. The tilted boxes represent the interactive user terminal. A software development session usually begins with the user creating or modifying source code using the syntax-checking Source Editor. The created source code is then sent to the compiler and transformed into interpretive code. The interpretive code can either be debugged immediately or can be transformed by the SAVE command into a form suitable for later use by the debugger. In the diagram, the boxes above the dotted line illustrate a single system from the edit phase all the way through the debug phase. The SAVE command box illustrates that interpretive code can be collected into a library of interpretive code segments which are suitable for loading into the debugger.

When the Host Debugger is started, the user is informed if unresolved external references are detected in the interpretive code. Unresolved references must be resolved by explicit user commands to the debugger (via the LOAD command). Unresolved references are caused by undefined FORWARD or EXTERNAL routines or undefined NULLBODY routines. In the diagram, two interpretive code segments (previously compiled and saved) were loaded into the debugger using commands of the form LOAD("Interpretive Code1") and LOAD("Interpretive Code2"). As mentioned previously, the interpretive code file need not exist; a previously compiled and saved system may be loaded explicitly.

The debuggger prompts the user for commands with the characters "<>". Immediately after the debugger is started, a limited set of commands are valid which enable the debugging session to be set up properly. For example, commands may be given to load interpretive code, to display a list of unresolved references, to debug certain processes, and to execute a file of previously built commands (such as a sequence of LOAD or DEBUG commands). The initially valid commands include: GO, QUIT, HELP, DEBUG, LOAD, SE, and COPY.

When the initial GO command has been given, after a reasonable amount of time, a message of the following form should appear:

```
run-time support now initialized
```

At this point before user processes are created, file connections should be made using the CIF and COF commands. A subsequent GO command causes execution to begin.

6.3.4.1 GO Command. This commandd is used to resume execution of the user's system after it has become suspended for some reason, e.g. encountering a breakpoint. It is also used to start execution of the user's system when the debugger is initially invoked. Entering a blank command line is equivalent to entering a GO command.

The Executive RTS creates a process called the idle process with the least possible urgency, priority 32767. The idle process is always ready and is the last member of the scheduling queue. It's name as can be seen in the DAP command is "IDLE$P". If this process ever becomes active, the following message is displayed by the debugger:

```
idle instruction
```

Execution of the idle process places the processor in an idle state (executes the IDLE instruction) in which it remains until an interrupt occurs.

6.3.4.2 QUIT Command. This command is used to terminate the current debugging session.

6.3.4.3 HELP Command. This command is used to display information about the available debugger commands and their parameters. The syntax of this command is:

```
HELP ( command name )
```

The optional parameter is the name of a debugger command. If a command name is given, detailed information for the specific command is displayed; otherwise, a summary of all commands is displayed. The HELP command on a TX host develepment system does not allow a parameter. The only HELP available for a TX system is a summary of all commands.

6.3.4.4 Debug Process - DEBUG Command. This command is used to select a specified process for debugging. When a process is selected for debugging, a breakpoint occurs every time a process of the given name is created. The syntax of this command is:

```
DEBUG ( process name, flag )
```

The process name parameter must be the name of a process (a process number is not allowed). This enables the command to be specified before any process of the given name has been created. The flag parameter is one of the Boolean values TRUE or FALSE (T and F are also accepted as abbreviations). The value TRUE selects the process for debugging; the value FALSE indicates the process is not selected for debugging. If the flag parameter is not given, the default value of TRUE is assumed.

6.3.4.5 LOAD Command. This command is used to load a previously compiled and saved code segment. The syntax of this command is:

```
LOAD ( "pathname" )
```

The single pathname parameter is a string enclosed in double quotes. The pathname is the file name for the file on which the code segment was saved using the SAVE command of the Microprocessor Pascal Development System. This command along with the SAVE capability provides a convenient form of separate compilation for large and small systems. It also enables the user to replace standard EXTERNAL run-time support routines with specially constructed versions.

If the debugger detects that the module being loaded was not saved with debug information, the following warning is issued.

```
warning: module not saved with DEBUG information
```

If the system to be debugged is not one most recently compiled, at least one LOAD command must be given to load a previously compiled system. Note: the first module which is loaded must contain the body of the system to be debugged or the first module must contain a dummy system body (nullbody) with the same name as the system to be debugged.

6.3.4.6 SE Command. This command is used to show the names of any unresolved external routines. The syntax of this command is:

```
SE
```

The list of unresolved externals (if any) contains the names of routines which were declared as EXTERNAL but have not yet been defined. This command is used primarily in conjunction with the LOAD command. Note that the same name may occur multiple times in the list. There is one entry in the list corresponding to a single reference to the given external routine.

6.3.4.7 COPY Commmand. This command is used to execute a series of commands from an external file (disc file, card deck, cassette, etc). The syntax of this command is:

```
COPY ( "pathname" )
```

The COPY command provides a convenient way to perform a sequence of frequently executed commands without having to enter the commands from the terminal each time. For example, when the debugger is initially invoked, a series of LOAD commands may be required to load all code segments comprising a system. The LOAD commands can be stored on a disc file and a single COPY command issued to execute all the LOAD commands. COPY commands cannot be nested; i.e. a COPY file must not contain COPY commmands.

6.3.5 Status Displays

Two commands are provided to display the status of processes being debugged. The DAP (Display All Processes) command is used to obtain the status of all processes in the system. The DP (Display Process) command shows the state of a single process.

6.3.5.1 Display All Processes - DAP Command. This command lists the status of every process currently known to the system. Consider the following example:

```
Status Summary of All Existing Processes

                         Site of                              Enabled    Stmt
     Process Name        Execution        Status        Pri   Traces     Bkpts?

 1   ASR733              runtime code   Wait Sema        6     P          no
 2     CSXIN             runtime code   Wait Sema        6     P          no
 6       FORMAT          runtime code > Wait Sema        6     S,P        yes
 3     CSXIN                            Terminated             S,R,P      yes
 7       FORMAT          runtime code   Hold             6     P          no
 4     CSXOUT            CSXOUT   7     Wait File        6     P          no
 5     CSXOUT            PUTCHA  10     Ready            6     P          no
 8     KEYIN             KEYIN    3     Active           6                no
 9     PRINT             PRINT    5     Ready            6                no
```

Each of the process names are indented to show their static lexical nesting level. The integer in the first column is a unique identification number for each process. The process can be referred to by this identification number in commands which require a process parameter. The site of execution indicates a routine name and statement number unless the process is currently executing in run-time support code. In this case, the site of execution displayed is "runtime code". The status column indicates the current status of each process. The (single) active process is indicated by "Active". If a process is ready to execute, its status is "Ready"; otherwise it is waiting with a status of "Wait". A waiting process is usually waiting for one of the following reasons: "Wait File" (waiting on file management services), "Wait Sema" (waiting on a semaphore), "Wait Prcs" (waiting for process management services), or "Wait Mem" (waiting for memory management services). The status indicates "Hold" if an HP command caused a process to be temporarily held from normal scheduling. If an HP command has been given for a process but the process cannot be held immediately (for example, it may be waiting on a semaphore), the status of the process is displayed followed by "(h)" which indicates a pending hold is to take place, immediately before the process would otherwise become the active process. The default process is indicated by the character ">" which immediately preceeds the status. The PRI column contains the priority of each process. The next column lists the kind of traces enabled for each process where P is a process scheduling trace, S is a statement trace, and R is a routine entry/exit trace. The last column shows whether any statement breakpoints are set for each process.

6.3.5.2 Display Process - DP Command. This commnd displays a detailed status for a single process. The syntax of this command is:

```
DP (  process  )
```

If no process is specified, the default process is assumed. Consider the following example:

```
Static/Dynamic Calling Order for Process CSXIN(2)

     Stack size (bytes) = 1484
     Stack used (bytes) Maximum = 1440  Current = 1146

     Call Order           Name               Statement
          -            ASR733(1)                 -
          1              CSXIN                   5
          2                SETUP                 12
          3                COMMAN                27
          4                  LINEIN              7
          5                    GETCHA            8
```

The top line of the display indicates the process name and number of the process being displayed. The next line shows the maximum amount of stack space available for the process. The next line gives an indication of how much of the stack has been used and how much is currently used. Finally the names of all routines nested within the process and all ancestor processes are listed. Each name is indented to indicate its static lexical nesting level. The current statement number for each routine is also listed.

The call order, listed in the left column, represents the order in which the routines were called dynamically. This is the number that can be used as the value of a routine parameter in subsequent commands to specify a given routine. Note that routine number 1 is always the process being displayed. Any names which are displayed before routine number 1 are ancestor processes. The ancestors of a process must be displayed since their stack frames are accessible. To see the status of an ancestor process, another DP command can be given for the desired ancestor.

6.3.6 Breakpoints/Single Step

Microprocessor Pascal source statements are numbered by the compiler and the compiled code is instrumented with these numbers if the DEBUG compile option is turned on. This allows breakpoints to be set and reset for any Microprocessor Pascal statement. If two statements appear on the same source line, the statement number listed by the compiler is that of the first statement on the line. When a breakpoint is encountered, execution is suspended so the user can examine/modify the state of the system. Upon encountering a breakpoint, the debugger displays the following kind of message:

```
*** Breakpoint ***     pname(i).rname   Statement n
```

where "pname" is the name of the process, "i" is the process number, "rname" is the routine in which the breakpoint was encountered, and "n" is the statement number for the breakpoint.

When a breakpoint is encountered, the breakpoint message is issued before the specified statement is executed.

Breakpoints are associated with individual processes. Therefore, a routine which is called from two separate processes of the same name can be breakpointed at different statements depending on which process invoked the routine.

In addition to statement breakpoints, execution can be suspended at any statement by simply pressing the CMD key (this feature is currently supported only on DX host development systems, pressing the CMD key on a TX system causes the debugging session to terminate). When the CMD key is pressed, the debugger displays the following message before interacting with the user for further commands:

```
*** Anonymous Bkpt ***
```

6.3.6.1 Assign Breakpoint - AB Command. This command is used to assign a statement breakpoint to any routine. The syntax of this command is:

```
AB ( routine,  statement number  )
```

The routine parameter identifies the routine in which to set the breakpoint. The statement number, if given, specifies at which statement to breakpoint; the default statement number is 1.

6.3.6.2 Delete Breakpoint - DB Command. This command is used to delete a statement breakpoint from any routine. The syntax of this command is:

```
DB ( routine,  statement number  )
```

The routine parameter identifies the routine which contains the breakpoint. The statement number, if given, indicates which breakpoint to delete; the default statement number is 1.

6.3.6.3 Delete All Breakpoints - DAB Command. This comand is used to delete all breakpoints from any process in the system. The syntax of this command is:

```
DAB ( process )
```

The process parameter specifies the process in which breakpoints are to be deleted.

6.3.6.4 List Breakpoints - LB Command. This command is used to list all breakpoints set in the specified process. The syntax of this command is:

```
LB (  process  )
```

The process parameter is optional. If no process is specified, the default process is assumed. The list displays the process name, routine name and statement number for each breakpoint set in the specified process.

6.3.6.5 Single-Step Mode - SS Command. This command is used to perform single-step execution. The syntax of this command is:

```
SS (  process ,  flag  )
```

The process parameter is optional. If no process is given, the default process is assumed. The flag parameter must be one of the Boolean values TRUE or FALSE (T and F are also accepted as abbreviations). A TRUE value turns on single-step mode; a FALSE value turns off single-step mode. If the flag parameter is not given, a value of TRUE is assumed as the default. While in single-step mode, statements are executed one at a time. A breakpoint is forced between every statement. A message of the following form is displayed:

```
*** Single-Step ***     pname(i).rname   Statement n
```

where "pname" is the name of the process, "i" is the process number, "rname" is the routine name of the currently executing routine, and n is the statement number.

Any single-step message is written before the statement is executed. To execute the specified statement, a GO command or a blank command line must be entered.

6.3.7 Showing/Modifying Data

There are four kinds of variables that can be examined and modified using the debugger. These are stack variables, heap variables, common variables, and indirect variables (VAR parameters). Commands are also provided to examine and modify absolute memory locations.

The following example is the display resulting from a SF (show frame) command. The stack frame in the example happens to be 26 (hex) bytes in length.

```
A4DA (0000) FFFF 0000 0001 5341 4D50 4C45 2020 0000  (......SAMPLE  ..)
A4EA (0010) 0002 0004 0006 0008 000A 000C 000E 0010  (................)
A4FA (0020) 0012 0014 0016                           (......          )
```

The first word of every display line is the absolute memory address of the first data word displayed on the line. In the example, the data displayed on the first line starts at memory address A4DA. Immediately following the address is the displacement into the specified stack, heap, or common area. The displacement is enclosed in parentheses. In the example, the displacement for the first line is zero (0000). For consistency throughout the display, the displacement is given using a hexadecimal format. Each line contains up to eight words of data. At the end of each line, the data is displayed as a string of 16 ASCII characters enclosed in parentheses; those bytes which represent non-printable ASCII characters are displayed as a period. Notice that the only printable characters in this example are those for the string "SAMPLE " which starts at displacement 6.

6.3.7.1 Show Frame - SF Command. This command is used to display a stack frame (or portion thereof). The syntax of this command is:

```
SF ( routine , displacement , length )
```

A stack frame is created each time a routine is entered. The stack frame contains the parameters and local variables for the routine. Using the stack displacements listed by the compiler, the value of any parameter or variable may be found by displaying the appropriate stack frame.

The first parameter specifies the stack frame to be displayed. If the routine parameter is not given, the latest routine called as part of the default process is assumed to be the default routine. To show a stack frame of a routine which has multiple instances, the dynamic calling number must be given (this number can be found using the DP command, see previous description). If a single routine name is given, it is assumed that the routine is in the default process. To show the stack frame of a routine in a different process, the process name should be specified first followed by a period, followed by the routine name. For example "CSXIN.SETUP" could be used to show the stack frame for the routine SETUP in the process CSXIN.

The optional second parameter specifies the byte displacement into the stack frame at which to start the display. The displacement corresponds to the number listed by the compiler in the left margin for each variable. The default displacement is zero. Negative displacements are not allowed.

By default, the entire stack frame is displayed. This can be overriden by specifying the length, in bytes, to be displayed. If the displacement and length extends beyond the stack frame, only the bytes of the specified stack frame are displayed.

6.3.7.2 Show Heap - SH Command. This command is used to show a heap packet. The syntax of this command is:

```
SH ( address , displacement , length )
```

The contents of any heap packet may be displayed with the show heap command. A heap packet is identified by an absolute memory address which is specified by the first parameter to this command. This address can be determined by finding the value of the specific pointer variable (using the SF command) which was returned by NEW. If no parameter is given to SH, all heap packets (if any) for the default process are displayed.

The second parameter is the byte displacement into the packet at which to start the display. If omitted, a displacement of zero is used. Negative displacements are not allowed.

The third parameter is the number of bytes to display. If omitted, the entire heap packet is displayed. If the displacement and length extends beyond the heap packet, only the bytes within the packet are displayed.

6.3.7.3 Show Common - SC Command. The contents of a named common area specified by COMMON and ACCESS declarations may be displayed using this command. The syntax of this command is:

```
SC ( common name, displacement , length )
```

The first parameter is the name of the common area.

The second parameter is the byte displacement into the common at which to begin the display. If omitted, zero is assumed. Negative displacements are not allowed.

The third parameter is the number of bytes to be displayed. If omitted, the entire common area is displayed. If the displacement and length extends beyond the length of the common, only the bytes in the specified common are displayed.

6.3.7.4 Show Indirect - SI Command. This command is used to display the value of an indirect variable. The syntax of this command is:

```
SI ( routine, displacement,  length  )
```

An indirect variable is a parameter passed by reference (i.e. a VAR parameter). The indirect variable is addressed through a cell in the stack frame of the specified routine at the specified displacement. The length parameter is the number of bytes to be displayed. If no length is given, one word is displayed.

6.3.7.5 Show Memory - SM Command. This command is used to show the contents of an absolute memory location. The syntax of this command is:

```
SM ( address,  length  )
```

The first parameter is the address of the memory area to be displayed. The second parameter is the length, in bytes, to be displayed. If no length is specified, one word is displayed.

6.3.7.6 Modify Frame - MF Command. This command is ued to modify a single word value in the specified stack frame. The syntax of this command is:

```
MF ( routine,  displacement ,  verify value , new value )
```

The first two parameters have the same meaning as in the SF command. The verify value is the old value for the word to be modified. If the verify value does not match the old value, an error occurs. The final parameter is the new value for the word. As in all of the modify commands, no check is performed if the verify value is omitted; the specified location is modified regardless of its current contents.

6.3.7.7 Modify Heap - MH Command. This command is used to modify a single word value in heap. The syntax of this command is:

```
MH ( address,  displacement ,  verify value , new value )
```

The first two parameters have the same meaning as in the SH command. The verify value is the old value for the word to be modified. If the verify value does not match the old value, an error occurs. The final parameter is the new value for the word.

6.3.7.8 Modify Common - MC Command. This command is used to modify a single word in a common. The syntax of this command is:

```
MC ( common name,  displacement ,  verify value , new value )
```

The first two parameters have the same meaning as in the SC command. The verify value is the old value for the word to be modified. If the verify value does not match the old value, an error occurs. The final parameter is the new value for the word.

6.3.7.9 Modify Indirect - MI Command. This command is used to modify a single word indirect variable (VAR parameter). The syntax of this command is:

```
MI ( routine, displacement,  verify value , new value )
```

The first two parameters have the same meaning as in the SI command. The verify value is the old value for the word to be modified. If the verify value does not match the old value, an error occurs. The final parameter is the new value for the word.

6.3.7.10 Modify Memory - MM Command. This command is used to modify the contents of any single (word) location in memory. This is the most dangerous command in the debugger's vocabulary so extreme caution should be exercised. The syntax of this command is:

```
MM ( address,  verify value , new value )
```

The first parameter is the address of the word to be modified. The second parameter is used to verify the old value for the word. If the verify value does not match the old value, an error message occurs. The third parameter is the new value for the word.

6.3.8 Tracing Commands

There are three kinds of tracing available. Tracing is useful to examine the behavior of the scheduling algorithm, to observe the dynamic behavior of routine calls and exits, and to determine the actual control flow of statements. Trace data is written to the user's terminal and to a log file so a hard copy can be obtained upon termination of the debugging session. For conveniece, however, the display of trace data at the terminal can be suppressed using the TOFF (Trace OFF) command. It may be enabled again using TON (Trace ON).

6.3.8.1 Trace Process scheduling - TP Command. This command turns process scheduling tracing on or off for the specified process. The syntax of this command is:

```
                    TP (  process ,  flag  )
```

The process parameter is optional. If no process is given, the default process is assumed. The flag parameter must be one of the Boolean values TRUE or FALSE (T and F are also accepted as abbreviations). A TRUE value enables tracing; a FALSE value disables tracing. If the flag parameter is not given, a value of TRUE is assumed as the default. Process scheduling tracing causes a trace to be generated each time the given process is scheduled, i.e. becomes active or inactive. This enables the user to examine the behavior of the scheduling algorithm.

Example:

```
             .
             .
             .
    *** Trace ***      ASR733(1)        Process Active
    *** Trace ***      ASR733(1)        Process Inactive
    *** Trace ***      CSXIN(2)         Process Active
    *** Trace ***      CSXIN(2)         Process Inactive
    *** Trace ***      FORMAT(6)        Process Active
    *** Trace ***      FORMAT(6)        Process Inactive
    *** Trace ***      CSXOUT(4)        Process Active
             .
             .
             .
```

6.3.8.2 Trace Routine entry/exit - TR Command. This command turns routine entry/exit tracing on or off for the specified process. The syntax of this command is:

```
                    TR (  process ,  flag  )
```

The process parameter is optional. If no process is specified, the default process is assumed. The flag parameter must be one of the Boolean values TRUE or FALSE (T and F are also accepted as abbreviations). A TRUE value enables tracing; a FALSE value disables tracing. If the flag parameter is not given, a value of TRUE is assumed as the default. When routine tracing is enabled for a process, each routine entry or exit is traced, excluding calls to run-time support code. The trace information consists of the process name and number, the routine name, and whether the routine was entered or exited.

Example:

```
               .
               .
               .
    *** Trace ***        CSXIN(2).CSXIN        Routine Entry
    *** Trace ***        CSXIN(2).SETUP        Routine Entry
    *** Trace ***        CSXIN(2).COMMAN       Routine Entry
    *** Trace ***        CSXIN(2).LINEIN       Routine Entry
    *** Trace ***        CSXIN(2).GETCHA       Routine Entry
    *** Trace ***        CSXIN(2).GETCHA       Routine Exit
    *** Trace ***        CSXIN(2).LINEIN       Routine Exit
               .
               .
               .
```

6.3.8.3 Trace Statement flow - TS Command. This command turns statement execution tracing on or off for the specified process. The syntax of this command is:

```
TS ( process , flag )
```

The process parameter is optional. If no process is specified, the default process is assumed. The flag parameter must be one of the Boolean values TRUE or FALSE (T and F are also accepted as abbreviations). A TRUE value enables tracing; a FALSE value disables tracing. If the flag parameter is not given, a value of TRUE is assumed as the default. When this trace is enabled for a process, every time a statement instruction is encountered (statement instructions only exist in routines compiled with the DEBUG option), a new line is written to the trace file. The line contains the process name and number, the routine name, and the statement number that was executed.

The trace information for statements is written before the specified statement is executed.

Example:

```
            .
            .
            .
    *** Trace ***     CSXIN(2).CSXIN      Statement 1
    *** Trace ***     CSXIN(2).CSXIN      Statement 3
    *** Trace ***     CSXIN(2).CSXIN      Statement 4
    *** Trace ***     CSXIN(2).SETUP      Statement 1
    *** Trace ***     CSXIN(2).SETUP      Statement 2

            .
            .
```

6.3.8.4 Trace echo OFF - TOFF Command. This command disables all trace information from being displayed on the terminal. The TOFF command has no parameters. This command only affects the display of trace data at the interactive terminal; the trace information is still written to the log file. It is then possible to obtain an execution trace of a system when it may be impractical to examine a large amount of trace data interactively.

6.3.8.5 Trace echo ON - TON Command. This command enables the display of all trace information at the terminal. The default is to have tracing enabled at the terminal.

6.3.9 Monitor Process Scheduling

The commands described in this section give the user limited control over the scheduling of processes. The user can "hold" a process, which temporarily blocks it from becoming active until it is explicitly "released". Also, process breakpoints can be assigned, forcing a breakpoint immediately before a process would become active (scheduled for execution).

6.3.9.1 Select Default Process - SDP Command. This command is used to select the default process. The syntax of this command is:

```
SDP ( process )
```

The single parameter indicates the process to be selected as the default process. If a name is given, it refers to the youngest instance of the process. A number may be given to select an older instance of a particular process. The number of a process can be found in the left column of the display from the DAP (display all processes) command.

The specified process is used as a default in subsequent commands which have a process parameter; this provides a convenient shorthand notation for many subsequent commands.

The user's SYSTEM is implicitly chosen to be the default process when the debugger is invoked. The SDP command may be used to change this default. However, the default process cannot be selected before the process has been created. Also, since many commands allow one to implicitly reference the default process, it is imperative that a default process exist at all times. For this reason, whenever the default process terminates, a new default process (the currently active process) is chosen by the debugger as the default process and the user is notified of this change. The message is of the form:

```
*** Default Process Terminated ***  name(n)
*** New Default Process ***  new_name(m)
```

where "name(n)" is the name and process number of the terminated process and "new_name(m)" is the name and process number of the new default process.

6.3.9.2 Assign Breakpoint to Process - ABP Command. This command is used to set a process breakpoint. The syntax of this command is:

```
ABP ( process )
```

When a process breakpoint is set, a breakpoint occurs just before the process becomes active. Process breakpoints persist until they are deleted using DBP or until the process terminates.

6.3.9.3 Delete Breakpoint from Process - DBP Command. This command is used to delete a process breakpoint. The syntax of this command is:

```
DBP ( process )
```

6.3.9.4 Hold Process - HP Command. This command is ued to temporarily suspend a process. When a process is held, it is not eligible for execution until an explicit release command is given by the user. The syntax of this command is:

```
HP ( process )
```

This command can be issued regardless of the current status of a process. For example, a process may be waiting on a semaphore when an HP command is issued. Whereas the HP command does not take affect until the semaphore is signaled, this delay remains transparent to the user.

6.3.9.5 Release Process - RP Command. This command releases a process which was previously "held" by a HP command. It makes the specified process eligible for execution via the normal scheduling algorithm. The syntax of this command is:

```
RP ( process )
```

6.3.10 Interprocess File Simulation

Executive RTS interprocess files are simulated by the debugger with extensions allowing host DX (or TX) files and devices to be connected to RTS channels as producers or consumers of components (See section 10). Normally, these "host connections" are made implicitely according to channel names. If a channel has a name which is also the name of a host file or device, that channel will be implicitely connected to the host file. The standard procedure SETNAME and the standard function FILENAMED may be used to direct a host connection. For example,

```
setname( f, 'dsc2:test/dta' );
```

will cause the file F to have the name "DSC2:TEST/DTA". When F is RESET, it is connected to a channel of the same name, which is connected to the TX file "DSC2:TEST/DTA". A READ(F,USER_VARIABLE) will cause a host request to be made to receive the next component from the TX file "DSC2:TEST/DTA".

The capability of connecting channels to host files allows the user to view the inputs and outputs of a process in isolation from other processes. It is, however, undesirable to require that the user's source code explicitely reference host files. Therefore, two debug commands are provided with which the user may specify a run-time mapping of internal channel names to external host file names. Using the commands CIF (Connect Input File) and COF (Connect Output File) the user can, at debug time, specify that a particular host file is to act as a producer and/or consumer to a particular channel.

Note that connection of an output file to ME does not imply that information written to the output file is sent to the debugger log file.

6.3.10.1 Connect Input File - CIF Command. This command allows the user to specify a particular host file that is to be used as input (a producer of components) to a particular channel. The host connection is actually made when the first reading file variable connects to the channel. If the channel already has reading files connected to it, the host connection takes place when all reading files connected to the channel become disconnected (by calling RESET, REWRITE, or CLOSE) and another reading file connects to the channel. Once the connection has taken place, a READ from a connected file variable will return the next buffered channel component if it exists. If no components are buffered in the channel when the READ is done, a host request is made to transfer the next component from the connected host file. The syntax of this command is:

```
CIF ( "internal channel name", "input host file name" )
```

Both the INTERNAL CHANNEL NAME and INPUT HOST FILE NAME are string parameters and, therefore, must be enclosed in double quotes. The INTERNAL CHANNEL NAME is the internally known channel name to which the host file named INPUT HOST FILE NAME is to be connected as a producer of components. An example of the use of this command is:

```
CIF("f","dsc2:simulat/dta")
```

This causes the host file DSC2:SIMULAT/DTA to be connected as a producer of components to the channel named F when the first consuming file variable connects to the channel. If the INPUT HOST FILE NAME is the user's terminal, "me", the user is prompted for input whenever host requests are made.

6.3.10.2 Connect Output File - COF Command. This command allows the user to specify a particular host file that is to be used as output (a consumer of components) from a particular channel. The host connection is actually made when the first writing file variable connectes to the channel. If the channel already has writing files connected to it, the host connection takes place when all writing files connected to the channel become disconnected (by calling RESET, REWRITE, or CLOSE) and another writing file connects to the channel. Once the connection has taken place, a WRITE to a connected file variable will transfer the component to the connected host file. The syntax of this command is:

```
COF ( "internal channel name", "output host file name" )
```

Both the INTERNAL CHANNEL NAME and OUTPUT HOST FILE NAME are string parameters and, therefore, must be enclosed in double quotes. The INTERNAL CHANNEL NAME is the internally known channel name to which the host file named OUTPUT HOST FILE NAME is to be connected as a consumer of components. An example of the use of this command is:

```
COF("f","dsc2:simulat/out")
```

This causes the host file DSC2:SIMULAT/OUT to be connected as a consumer of components from the channel named F when the first producing file variable connects to the channel.

6.3.11 Interrupt Simulation - SIMI command

This command is used to simulate an interrupt at the specified level. The syntax of this command is:

```
SIMI ( level )
```

If a process is waiting on the specified interrupt, execution of this command causes the process to react as if the interrupt had actually occurred. If the processor mask does not currently allow the interrupt, the interrupt is maintained by the debugger as a "pending interrupt" until the mask is raised, at which time the simulated interrupt is serviced.

6.3.12 Selection of CRU mode - CRU Command

The standard procedures TB, LDCR, SBO, SBZ, and STCR may be called from any point in the user's program. The CRU debugger command may be used to control how CRU instructions are to be handled. The syntax of this command is:

```
CRU (  process , cru mode)
```

The process parameter specifies to which process the command applies. If omitted, the default process is used. The second parameter specifies how CRU instructions are to be handled. The value of the second parameter must be one of the following: EXECUTE, OFF, or DEBUG. If EXECUTE is specified, CRU instructions are directly executed. If OFF is specified, all CRU instructions are ignored. If DEBUG is specified, all CRU input and output is simulated by the user. The following paragraphs describe how this interaction takes place. The default mode for CRU instructions is DEBUG.

6.3.12.1 Test CRU Bit - TB. The following message and prompt for input is displayed:

```
"Test CRU Bit"  Address = nnnn, True or False?:
```

The user is expected to respond with a TRUE or FALSE value.

6.3.12.2 Load CRU Value - LDCR. Te following message is displayed:

"Load CRU Value" Address = nnnn, Width = nn, Value = nnnn

This message displays the value that is to be loaded into the specified CRU address.

6.3.12.3 Set Bit to Logic One - SBO. The following message is displayed:

"Set Bit to One" Address = nnnn

This message displays the CRU address to be set to the value one.

6.3.12.4 Set Bit to Logic Zero - SBZ. The following message is displayed:

"Set Bit to Zero" Address = nnnn

This message displays the CRU address to be set to the value zero.

6.3.12.5 Store CRU Value - STCR. The following message and prompt for input is displayed:

"Store CRU Value" Address = nnnn, Width = nn, Value?:

The user is expected to respond with the value to be stored.

6.4 ERROR MESSAGES

This section contains a list of debugger error messages with an explanation for each one.

6.4.1 Command Syntax Errors

When a command is improperly formed or cannot be recognized by the debugger, one of the following error messages is given.

command is not valid

This error occurs when the specified command is not currently valid. The HELP command can be used to display a list of all debugger commands. However, when the debugger issues its initial prompt <> , only a limited subset of these commands are valid. The initially valid commands are: HELP, GO, QUIT, LOAD, SE, DEBUG, and COPY.

incomplete command

This error occurs when a command is improperly terminated. If a command has parameters, the parameter list must be enclosed in parentheses.

extra characters will be ignored

This error occurs when the command contains extra characters to the right of an otherwise well-formed command. This error is usually only a warning.

too many parameters

This error occurs when the command contains too many parameters. Use the HELP command to check the number and meaning of parameters for the command.

missing parameter(s)

This error occurs when the command is missing one or more required (non-optional) parameters. Use the HELP command to check the number and meaning of parameters for the command.

wrong kind of parameter

This error occurs when the command contains a parameter that is the wrong kind. For example, an integer constant appearing where an identifier is expected.

parameter syntax error

This error occurs when a command parameter is improperly formed. Parameters can only be one of the following: integer constant, identifier, string (delimited by double quotes), integer constant or identifier followed by a period followed by an integer constant or identifier.

parameter must be a Boolean value

This error occurs when a Boolean-valued parameter was expected and not received (either TRUE, FALSE, T, or F). Use the HELP command to check which parameter is supposed to be a Boolean value.

unrecognized CRU mode

This error occurs when a CRU command is given and the CRU mode is not EXECUTE, DEBUG, or OFF.

pathname must be a string

This error occurs when a file pathname parameter (e.g. COPY(pathname)) is not specified as a string, between double quotes.

6.4.2 Breakpoint Command Errors

The following errors may occur when using the breakpoint commands, AB, DB, DAB, and LB. Some of these are merely warnings. Note that LB (list breakpoints) can be used to list the breakpoints that are set for a given process.

breakpoint already assigned

This message indicates that a breakpoint is already assigned to the specified routine and statement number. This message is a warning (no action is performed).

no such breakpoint

This error indicates an attempt was made to delete a non-existent breakpoint. This message is a warning (no action is performed).

no breakpoints set

This message is the result of a LB (list breakpoints) command when there are no breakpoints set.

non-existent statement number

This message indicates that a non-existent statement number was referenced when attempting to assign a breakpoint. Check the compiler listing to ensure that the statement number in the assign breakpoint command does not exceed the maximum statement number listed.

6.4.3 Show/Modify Command Errors

The following errors can result from the use of the show and modify commands, SF, SH, SC, SI, SM, MF, MH, MC, MI, and, MM.

no such routine

This message indicates an error in a routine parameter. If a routine name was specified as a parameter, the name was probably misspelled. If a dynamic calling number was specified, check to make sure there is such a dynamic calling number listed using the DP (display process) command.

no such process

This message indicates an error in a process parameter. If a process name was specified as a parameter, the name was probably mis-spelled. If a process number was specified, check to make sure that the process exists (use the DAP command).

stack frame not found

This error occurs when the stack frame for a routine does not exist. Either the routine parameter is in error (bad name or dynamic calling number) or the routine is not currently active (has "returned" or has never been called).

invalid heap packet

This error occurs when the address of a heap packet is incorrect. Only heap packets which have been allocated dynamically (by NEW) can be displayed or modified.

common not found

This error occurs when the common name parameter is incorrect. Double check the common name given with the one declared in your source code.

bad displacement

This error occurs when a bad displacement into a memory area is specified. This can happen in one of the following cases: the displacement is beyond the length of the specified stack frame, the displacement is beyond the length of the specified heap packet, or the displacement is beyond the length of the specified common area.

verify error

This error occurs when a modify command (MF, MH, MC, MI, MM) contains a verification value and the value to be modified does not match the verification value. In this case, the memory location is not modified with the new value.

6.4.4 Miscellaneous Errors

The following errors are more general in nature than the ones discussed previously.

cannot get system memory

This error indicates that sufficient system memory space is not available. The user's system memory requirements should be examined and possibly modified.

too many modules in segment

This error occurs when a single segment contains more than 256 routines (internal and external) and commons. This is a fixed size constraint. To avoid this error, split the segment into two separate segments so that the total number of routines and commons in each one does not exceed 256.

no interpretive code found, use LOAD command

This is a warning message which indicates the file containing the interpretive code for the user system was not found. The LOAD command can be used to load a previously saved code segment.

warning: module not saved with DEBUG information

This is a warning message which indicates that the module being loaded does not contain debugging information. If desired, the module should be re-saved with debug information.

unresolved externals, use LOAD or SE

This error occurs when the user's system contains references to external routines which have not been resolved yet. The SE command displays a list of unresolved routine names. The LOAD command can be used to load saved code segments.

invalid SAVE file

This error occurs when the file given in a LOAD command either does not exist or cannot be opened. The file name given by the user is probably incorrect.

cannot redefine an external

This error occurs when the debugger detects two external routine definitions in separate code segments. The first definition for an external routine is the one used in all subsequent references.

cannot open COPY file

This error occurs when a COPY command is given and the COPY file either does not exist or cannot be opened for some reason.

cannot nest COPY commands

This error occurs when an attempt is made to include COPY commands in COPY files. Nested COPY commands are not allowed.

already specified for debug

This error occurs when the user performs two separate DEBUG commands for the same process.

not specified for debug

This error occurs when the user specifies DEBUG(process, false) and the process was never specified for debugging.

process is not held

This error occurs when an RP (release process) command is given for a process which is not presently held.

process is already held

This error occurs when an HP (hold process) command is given for a process which is already held.

no process waiting on interrupt

An attempt was made to service an interrupt and there was no process waiting on the interrupt.

interrupt level must be in range 1..15

This error occurs when a SIMI command parameter is not in the range 1 to 15.

more urgent interrupt in progress

This error occurs when a SIMI command is given and the interrupt cannot be serviced because a more urgent interrupt is in progress.

waiting process less urgent than interrupt

This error occurs when a SIMI command is given and the only waiting interrupt process is less urgent than the interrupt level specified.

bad internal file name

This error occurs if the internal file name parameter in a CIF or COF command is improperly formed.

bad external file name

This error occurs if the external file name parameter in a CIF or COF command is improperly formed.

internal error, or unknown error

These errors should not normally occur. The probable cause for such errors is that some data structures used by the debugger were destroyed either explicitly (using the modify memory command) or implicitly by the user's system. Please contact the staff of Texas Instruments if such an error occurs and its cause cannot be determined.

SECTION VII

CONVENTIONAL PASCAL PROGRAM EXECUTION

7.1 EXECUTION OVERVIEW

The Microprocessor Pascal System provides an executive which will load a conventional Pascal program and execute it without any interactive debugging capability. The program must be a single PROGRAM without any SYSTEM or PROCESSes and no Executive Run Time Support calls. Only standard Micrprocessor Pascal System routines may be called. This capability is useful when writing a general utility in Pascal or when testing an algorithm on a set of data.

7.2 PROGRAM SEGMENTS

The executive prompts the user as to whether he wants to execute the last program which was compiled, or one which was saved. This prompt appears as follows:

```
        DO YOU WANT "PCODE" LOADED? YES
```

Normally the program to be executed is the last program which was compiled, in which case, the user should simply press the RETURN key. Otherwise if a utility program which was saved is to executed, "NO" should be entered. When this is done, the name of the segment for the program must be entered in response to the following prompt:

```
        ENTER PATHNAME OF MAIN SEGMENT:
```

If the program to be executed has any external references, external segments may be loaded. The execute program will prompt for the segment pathname as follows:

```
        ENTER PATHNAME OF EXTERNAL SEGMENT:
```

If the external references should be ignored, then the RETURN key should be entered without specifing a pathname.

Please note that the concurrent characteristics specified in the program are ignored. The amount of stack and heap given to the user's program is one area of memory and used as needed for either stack or heap. In the DX version, the combined amount of stack and heap is requested, but in the TX version the total remaining amount of memory is given to the user for stack and heap.

7.3 EXECUTION MESSAGES

After the complete program has been loaded, control is given to the user's program which is indicated by the following message:

```
EXECUTION BEGINS
```

After the user's program has completed execution, control is given back to the executive and the amount of stack and heap in bytes which was used by the user's program is indicated by the following message:

```
STACK USED = xxxxx  HEAP USED = xxxxx
```

If the user's program terminated abnormally then the following message precedes the stack and heap utilization message:

```
ABNORMAL USER PROGRAM TERMINATION
```

7.4 I/O SUPPORT

The user's program may use Microprocessor Pascal I/O to access host files. All Microprocessor Pascal files are supported including TEXT, sequential, and RANDOM files. All MESSAGEs are displayed on the terminal. The destination of the OUTPUT file is specified during the initial prompt. All other files may be connected to host file via the SETNAME procedure. If the SETNAME procedure is not used, the run-time support will prompt the user for the pathname to be connected to the file as follows:

```
INPUT PATHNAME FOR "filename" :
```

Note: The pathname may not contain DX/10 synonyms because no synonym mapping is performed before the file is opened.

7.5 RUN-TIME SUPPORT ERROR MESSAGES

If an error is found during an I/O operation, the following message will be generated:

```
I/O ERROR : ee ss  NAME= filename
```

where "ee" is the I/O error type, described in Appendix E, "ss" is the I/O service call status, and "filename" is the file variable name.

If an error is found during a TEXT I/O operation, the following message will be generated:

```
TEXT FILE I/O ERROR : ee  NAME= filename
```

where "ee" is the TEXT I/O error type, described in Appendix E, and "filename" is the file variable name.

The following error messages are generated by the heap management routines when an error is found.

HEAP OVERFLOW - no more heap space is available to allocate the current heap packet.

INVALID HEAP PACKET POINTER - the pointer being DISPOSEd does not point to a valid heap packet.

INVALID HEAP PACKET LENGTH - the length of the heap packet being DISPOSEd is bad.

Normally when any error is found by the run-time support routines, the user's program is halted and control is given back to the execute program. When this is done the following message is generated:

```
HALT CALLED
```

7.6 ABNORMAL TERMINATION MESSAGES

If the user's program is terminated abnormally, the following message will be displayed which indicates the type of error which occurred.

```
*** RUN TIME ERROR DETECTED *** reason
```

The "reason" for the termination will be one of the following:

INVALID OPCODE - an illegal interpretive code operator was found

STACK OVERFLOW - the user's program consumed the entire allocated stack area

INVALID CALL - a procedure was called which was unresolved

DIVIDE BY 0 - an attempted divide by zero

FLOATING POINT - a floating point underflow or overflow was detected

SET RANGE - a set element less than 0 or greater than 1023 was detected

ASSERT - an ASSERT statement failed

CASE - no CASE statement alternative was found

SUBSCRIPT - an array subscript expression was not within the declared bounds

POINTER - a pointer equal to NIL was being referenced

SUBRANGE - an assignment of a value to a subrange variable was not within the declared bounds

"HALT" CALLED - a run-time support error was found

After the error message is generated, a trace back of the user's program is generated which indicates where the error occurred. The first line listed was where the error occurred and the rest indicate how the routine was called. If the program was loaded from interpretive code or from a segment which was saved with "debug" information, the name of user routines will be listed. If this was not done, the name will not be listed. If the program was compiled with the "debug" option, then the statement number will be listed. An example of the trace back listing which is generated is shown below:

```
-- RUN TIME SUPPORT
-- ROUTINE = routine name  STATEMENT NUMBER = nn
```

The first line indicates that the routine name was not known, and the second line shows the format for those routines where the name is known.

When using the DX version of the execute program, a dump of each routine's stack frame is given following the above header. This dump is similar to the dump created by the Host Debugger.

SECTION VIII

MICROPROCESSOR PASCAL SYSTEM

8.1 LANGUAGE VOCABULARY AND REPRESENTATION

Each Microprocessor Pascal System system is composed of symbols from a finite vocabulary. The vocabulary consists of identifiers, numbers, strings, operators, and keywords. They are called lexical symbols, and in turn are composed of sequences of characters. Their representation therefore depends on the underlying character set.

8.1.1 Character Set

The Microprocessor Pascal System character set consists of the letters A - Z, a-z, the digits 0 - 9, and the special characters

    + - * / " . , ; : = '<> ( )[ ] {} #^ @

These characters are used to form special symbols which have a fixed meaning in the language.

8.1.2 Special Symbols

Special symbols are used for operators and delimiters. The special symbols are:

```
+   -   *   /   :=   =  <> <  <= >=  >  :: @^
(   )  [   ]  {...}  ..   .   ,   ;   :   '   "  #
```

Note: (. .) is a substitute for [ ] which is used to delimit array indices and sets, (* *) is a substitute for { } which is used to delimit comments, and @ is a substitute for ^ which is used with pointer types. These alternate symbols are provided since the symbols they replace are not available on all systems.

8.1.3 Keyword Symbols

Keyword symbols are reserved words with a fixed meaning; they may not be declared as identifiers. They are written as a sequence of letters and are interpreted as a single symbol.

ACCESS
AND
ANYFILE
ARRAY
ASSERT
BEGIN
BOOLEAN
CASE
CHAR
COMMON
CONST
DIV
DO
DOWNTO
ELSE
END
ESCAPE
FALSE
FILE
FOR
FUNCTION
GOTO
IF
IN
INPUT
INTEGER
LABEL
LONGINT
MOD
NIL
NOT
OF
OR
OTHERWISE
OUTPUT
PACKED
PROCEDURE
PROCESS
PROGRAM
RANDOM
REAL
RECORD
REPEAT
SEMAPHORE
SET
START
SYSTEM
TEXT
THEN
TO
TRUE
TYPE
UNTIL
VAR
WHILE
WITH

8.1.4 Identifiers

Identifiers are used as names denoting user defined or predefined entities. An identifier consists of a letter or $, followed by any combination of letters, digits, $, or _. Upper- and lower-case letters are allowed but a lower-case letter is treated the same as if it were the corresponding upper-case letter. For example, the identifier DATA_SIZE is the same identifier as Data_Size. A maximum length is imposed by the restriction that identifiers may not cross card boundaries and hence may not be more than 72 characters long. All characters in an identifier are significant, however, an identifier used to denote a system, program, process, procedure, function, or common should be unique within the first 6 characters. To avoid conflict with run-time support routine names, user routine names should not contain any $ characters. Also, if AMPL is to be used for target debugging, routine names should not contain any _ characters.

Examples:

Legal Identifiers

X
$VAR
LONG_IDENTIFIER
NUMBER_3
READ

Illegal Identifiers

```
ARRAY       ( Reserved word )
_ROOT3      ( Can't start with _ )
3RDVAL      ( Can't start with 3 )
MAX VALUE   ( Can't contain blank )
TOTAL-SUM   ( Can't contain - )
```

Note: Some identifiers are standard, that is, they are predeclared

Some identifiers are standard, that is, they are predeclared with a given meaning. However, they may be redefined by the user, in which case the standard meaning no longer applies. For example, if the standard routine name READ is redefined, the standard routine READ may not be called.

8.1.5 Constants

Constants are either unsigned integer constants, long integer constants, real constants, string constants, or character constants.

8.1.5.1 Integer and Long Integer Constants. An integer constant is written as a sequence of decimal digits. An integer constant may also be a sequence of hexadecimal digits preceded by a # sign. Either form may be followed by an L to indicate a LONGINT constant.

Examples:

```
      Legal INTEGER and LONGINT constants
           133
           #26B
           #AFL
           00022
           252410L
```

8.1.5.2 Real Constant. A real constant is either written as two sequences of decimal digits separated by a decimal point or using an exponential notation. Note that a decimal point must be surrounded on both sides by decimal digits. The general syntax allowed is:

```
      nnn.nnn   or   nnn.nnnEmm   or   nnnEmm
```

The number nnnEmm represents the real number nnn times 10 to the power mm.

Examples:

```
      Legal REAL constants
           11.75
           726E2
           9.8E-4
           102.4E+2

      Illegal Numbers
           .005      ( Decimal point not surrounded by digits )
           75.E-2    ( Decimal point not surrounded by digits )
           2.0E1.5   ( REAL exponent not allowed )
           #47A.2    ( HEX notation illegal with decimal point )
```

8.1.5.3 String Constant. A string constant is written as a sequence of characters enclosed by apostrophes. A string cannot be longer than 70 characters. Any ASCII character code may be represented in a string by a # followed by two hexadecimal digits. This enables unprintable characters to be included in strings. Within a string, ' is represented by '' and # is represented by ##. A string constant is of the following type:

```
        PACKED ARRAY [1.. length] OF CHAR
```

where length is the number of characters in the string constant.

Examples:

```
        Legal STRING constants
              '  THIS IS A STRING  '
              'UNPRINTABLE CHARACTER #0D'
              'EXAMPLE ##3'
              'CAN''T'
```

8.1.5.4 Character Constant. A character constant is written as one character enclosed by apostrophes. The character may be represented by two hexadecimal digits preceded by a #. As in a string constant, the character ' is represented by '' and # is represented by ##.

Examples:

```
        Legal CHARACTER constants
              '7'
              'p'
              ''''
              '+'
              '#0F'
```

8.1.6 Separators

At least one separator must occur between any two constants, identifiers, keywords, or special symbols. No separator may occur within these elements, except that spaces may occur within strings. Separators are spaces, ends of lines, comments, or remarks. For example, in

```
                        WHILE X<10
```

a space separates WHILE and X. It is not equivalent to write:

```
                        WHILEX<10
```

A comment is any sequence of characters beginning with (* or { and ending with *) or }, except that (* or { does not begin a comment within a string. Comments may not be nested; a warning message is generated if an open comment symbol is found within a comment. A remark is any sequence of characters beginning with a " and extending to the end of the logical record, except that " within a string does not begin a remark.

Examples:

```
                        {This is a comment}
                        (* This is also a comment *)
                        " the rest of the line is a remark
```

8.2 DECLARATIONS

The text of a Microprocessor Pascal system consists of declarations of objects and a sequence of statements that operate on the declared objects. Objects which may be declared are labels, constants, data types, variables, commons, programs, processes, and routines (procedures and functions). Declarations are used to give unique names to each object. In general, the identifier naming an object must be explicitly declared before it may be used in any statement. This redundancy enables the compiler to detect spelling errors and the inconsistent use of declared objects. In addition to explicit declarations, there are three kinds of implicit declarations, namely, FOR control variables, ESCAPE labels, and WITH variables.

Each declaration has a scope, which can be thought of as the range of the system text over which the declaration is effective. The unit of scope for explicitly declared objects is a Microprocessor Pascal System module (system, program, process, procedure, or function). This means that once an identifier has been declared to denote a variable, for example, the variable is accessible by means of this identifier throughout the module where the declaration appeared, unless the identifier is redeclared within some inner unit of scope (module).

Other units of scope include FOR statements (implicit declaration of the control variable), record type declarations, structured statements (implicit declaration of ESCAPE labels), and WITH statements (implicit declaration of synonyms for record variables).

Extent is the time during system execution that a computational quantity may be considered to exist. The extent of a variable is the time during which space is allocated for the variable. The extent of all statically declared quantities is the duration of execution of the unit of scope in which they are declared, with the exception of COMMON variables, whose extent is the entire system execution.

The extent of dynamically allocated variables is that portion of system execution between the call of NEW which creates them and the call (if any) to DISPOSE which frees the space allocated to them.

Within an Microprocessor Pascal system, modules may only be nested to a maximum lexical level of 10.

8.2.1 System Declaration

A Microprocessor Pascal system is the superstructure which contains all the programs and processes of a single user task. The system declarations define all globally known items, such as constants, types, commons, and utility routines. All programs are also defined within the system.

The syntax of a system is as follows:

```
SYSTEM <identifier> ;
  <system block> .
```

where <system block> is as follows:

```
<label declaration part>
<system data declarations>
<access declaration part>
<system routine declarations>
<process body>
```

where <system routine declarations> may be any of the following: .

```
<program declarations>
<procedure declarations>
<function declarations>
```

and <process body> is of the form:

```
BEGIN <concurrent characteristics>
  <statement list>
END
```

The <concurrent characteristics> are described in Section 8.2.12.

The <system data declarations> are described in Section 8.2.3; they do not include any <variable declaration part>s. In the <system block>, any of the declaration parts may be missing.

Example:

```
SYSTEM EXAMPLE_SYSTEM;
  LABEL . . .          " label declarations
  CONST . . .          " constant declarations
  TYPE . . .           " type declarations
  COMMON . . .         " common declarations
  PROCEDURE . . .      " utility procedure declarations
  FUNCTION . . .       " utility function declarations
  PROGRAM . . .        " program declarations
BEGIN                  " system body
 {# concurrent characteristics}
END;
```

8.2.2 Label Declaration Part

Label declarations specify all labels which may be referenced within the body section of a module by a GOTO statement.

Label declarations are of the form:

```
          LABEL <integer list>;
```

The <integer list> is simply a list of unsigned integer constants separated by commas.

Example:

```
       LABEL 3,15;
```

A label which marks a statement must be declared in the label declaration section of the module. Only one statement may be prefixed with a given label and labels may not be multiply declared in a single scope.

8.2.3 Data Declarations

The <data declarations> section consists of a combination of four separate declaration parts:

```
          <constant declaration part>
          <type declaration part>
          <variable declaration part>
          <common declaration part>
```

These declaration parts may be repeated any number of times within a module and may appear in any desired order. The individual declaration parts are described in the subsequent sections. The <system data declarations> are the same as the <data declarations> except that they may not include any <variable declaration part>s.

To facilitate the use of the ?COPY statement, Microprocessor Pascal allows declaration parts, within the <data declarations>, to appear in any order. Caution must be taken when this is done because it increases the possibility of unintentionally redeclaring some data item(s). The following example illustrates this point.

Example:

```
       SYSTEM TEST;
         TYPE T = . . . ;
         . . .
         PROGRAM SAMPLE;
           VAR X:T;
           TYPE T = . . . ;
           VAR Y:T;
           . . .
           BEGIN        {SAMPLE}
             . . .
           END;         {SAMPLE}
        . . .
```

Note:

Within the procedure SAMPLE, the variables X and Y may not be compatible since the type T has been redeclared between the two variable declarations involved.

8.2.3.1 Constant Declaration Part. A constant declaration introduces an identifier as a synonym for a constant. The value associated with the constant identifier may not be changed during system execution.

Constant declarations are of the form:

```
CONST <constant declaration list>
```

where <constant declaration list> is one or more of the following:

```
<identifier> = <constant> ;
```

where <constant> may be a signed real constant, string constant, character constant, an integer constant expression, or a previously defined constant identifier.

Example:

```
CONST MAX = 100;
      ASTERISK = '*';
      ONE_HALF = 0.5;
```

This is the only place where integer constant expressions can be used in place of an integer constant. Integer constant expressions are described in Section 8.3.5.5.

8.2.3.2 Type Declaration Part. A type declaration associates an identifier with a data type. Data types are discussed in Section 8.3. A data type determines the set of values a variable of that type may assume, and the set of basic operations that may be performed on them.

Type declarations are of the form:

```
TYPE <type declaration list>
```

where <type declaration list> is one or more of the following:

```
<identifier> = <type> ;
```

and <type> is defined in Section 8.3.

Example:

```
TYPE VECTOR = ARRAY [1..10] OF REAL;
     DAYS = (MON,TUES,WED,THURS,FRI,SAT,SUN);
     DIGITS = '0'..'9';
     COMPLEX =
       RECORD
```

```
            RE,IM:REAL;
          END;
```

8.2.3.3 Variable Declaration Part. A variable declaration defines a named data structure that can contain values of a single type. Variable declarations are not allowed at the system level.

Variable declarations are of the form:

```
        VAR <variable declaration list>
```

where <variable declaration list> is one or more of the following:

```
       <identifier list> : <type> ;
```

where <identifier list> is a list of identifiers separated by commas.

Example:

```
      VAR NYEARS:INTEGER;
          AMOUNT,VALUE,RATE:REAL;
          TEN_YEARS:VECTOR;
          PROFIT:ARRAY [1..10] OF BOOLEAN;
```

Note:

The type VECTOR was defined in te example in Section 8.2.2.

8.2.3.4 Common Declaration Part. A common declaration is used to declare variables which may be shared with other modules falling within the scope of the common declaration, or with externally compiled modules. Common variables are not allocated on the stack, therefore they exist during the entire life of the system. This makes it possible to "save" the value of a "local" variable from one activation of a module to the next. Since this location may be externally referenced all references to the same common identifier reference the same location.

Common declarations are of the form:

```
        COMMON <common declaration list>
```

where <common declaration list> is one or more of the following:

```
       <identifier list> : <type> ;
```

where <identifier list> is a list of identifiers separated by commas.

Example:

```
      COMMON ROOT1,ROOT2:REAL;
             INITIAL_VALUE,FINAL_VALUE:INTEGER;
```

Note: Common identifiers must be unique within the first six characters.

8.2.4 Access Declaration Part

An access declaration serves to identify all common variables which are to be referenced in the body containing the access declaration. Any common variable access which is not legalized by an access declaration will cause an error.

Access declarations are of the form:

```
          ACCESS <identifier list> ;
```

where <identifier list> is a list of identifiers separated by commas.

Example:

```
       ACCESS ROOT1,ROOT2;
```

The normal scope rules do not apply to access declarations, so that even if a module falls within the scope of an access declaration at a higher level, the common variable is not accessible within the module unless an explicit access declaration appears in that module. Each access declaration must fall within the scope of the common declaration of the identifier for which access is declared.

8.2.5 Program Declarations

A program declaration specifies an independent process which does not share any global variables, except possibly commons, with any other programs.

A program declaration is of the form:

```
          PROGRAM <identifier><program parameters> ;
            <program block> ;
```

where the optional <program parameters> are as follows:

```
          ( <program parameter> ; ... ; <program parameter> )
```

and where a <program parameter> is as follows:

```
          <identifier list> : <type identifier>
```

where <identifier list> is simply a list of identifiers separated by commas and type identifier is either a standard type identifier or a user defined type identifier. All program parameters are value parameters(section 8.2.9). The type of a program parameter must not be a pointer type because there is a distinct heap region for each program.

A <program block> is similar to a <system block> except it allows declarations of variables and processes within programs. Therefore a program block is as follows:

```
          <label declaration part>
          <data declarations>
          <access declaration part>
          <program routine declarations>
          <process body>
```

where <program routine declarations> may be any of the following:

```
          <process declarations>
          <procedure declarations>
          <function declarations>
```

and <process body> is described in Section 8.2.1.

Example:

```
       SYSTEM . . . ;

         PROGRAM EXAMPLE_PROGRAM(PARM1,PARM2:INTEGER; PARM3:ANYFILE);
           CONST . . .             " constant declarations
           TYPE . . .              " type declarations
           VAR . . .               " variable declarations
           PROCEDURE . . .         " procedure and function declarations
           PROCESS . . .           " process declarations
         BEGIN                     " program body
           {# concurrent characteristics}
         END;

       BEGIN                       " system body
       END;
```

8.2.6 Process Declarations

A process declaration specifies a subordinate process to either a program or another process. A process has access to variables declared globally to it.

A process declaration is of the form:

```
          PROCESS <identifier> <process parameters> ;
           <process block> ;
```

where <process parameters> are optional; they are the same as for programs except <process parameters> may be a pointer type also.

A <process block> is the same as a <program block> defined above. Note that processes may be declared within other processes.

Example:

```
       SYSTEM . . .
```

```
    PROGRAM . . .

      PROCESS EXAMPLE_PROCESS( . value parameters   );
         " local declarations including other processes
      BEGIN
        {# concurrent characteristics}
      END;

    BEGIN   " program body
    END;
  BEGIN   " system body
  END;
```

8.2.7 Procedure Declarations

A procedure declaration specifies a routine which may be invoked to perform an action and return after it has completed it.

A procedure declaration is of the form:

```
PROCEDURE <identifier> <procedure parameters> ;
  <procedure block> ;
```

where <procedure parameters> are optional, but if provided are as follows:

```
( <procedure parameter> ; ... ; <procedure parameter> )
```

where <procedure parameter> may be either:

```
<identifier list> : <type identifier>
```

or

```
VAR <identifier list> : <type identifier>
```

The first case indicates that the parameters are value parameters and the second case indicates that the parameters are reference parameters. These two types of parameters are discussed in Section 8.2.9.

A <procedure block> is similar to a <process block> except that processes may not be declared within procedures. Therefore a procedure block is as follows:

```
<label declaration part>
<data declarations>
<access declaration part>
<procedure and function declarations>
<compound statement>
```

Example:

```
      PROCEDURE EXAMPLE_PROCEDURE
```

```
    (VALUE_PARAMETER:INTEGER;
     VAR REFERENCE_PARAMETER:INTEGER);
    " local declarations
  BEGIN   " body of procedure
  END;
```

8.2.8 Function Declarations

A function declaration specifies a routine which may be invoked within an expression to return a single value.

A function declaration is of the form:

```
FUNCTION <identifier><function parameters> :
   <type identifier> ;
  <function block> ;
```

where <function parameters> are the same as <procedure parameters> and a <function block> is the same as a <procedure block>.

Notice that the type of the function result must be given. The type of a function must either be a simple type or a pointer type. Structured result types are not allowed. The statement section of a function should have at least one assignment statement which assigns a value to the function identifier or the function returns an undefined result.

Example:

```
  FUNCTION EXAMPLE_FUNCTION
    (VALUE_PARAMETER:INTEGER):INTEGER;
    " local declarations
  BEGIN   " body of function
    EXAMPLE_FUNCTION := VALUE_PARAMETER;
  END;
```

One source of errors which can be difficult to discover is a function which not only returns a value through the identifier which names the function, but also changes the value of non-local variables. An action which changes the value of a non-local variable is called a side-effect. The following rules should be followed to ensure that side effects do not occur:

1. The left hand side of an assignment statement should not be a non-local variable, a variable parameter, or a common variable.

2. Procedures should not be invoked from within functions.

8.2.9 Parameter Kinds

There are two kinds of parameter substitution:

(1) Value substitution - This is the normal or default situation. The actual parameter is evaluated and the resulting value is assigned to the corresponding formal parameter. This is refererred to as call by value, and prevents the called module from changing the value of the actual parameter in the calling module.

(2) Variable substitution - The address of the actual parameter is passed to the called module and this address is used to access the actual parameter indirectly. This form of parameter passing is known as call by reference. Note that in this case, within the body of the module an assignment to an identifier which is a formal parameter changes the actual parameter in the calling module. Variable parameter substitution is specified by preceding the formal parameter section with the reserved word VAR.

Value parameter transmission offers the security of preventing inadvertant changes to program values by a routine. It also may be an efficient way to pass simple variables as parameters. However, since this method involves copying values, it may be inefficient to pass structured data such as large arrays by value.

8.2.10 EXTERNAL Declarations

A program, process, procedure, or function may be declared to be externally defined. To do this the block is simply replaced by the identifier EXTERNAL. This is useful when programs or processes are separately compiled but need to be invoked by other modules. An EXTERNAL routine should not reference any global data except that which is passed to it in the <parameter list>.

The form of an external declaration is one of the following:

```
          PROGRAM<identifier><program parameters> ; EXTERNAL ;
or
          PROCESS <identifier><process parameters> ; EXTERNAL ;
or
          PROCEDURE <identifier><procedure parameters> ; EXTERNAL ;
or
          FUNCTION <identifier><function parameters> :
              <type identifier> ; EXTERNAL ;
```

Example:

```
        FUNCTION SQRT (X:REAL):REAL; EXTERNAL;
```

8.2.11 FORWARD Declarations

A program, process, procedure, or function may be forwardly declared. This is necessary when two or more modules call each other which is called indirect or mutual recursion.

The form of a forward declaration is one of the following:

```
            PROGRAM <identifier><program parameters> ; FORWARD ;
or
            PROCESS <identifier><process parameters> ; FORWARD ;
or
            PROCEDURE <identifier  <procedure parameters> ; FORWARD ;
or
            FUNCTION <identifier><function parameters> :
                <type identifier> ; FORWARD ;
```

When the module is subsequently declared, with its block, the parameters and the function result type must be omitted. The actual declaration of a module forwardly declared must have its block defined at the same level.

Example:

```
        FUNCTION F (X:REAL):REAL; FORWARD;

        PROCEDURE P (M:REAL);
             . . .
           BEGIN      {P}
              X := F(A)
           END;       {P}

        FUNCTION F;
             . . .
           BEGIN      {F}
              P(T)
           END;       {F}
```

8.2.12 Concurrent Characteristics

The <Concurrent characteristics> specify parameters that concern multiprogramming. They include the amount of memory needed by, and the priority of, a system,program or process. Memory requirements must be set if the user has more than one site of execution in his code. The form of the <concurrent characteristics>, as used in the <process body> (Section 8.2.1) is one of the following:

```
                {# <concurrent characteristic list>}
                                  or
                (*# <concurrent characteristic list> *)
```

where the <concurrent characteristic list> is one or more <concurrent characteristic>s separated by semicolons. The <concurrent characteristic> is of the form:

<concurrent keyword> = <concurrent value>

The possible <concurrent keyword>s and their associated meanings are given below.

Keyword	Meaning
HEAPSIZE	Number of words allocated for the system, program, or process heap
PRIORITY	Relative urgency of the system, program, or process
STACKSIZE	Size of the stack region, in words, to be used by the system, program, or process

The <concurrent value> may be a parameter of the program or process, an integer constant, or integer constant identifier. These compile-time specifications may only appear immediately following the initial BEGIN of a system, program, or process declaration. See Section 12 for complete details on how to choose appropriate values for STACKSIZE and HEAPSIZE.

NOTE: A program or process hs its stack and heap allocated out of its parents heap. If STACKSIZE is not set, it defaults to zero. If HEAPSIZE is not set and if the program or process needs any heap, it is then taken from its parents heap. In the host environment, a system with no specified heap takes all the heap allocated by the user in response to the prompts from the EXECUTE or DEBUG commands. In the target environment, the system with no specified heap will take all the heap allocated at power up initialization. A program or process cannot give heap to a process it hs spawned unless enough heap has been explicity allocated to it by a HEAPSIZE concurrent value.

Examples:

```
        {# PRIORITY = 2; HEAPSIZE = HEAPAMT; STACKSIZE = 256}
        (*# STACKSIZE = MAXSTACK; PRIORITY = INTERRUPT *)
```

> Note: The <concurrent characteristics> specified are ignored in a conventional Pascal program that is to be executed in a debug environment.

8.2.13 Conventional Pascal Program

A system may consist of a single conventional Pascal program if it is the only program or process in the system. If this is the case, no processes may be declared within the program.

The syntax for a conventional Pascal program is:

```
          PROGRAM <identifier> ;
            <program block> .
```

where the <program block> does not include process declarations.

Notice that in this form no program parameters are allowed, because an implicit invocation of this one program is done. Also when this form is used, no system header or system block is given.

8.3 DATA TYPES

A data type defines the set of values a variable of that type may assume and the set of basic operations that may be performed on them. Each variable is associated with one and only one type. The simple types consist of the standard types INTEGER, LONGINT, REAL, BOOLEAN, and CHAR; plus the user defined scalar or subrange types. Structured types are composed from other types by describing the types of their components and by indicating a structuring method. The structuring methods available consist of arrays, records, sets, pointers, semaphores, and files.

A type declaration introduces an identifier as the name of a data type. The form of a type declaration is described in Section 8.2.

8.3.1 Simple Types

Simple types consist of the standard types INTEGER, LONGINT, BOOLEAN, CHAR, or REAL, or the user defined scalar or subrange types. An enumeration type is any simple type except REAL. An enumeration type is characterized by the set of its distinct values, upon which a linear ordering is defined.

Enumeration types are used for counting purposes, for example to index into an array or to control the number of iterations of a FOR statement.

The basic operators for variables of enumeration type are assignment (see Section 8.6.1.1 for the assignment statement) and the relational operators described below:

<	less than
=	equal
>	greater than
<=	less than or equal
<>	not equal
>=	greater than or equal

The standard functions applying to enumeration types are:

SUCC(X)	the successor of X
PRED(X)	the predecessor of X
ORD(X)	the integer ordinal value of X (applies to all enumeration types except INTEGER and LONGINT)

8.3.1.1 INTEGER and LONGINT Types. A value of type INTEGER is an element of a finite set of whole numbers which range from -32768 to 32767 (signed 16-bit quantity). A value of type LONGINT ranges from -2147483648 to 2147483647 (signed 32-bit quantity). A non-suffixed integer constant is of type INTEGER if its value lies within the range given above, or LONGINT if its value lies outside the subrange defined by INTEGER but within the subrange defined by LONGINT. If an integer constant is suffixed with an L, it is of type LONGINT.

The basic operators defined for INTEGER and LONGINT operands are:

+	unary plus or add
-	negate or subtract
*	multiply
/	divide (produces REAL result)
DIV	divide (divide and truncate)
MOD	modulus -> A MOD X = A - ((A DIV X) * X))

The standard functions applying to arguments of type INTEGER (LONGINT) are:

Function	Value	Result Type
ABS(X)	Absolute value of X.	INTEGER (LONGINT)
SQR(X)	X Squared.	INTEGER (LONGINT)
CHR(X)	The character with the ordinal value of X.	CHAR
ODD(X)	TRUE if X is odd, FALSE otherwise.	BOOLEAN
FLOAT(X)	X converted to REAL value.	REAL
LINT(X)	The INTEGER X converted to a LONGINT value.	LONGINT
TRUNC(X)	The LONGINT X converted to an INTEGER value.	INTEGER

The standard function LOCATION may be used to obtain the address of a unpacked variable (Section 8.4). LOCATION may also be used to obtain the entry point of a routine. In both cases the result type returned by LOCATION is of type INTEGER.

The arithmetic relational operators apply to INTEGER or LONGINT operands and yield a Boolean result.

8.3.1.2 BOOLEAN Type. A value of type BOOLEAN is one of the logical truth values denoted by the reserved words TRUE and FALSE.

Operators defined for Boolean operands which yield Boolean values are:

NOT	logical negation
AND	logical conjunction
OR	logical disjunction

The constants TRUE and FALSE are predeclared keywords so that the ordinal value of FALSE is strictly less than TRUE (FALSE < TRUE). Thus, the relational operators apply to BOOLEAN operands and yield a Boolean result. Notice that each of the 16 Boolean operations can be defined using the Boolean operators listed above and the relational operators. For example, if P and Q are of type BOOLEAN,

P <= Q	expresses implication (P implies Q)
P = Q	expresses equivalence

P <> Q   expresses exclusive OR

Because of the precedence rules, expressions involving Boolean and relational operators may have to be parenthesized to obtain the desired result.

8.3.1.3 CHAR Type. A value of type CHAR is represented by its ASCII ordinal value. Character constants are written as a single character surrounded by apostrophes (single quotes). A character constant can be written by specifying its hex value, e.g. '#0D' is an ASCII carriage return. The following standard function applies to characters:

ORD(X)   The result (of type INTEGER) is the ordinal value of the character X.

8.3.1.4 Scalar Type. A scalar type is a user-defined type. The values of a scalar type are elements of a set of identifiers specified by the user. Each identifier defines a value of the type, and the order in which they are written defines the order of the type. The form of a scalar type declaration is:

(<scalar identifier list>)

where <scalar identifier list> is a list of identifiers separated by commas.

Examples:

```
        TYPE DAYS = (MON,TUES,WED,THURS,FRI,SAT,SUN);
             COLOR = (WHITE,RED,ORANGE,YELLOW,GREEN,BLUE,
                       PURPLE,BLACK);
```

The standard type BOOLEAN may be represented by the scalar type:

```
        TYPE BOOLEAN = (FALSE,TRUE);
```

which defines the standard identifiers FALSE and TRUE, and specifies that FALSE < TRUE.

The standard function ORD returns the ordinal number of a scalar value. The ordinal number of the first identifier in a scalar type is zero. Each identifier that appears as a value in a scalar type cannot be used for any other purpose within that scope (i.e. a scalar type definition does not open a new scope and the normal scope rules still apply).

8.3.1.5 Subrange Type. A type may be defined as a subrange of any previously defined enumeration type by specifying the least and largest values in the subrange. The form of a subrange type is:

<enumeration constant> .. <enumeration constant>

An <enumeration constant> is a constant value of some enumeration type. The first <enumeration constant> is the lower bound and the second <enumeration constant> is the upper bound. The lower bound must be less than or equal to the upper bound.

Examples:

```
       TYPE DIGITS = '0'..'9';
            WORKDAYS = MON..FRI;
            INDEX = 1..100;
```

Note: The type DAYS, as defined in section 8.3.1.4, is used as the enumeration type for the subrange WORKDAYS.

8.3.1.6 REAL Type. The type REAL may be used to represent REAL values with 6-7 decimal digits of precision. The range of absolute values that can be represented is approximately 1.0E-78 to 1.0E75.

The following operators accept operands of type REAL and yield a real value:

+	unary plus or add
-	negate or subtract
*	multiply
/	divide

The assignment operator may be used to assign a REAL value to a REAL variable. The relational operators are defined for REAL operands and yield a Boolean result.

The standard functions accepting a REAL argument and producing a REAL result are:

ABS(X)	absolute value of X
SQR(X)	X squared

Standard functions with a REAL argument yielding INTEGER results are:

TRUNC(X)	truncate the fractional part of X
ROUND(X)	round X to the nearest integer, i.e., TRUNC(X + 0.5) if X >= 0 or TRUNC(X - 0.5) if X < 0

Similar standard functions, LTRUNC and LROUND, yield a result of type LONGINT.

8.3.2 Structured Types

Structured types can be constructed from other types which are called components. The structuring methods available include the array, record, set, file, pointer, and semaphore.

8.3.2.1 Array Type. An array type consists of a group of components which are all of the same type. The form of an array type definition is:

```
        ARRAY  <index type list>  OF <component type>
```

The <component type> may be of any type except a file type. Note that the <component type> may itself be an array type. The <index type list> is a list of <index types> separated by commas. The number of <index types> in the declaration determines the dimension of the array. There is no limit to the number of dimensions an array may have. Each <index type> must be one of the enumeration types: BOOLEAN, CHAR, subrange, or scalar. Thus, the number of components in an array is static (fixed at compile time).

Examples:

```
       One Dimensional Arrays
            VAR VECTOR,ARRAYROW:ARRAY [1..10] OF REAL;
                SICKDAYS:ARRAY  DAYS  OF BOOLEAN;
```

Arrays with two or more dimensions are called multi-dimensional arrays. These may be defined in terms of one-dimensional arrays since the type

```
       ARRAY  [T1, T2]  OF BOOLEAN
```

is equivalent to

```
       ARRAY  [T1]  OF
         ARRAY  [T2]  OF BOOLEAN
```

Examples:

```
      Multi-dimensional arrays
            VAR TABLE:ARRAY [0..10 , 10..20] OF INTEGER;
                BOOK:ARRAY  [1..MAX , 1..80, 1..66] OF CHAR;
```

Components of an array are specified by their relative positions in the array, using expressions of the index type(s). Note that it takes n index expressions to specify a component of an n-dimensional array.

The only operator between array operands of compatible type is assignment (:=).

Examples:

```
       Using the array type definitions given above,
            SICKDAYS[FRI]  := TRUE;
            VECTOR[1]  := 3.1415;
            TABLE [1,20]  := 37;
            BOOK [5,1,25]  := 'Z';
       and
            ARRAYROW := VECTOR

            which is equivalent to
```

```
FOR I := 1 TO 10 DO
  ARRAYROW[I] := VECTOR I ;
```

A special array type called a string is defined to be a

```
PACKED ARRAY [T1] OF CHAR
```

where T1 must be of the form "1..<integer constant>". The <integer constant> specifies the length of the string. The number of characters in a string constant implicitly determines its type (the length of string). To assign a string constant to a variable of type string, the lengths of each must match exactly. A character constant is not considered to be the same as a string constant; therefore a single character may not be assigned to a variable of type string. The basic operators for variables of string type are assignment (:=) and the relational operators (<, =, >, <=, <>, >=).

Examples:

```
TYPE STRG = PACKED ARRAY [1..6] OF CHAR;
VAR WORD1,WORD2:STRG;

WORD1 := 'PASCAL';
WORD2 := 'RECORD';

IF WORD1 < WORD2
  THEN . . .
WHILE WORD2 <> WORD1 DO . . .
```

The standard procedures for arrays are:

```
PACK(A, I, Z)          means, for J := U to V do
                                 Z [J]  := A [J-U+I]
UNPACK(Z, A, I)        means, for J := U to V do
                                 A [J-U+I]  := Z [J]
```

where A is a variable of type ARRAY [M..N] OF T1,
Z is a variable of type PACKED ARRAY [U..V] OF T2,
T1 and T2 are compatible types, and
(N-M) >= (V-U).

UNPACK provides an efficient way to move all the elements of a packed array to an unpacked array, and PACK provides the converse function.

8.3.2.2 Record Type. A record type consists of a number of components of possibly different types called fields. Each field in a record type must have a distinct name. A field of a record can be of any type except a file type. The form of a record type definition is:

```
RECORD <field list> END
```

A <field list> can have a <fixed part> or a <variant part> or both. The <fixed part> is simply an arbitrary number of <record section>s separated by semicolons. A <record section> can be empty (contains nothing at all) or has the following form:

```
        <field identifier list> : <type>
```

A <field identifier list> is a list of field identifiers separated by commas.

Examples:

```
         TYPE COMPLEX =
                RECORD
                  RE,IM:REAL
                END;
              DATE =
                RECORD
                  MONTH:(JAN,FEB,MAR,APR,MAY,JUN,JUL,
                         AUG,SEP,OCT,NOV,DEC);
                  DAY:1..31;
                  YEAR:INTEGER
                END;
```

The only operator applying to records as structures is that of assignment (:=). The assignment operator applies to operands which are records of exactly the same type.

Examples:

Using the record type definitions given above,

```
              VAR INITIAL,FINAL:DATE;
                  C1,C2,C3:COMPLEX;

              INITIAL.DAY := 20;
              FINAL.YEAR := 1978;

              C1.RE := 3.4;
              C3.IM := 5.8;
```

and

```
              INITIAL := FINAL;
```

which is equivalent to

```
              INITIAL.MONTH := FINAL.MONTH;
              INITIAL.DAY := FINAL.DAY;
              INITIAL.YEAR := FINAL.YEAR;
```

Record Variants

The <variant part> of a record has the following form:

```
CASE <tagfield>:<tagtype> OF <variant list>
```

A <tagfield> is a field identifier of type <tagtype>. The <tagfield> (along with the ":" separator) is optional. However, the <tagtype> is required. The <tagtype> may be a standard type identifier: BOOLEAN, CHAR, INTEGER, or LONGINT; or a user defined type identifier. The <variant list> is an arbitrary number of <variant>s separated by semicolons. A <variant> can be empty (contains nothing at all) or has the following form:

```
<case label list>: ( <field list> )
```

A <case label list> is a list of <case label>s separated by commas. A <case label> is a either a constant value or subrange value. A <case label> must be compatible with the <tagtype> for the variant section.

A variant section is used to allow individual records to have some differences in their structure. Based on the value of one field (the "tagfield"), a particular set of "variant" fields for the record is selected. This permits a method for "overlaying" data since the record need only be as large as the largest variant part.

Example:

```
       TYPE CODE = (CREATE,CHANGE,DELETE);
            UPDATE =
              RECORD
                UPDATE_DATE:DATE;
                CASE ACTION:CODE OF
                  CREATE:(INITIAL_VAL:INTEGER);
                  CHANGE:(CHANGE_TYP:(ADD,SUB,REPLACE);
                  CHANGE_VAL:INTEGER);
                  DELETE:( );
                END;
```

Note that every field name in a record must be distinct, even those field names appearing in different variant parts of the record.

8.3.2.3 Set Type. A set type is used to define variables whose values are sets. A set type specifies a base type, and a value of the set is then any subset of values from the base type. The syntax is:

```
SET OF <base type>
```

where the <base type> is any enumeration type.

The lower bound of the set must have an ordinal value greater than or equal to zero; the maximum element must have an ordinal value less than or equal to 1023. Therefore, a set may have at most 1024 elements and the size of the set is determined by the value of the maximum element (a set is always based at zero).

Examples:

```
       TYPE CHARSET = SET OF CHAR;
            RANGESET = SET OF 0..7;
            COLOR:(RED,YELLOW,ORANGE,BLUE);
       VAR WEEK,WEEKEND:SET OF DAYS;
           SHADE1,SHADE2:SET OF COLOR;
           HUE:COLOR;
```

Note: The type DAYS was defined in Section 8.3.1.4.

Set values are written as a list of set elements, separated by commas, and enclosed in the set brackets [and] . A set element is an expression of the base type or a subrange of values of the base type. The empty set is denoted by [ ].

The basic operators for sets are:

+	set union
-	set difference
*	set intersection
<=	set inclusion (contained in)
>=	set inclusion (contains)
<	proper set inclusion
>	proper set inclusion
=	set equality
<>	set inequality
IN	set membership
:=	assignment

The operands must be compatible sets. In the case of the IN operator, the first operand is a set member, the second operand is a set. The result of a relation is a Boolean value. The result of the other operators is a set value that is compatible with the operands.

Examples:

```
       [4,5,6] + [5,4,3] = [3,4,5,6]
       [4,5,6] * [5,4,3] = [4,5]
       [2,3,4] - [5,4,3] = [2]
       ( [2,3,4] < [2,3,4] ) = FALSE
       ( [2,3,4] = [2,3,4] ) = TRUE
       ( [2,3,4] >= [2,3,4] ) = TRUE
```

Using the set type definitions given above,

```
            WEEKEND := [SAT,SUN];
            WEEK := WEEKEND + [MON..FRI];
            SHADE1 := [RED,YELLOW] * [YELLOW,ORANGE];
            SHADE2 := [YELLOW..BLUE];

            IF SHADE1 <> YELLOW  THEN . . .
            IF HUE IN SHADE2 THEN . . .
```

8.3.2.4 File Type. A file type is a structure which consists of a sequence of components which are all of the same type. The form of a file type is:

```
               FILE OF <component type>
                        or
            RANDOM FILE OF <component type>
                        or
                       TEXT
```

The <component type> of a file can be of any type except a pointer type or file type. It is recommended that the component type not contain pointers as a substructure, although the language makes no such restriction. The number of components, called the length of the file, is not fixed and may grow to any size, depending on the storage medium with which the file is associated.

The prefix RANDOM designates a random file in which components are accessible by their component number. This numbering is defined by default to be the natural ordering of the sequence of the components; the first component is number zero.

A special type of sequential file is a TEXT file which is a file of type CHAR and is divided into lines by end-of-line markers. INPUT and OUTPUT are standard predeclared TEXT files.

Examples:

```
       TYPE REC =
              RECORD
                NAME:PACKED ARRAY [1..15] OF CHAR;
                ID_NUM:INTEGER;
              END;

       VAR F:FILE OF INTEGER;
           EMPLOYEE:RANDOM FILE OF REC;
           TEMP:TEXT;
```

An additional predeclared file type is the type ANYFILE. The type ANYFILE may be used when passing file parameters to modules. A formal parameter of type ANYFILE matches an actual parameter of any file type, i.e. ANYFILE is a generic type. This method of parameter passing allows modules to manipulate those file characteristics which are independent of the file components and their types.

Example:

```
       PROGRAM EXAMPLE (FACTS:ANYFILE; . . .);
                 . . .
          BEGIN           { EXAMPLE}
                 . . .
          END;            {EXAMPLE}

       PROGRAM P (INPUT,OUTPUT;TEXT);
                 . . .
          VAR INFO:TEXT;
              DATA:FILE OF INTEGER;
                 . . .
          BEGIN          { P}
                    . . .
             START EXAMPLE(INFO,. . .);
                    . . .
             START EXAMPLE(DATA,. . .);
                    . . .
          END;            {P}
```

Standard procedures and functions are provided for file manipulation. Section 8.7 contains more information about the various file types and how they are managed.

8.3.2.5 Pointer Type. Variables may be referenced indirectly by means of a pointer, which may be thought of as the address of a variable. The form of a pointer type definition is:

```
       @<type identifier>
    or ↑<type identifier>
```

The <type identifier> is said to be bound to the pointer type. Pointer types may not point to file types. The <type identifier> need not be defined before the pointer type is defined, provided it is declared sometime later in the declaration Section. This is an example of a forward type declaration. Forward type declarations are permitted only with pointer types. Pointers may only point to variables of the type they are associated with. The standard predefined constant NIL is an element of every pointer type and points to no element at all.

Example:

```
       TYPE PTR = @LIST;
            LIST =
              RECORD
                VALUE:REAL;
                LOC:0..#FF;
                NEXT:PTR;
              END;
```

The operators applying to pointer operands with compatible types are:

```
:=    assignment
=     equal (the result is TRUE if the operands point
            to the same "address")
<>    not equal
```

Data can be dynamically allocated from a storage area called the heap using the standard procedure NEW. Data allocated from the heap remains allocated (independent of textual scope rules) until the storage area is dynamically deallocated by the standard procedure DISPOSE. These standard procedures are described in detail in Appendix C.

Typical uses of pointers are for constructing linked lists and binary tree data structures. A linked list of records can be created easily by defining a record which contains one field which is a pointer to the next record. Similarly, a binary tree of records can be constructed by defining a "right link" pointer and "left link" pointer for the record.

8.3.2.6 SEMAPHORE Type. The type SEMAPHORE is used in connection with low-level synchronization of processes. While the internal representation of a semaphore is transparent to the user, the underlying concept, used by the support routines, is that of counting semaphores. Operations on variables of type SEMAPHORE are performed by various functions and procedures which must be declared EXTERNAL by the programmer (section 8.2.10). Arithmetic operations are not valid for SEMAPHORE operands. A variable of type SEMAPHORE is considered to have the same space requirements as a variable of type pointer (Section 8.3.2.5).

Example:

```
VAR SEM1,SEM2:SEMAPHORE;
```

8.3.2.7 PACKED Types. The symbol PACKED may only prefix an array or record type definition -- this concept associated with the other structured types would be meaningless. If a structure is declared to be packed, several unstructured components of the structure, if possible, are stored in one word. Packing may economize the storage requirements of a data structure, but it may cause a loss in efficiency of access of its components.

An example of a packed type is a string type which is discussed in Section 8.3.2.1. A string type is a packed array of characters. This means that characters are packed together in the array structure (one character per byte).

A packed type is not compatible with an unpacked type that would otherwise be compatible. Passing an element of a packed structure as a VAR parameter (by reference) is not permitted.

If a type occurs in a packed structure, exactly as much storage as specified by the size algorithm will be allocated to it, subject to the following restrictions. Only enumeration types are packed. Every structured type starts on a new word boundary and consumes an integral number of words. The size algorithm, given below, specifies in terms of bits or words the internal representation for the value of a type. The size algorithm allocates either a portion of a word or an integral number of words. That is, if a type requires more than one word, it always uses an integral number of words, and not an integral number of words plus a fractiion of a word. Consequently, gaps of unused bits may occur. If a type does not occur in a packed structure, the size is a lower bound; the actual size is an integral number of words selected to facilitate efficient access to the type.

The size associated with each type is defined as follows:

CHAR: 1 word (8 bits in a packed structure)

INTEGER: 1 word (16 bits)

LONGINT: 2 words (32 bits)

BOOLEAN: 1 word (1 bit in a packed structure)

Scalar type: Let N be the ordinal of the largest member of the enumeration, and define NR(N) to be the least value I such that N < 2**I. Then the scalar type requires NR(N) bits.

Subrange type: Let L and U be the lower and upper bounds of the subrange. Then if L >= 0, the size is the same as for a scalar type which has the ordinal of the largest member of its enumeration equal to U. If L < 0, the size is Max(NR(-L-1),NR(ABS(U)) + 1.

REAL: 2 words (24 bit mantissa, 8 bit exponent)

Pointer type: 1 word (16 bit)

Array: If the array is not packed, each element occupies one or more consecutive words. Let S be the size of an element, that is, the size of the component type. If the array has E elements, the size of the array is E*S.

If the array is packed and the minimum size of an element is greater than a word, the space, S, allocated for each element is the minimum number of words which will contain it. If the array has E elements, the size of the array is E*S.

If the array is packed and the minimum size, S, of an element is less than a word, as many elements as possible are packed per word (D) with a possible number of bits left unused at the end of the word. The size of the array is (E-1) DIV D + 1 words.

Record: The size associated with a record type is the number of consecutive words needed to contain the fields in the fixed part plus the largest field list in the variant part. Fields are allocated in the order of the declaration.

If a record is not packed, a field occupies one or more words as required by the size of its associated type.

If the record is packed, a field is allocated within the remainder of the word allocated to the previous field provided it fits. If not, the field is left-justified at the beginning of the next available word and the previous field is right-justified in the previous word. If the size of the record is greater than a word, the last field of every variant is right-justified.

Field lists within the variant part are overlaid upon one another.

Set: The size of a set type depends on the size of its base type. If a base type of a set has an upper bound with ordinal value N, a set requires N DIV 16 + 1 words.

File: 1 word

The standard function SIZE applies to any type. SIZE(T) yields a result of type INTEGER which is the number of addressable units (bytes) required to represent the type T.

8.3.3 Type Compatibility

Two types are distinct if they are explicitly or implicitly declared in different parts of the program. A type is explicitly declared using a TYPE declaration. A type may be implicitly declared in a VAR declaration or in other places where a name is not associated with the type. Two types that are not distinct are identical.

Two types are compatible if one of the following is true:

1) they are identical types.

2) both are subranges of a single enumeration type.

3) both are string types with the same length.

4) both are pointer types which point to identical types.

5) both are set types with compatible base types.

6) both are file types with compatible element types.

Note: arrays and records are compatible only if they are identical.

There is no implicit conversion of types except from INTEGER and LONGINT to REAL and between INTEGER and LONGINT.

8.3.4 Overriding the Type Structure

The type of a data object is meant to be an invariant property of that object. Two ways exist to override the type structure of the language:

1) Use of type-transfer.

2) Using variants in record structures without checking tag fields.

When a type-transfer is used, the variable referenced is treated as if it were of the type given following the double-colon. A type transfer does not perform a value conversion; all that is altered is the apparent type of the variable. If the variable is a packed item, the type to which it is transferred must be representable within the boundaries of that item.

8.4 VARIABLES

Variables are used within the statement section of a module to reference the module's data structures. A variable may be either a simple identifier which references the entire variable, or a qualified variable which references a component of a structured variable.

8.4.1 Simple Variable

A simple variable is simply an identifier and it references the entire variable.

The form of a simple variable is as follows:

```
          <variable identifier>
```

8.4.2 Indexed Variable

An indexed variable is used to reference an element of an array.

The form of an indexed variable is as follows:

```
          <variable> [<expression> , ... , <expression>]
```

The expressions are used as subscripts for each of the n dimensions. If an array variable is declared to have n dimensions, the indexed variable may have from 1 to n subscript expressions. The types of the subscript expressions must correspond with the declared index types. There is also a compiler option, CKINDEX, which will check the value of a subscript to make sure it is within the declared bounds.

If a variable is declared:

```
          A : ARRAY  [1..10 , 1..20]  OF INTEGER;
then
          A  [2 , 4]
is equivalent to
          A  [2]  [4]
```

Examples:

```
       VECTOR[8]
       VECTOR [k+3]
       BOOK [1,5,10]
       SICKDAYS THURS
       TABLE [i,j]
       TABLE [10,15]
```

8.4.3 Record Variable

A record variable is used to reference a field within a record.

The form of a record variable is as follows:

```
          <variable> . <field identifier>
```

where <field identifier> is one of the fields declared within the record.

Examples:

```
       C1.RE
       C2.IM
       INITAL.DAY
       INITIAL.YEAR
       FINAL.MONTH
```

8.4.4 Pointer Variable

A pointer variable is used to reference the variable pointed to by a pointer type.

The form of a pointer variable is as follows:

```
          <variable> ^
       or <variable> @
```

The value of the pointer variable is undefined until either a value is assigned to it or until a NEW is performed on it to allocate an area of dynamic storage. The constant NIL can be assigned to any pointer variable which means it points to no variable at all. A compiler option, CKPTR, is available to check if any references are made through a NIL pointer.

Examples:

```
       TYPE PTYPE = @REC;
            REC =
              RECORD
                KEY:INTEGER;
                VALUE:REAL;
                LINK:PTYPE
              END;
       VAR P:PTYPE;
```

Using the type and variable declarations given above:

```
            NEW(P);
            P@.KEY := 38;
            IF P@.VALUE > 0
              THEN . . .
```

8.4.5 Type Transferred Variable

A type transferred variable is used to temporarily change the type of an existing variable.

The form of a type transfer variable is as follows:

```
          <variable> :: <type identifier>
```

Notice that the type is specified by a standard type identifier or by a user defined type identifier.

When a type transfer is applied to a variable no conversion is performed; only the apparent type of the variable is changed. Also the type transfer does not change the way the original variable was accessed and does not apply any further accessing unless followed by further variable qualification.

Example:

```
       TYPE BYTE = 0..#FF;
            RECTYPE =
              PACKED RECORD
                MSBYTE,LSBYTE:BYTE;
              END;

       VAR V:ARRAY[0..9] OF INTEGER;
           R:RECTYPE;
```

Valid type transfers:

```
       R.MSBYTE := V[0]::BYTE;
       V[1]::BYTE := R.LSBYTE;
```

There is one restriction to applying the type transfer. Namely, if the variable is a component of a packed structure, the size of the packed structure must be at least as large as the size of the type being transferred to. For instance, it is illegal to type transfer a packed byte (8 bit packed component) to the type INTEGER.

Example: The type transfer

```
       R.MSBYTE::INTEGER
```

is illegal since the size of the type INTEGER is larger than the eight bits required to represent the component R.MSBYTE.

The fundamental use of the type transfer is to overlay a type template on a data structure so that components of the structure may be treated as if they were of any desired type. This feature should be used with caution because there is no checking performed to detect when one may reference outside the original variable's area.

8.5 EXPRESSIONS

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expression consist of operands, operators, and function calls.

8.5.1 Operands

Operands are used to reference values of constants or variables.

An operand may be one of the following:

```
          <integer constant>
or
          <real constant>
or
          <string constant>
or
          <character constant>
or
          <constant identifier>
or
          NIL
or
          <set>
or
          <variable>
or
          <function call>
```

8.5.2 Operators

An operator specifies an operation to be preformed on one or two operands. An operator can only be applied to two operands if their data types are compatible. Some operators accept mixed mode operands. If either operand is of type REAL and the other is of type INTEGER or LONGINT, the latter is converted to REAL and hence the result is REAL. Also if either operand is of type LONGINT and the other is of type INTEGER, the INTEGER operand is converted to LONGINT and hence the result is LONGINT. Notice that the / operator always produces a REAL result but may accept INTEGER operands. Also notice the DIV and MOD only accept INTEGER or LONGINT operands and produce INTEGER or LONGINT results.

In order to evaluate an expression, it is necessary to know the meaning of each operator and its precedence, which specifies the order in which the operators are to be applied.

The operators are:

Group 1 : Multiplying operators:

*	multiplication; set intersection
/	real division
DIV	integer division (divide and truncate)
MOD	modulus, A MOD X = A - ((A DIV X) * X)

Group 2 : Adding operators:

+	addition; unary plus; set union
-	subtraction; unary minus; set difference

Group 3 : Relational operators:

=	equal
<>	not equal
<	less than; proper set inclusion
>	greater than; proper set inclusion
<=	less than or equal; set inclusion
>=	greater than or equal; set inclusion
IN	set membership

Logical operators:

Group 4 :	NOT	Negation
Group 5 :	AND	Conjunction
Group 6 :	OR	Disjunction

When used on strings, the relational operators denote alphabetical ordering according to the collating sequence of the underlying character set (ASCII).

The list of operators is in order of precedence, with groups of higher precedence listed first. In an expression, operators of highest precedence are evaluated first, and within each group, the operators have equal precedence and are evaluated from left to right within the expression. Parentheses may be used to explicitly determine the order of evaluation.

Examples:

Expression	Value
2 + 3 * 5	17
15 DIV 4 * 4	12
NOT (5 + 5 >= 20)	TRUE
6 + 6 DIV 3	8
3 < 5 OR 2 >= 6 AND 1 > 2	TRUE

In a Boolean expression of the form

X AND Y

if X is FALSE, Y is not evaluated and the value of the expression is FALSE. In a Boolean expression of the form

X OR Y

if X is TRUE, Y is not evaluated and the value of the expression is TRUE. This method of Boolean expression evaluation is called flow of control evaluation or short circuit evaluation.

8.5.3 Set Value

A set value may either be denoted by a set variable or a set constructor.

A set constructor has the following form:

<set element list>

where <set element list> is zero or more <set element>s separated by commas. If a set constructor has zero elements it denotes the empty set. Each <set element> may be one of the following:

<expression>

or

<expression> .. <expression>

The <expression> set element defines the value of the expression to be a member of the set. The second form of a set element specifies a subrange of values to be members of the set(the lower and upper values are included in the range).

All expressions within a set element list must be the same type. If the first form of a set element is used, the element in the set represented by the ordinal value of the expression is set. If the second form of a set element is used, each element between the first expression value and the second expression value are set including the two expression values. If the first expression value is greater than the second expression value, it is assumed to be an empty set element(specifies no members at all). Also no set element expression may be less than zero or greater than 1023.

Examples:

```
[RED,YELLOW]
[MON..THURS,SAT]
[ ]
```

8.5.4 Function Calls

A function call is used to invoke a function(with or without parameters) which computes a single value.

A function call has the following form:

```
<function identifier> <parameters>
```

where the optional <parameters> are as follows:

```
( <actual parameter> , ... , <actual parameter> )
```

where each <actual parameter> may be either a variable or an expression.

Each actual parameter must match the type of the formal parameter. Implicit type conversions are performed if the formal parameter is of type REAL and the actual parameter is either of type INTEGER or LONGINT , or if the formal parameter is of type LONGINT and the actual parameter of type INTEGER. Therefore, if a formal value parameter is of type REAL, an actual paramter of type INTEGER is converted to type REAL before it is passed.

8.5.5 Integer Constant Expressions

Integer constant expressions are nearly identical to regular expressions except all operands must be constants either of type INTEGER or LONGINT and only the following operators are valid:

+	unary plus or add
-	negate or subtract
*	multiply
DIV	divide
MOD	modulus

8.6 STATEMENTS

Statements describe the actions which a system performs on its data. Statements which contain other statements as components are structured statements; otherwise they are simple statements. Simple statements include: the assignment statement, procedure call statement, START statement, ESCAPE statement, GOTO statement, and ASSERT statement. Most structured statements are used to control the sequence in which statements are to be executed. They are used to form loops, branches, and transfers, which are known as the control structures of the language. The structured statements are: the compound statement, IF statement, CASE statement, FOR statement, WHILE statement, REPEAT statement, and WITH statement.

8.6.1 Simple Statements

Simple statements are statements that contain no other statements. These include the assignment statement, procedure call statement, START statement, ESCAPE statement, GOTO statement, and ASSERT statement.

8.6.1.1 Assignment Statement. An assignment statement specifies that an <expression> is to be evaluated and the resulting value is to be assigned to a <variable>. The form of an assignment statement is:

```
<variable> := <expression>
```

The type of <expression> must be compatible with that of the variable ; type compatibility is described in Section 8.3.3.3. The exception to this rule is when an implicit conversion is allowed. An expression of type INTEGER is implicitly converted to either LONGINT or REAL and an expression of type LONGINT is implicitly converted to either INTEGER or REAL. Direct assignments may be made to variables of any type, except files and semaphores.

Examples:

```
        X := A * NEXT DIV 2;
        FOUND := Z > TOTAL;
        COUNT := COUNT + 1;
        VALUE2 := SQR(VALUE);
        WORD := 'PASCAL';
```

Another use of the assignment statement is within the body of a FUNCTION where the computed value is assigned to the <identifier> which denotes the function. In this case the type of the <expression> must be compatible with the result type of the function (Section 8.2.8).

8.6.1.2 Procedure Statement. A procedure statement activates the specified procedure. The form of a procedure statement is one of the following:

```
<procedure name> ( <parameter list> )
                 or
           <procedure name>
```

where a <parameter list> is a list of the actual parameters, separated by commas, which are substituted for the formal parameters, given in the procedure heading (Section 8.2.7).

If a <parameter list> is specified, the actual parameters must match in number and type with the corresponding formal parameters that are declared in the procedure heading. If no <parameter list> is specified, the corresponding procedure must be declared to have no parameters. An empty parameter list (containing only a matched set of parentheses) in a procedure statement is equivalent to having no parameter list at all.

8.6.1.3 START Statement. A START statement is similar to a procedure statement except it invokes a program or process to execute concurrently in the system. The form of a START statement is one of the following:

```
START <identifier> ( <parameter list > )
                   or
           START <identifier>
```

where the <identifier> may be a <program name> or a <process name>. The <parameter list> is a list of actual parameters, separated by commas, which are substituted for the formal parameters, given in the program or process declaration (Sections 8.2.5 and 8.2.6).

If a <parameter list> is specified, the actual parameters must match in number and type with the corresponding formal parameters that are indicated in the program or process declaration. If no <parameter list> is specified, the corresponding program or process must be declared to have no parameters.

8.6.1.4 ESCAPE Statement. The ESCAPE statement is a structured jump statement. It is used to terminate execution of a structured statement, procedure, function, process, or program. The form of an ESCAPE statement is:

```
ESCAPE <identifier>
```

where the <identifier> may be an escape label, a procedure name, a function name, a process name, or a program name.

An escape label, followed by a colon, may prefix any structured statement. Each escape label is implicitly declared by its appearance in the program and may be referenced only within the structured statement it precedes.

An ESCAPE statement may only be used within the statement labeled by the corresponding escape label or within the scope of the procedure, function, process, or program mentioned. An ESCAPE from a structured statement causes processing to be continued at the statement immediately following the labeled statement. When an ESCAPE is executed from a module, control returns from the most recent activation of that module.

Any structured statement prefixed by an escape label may contain any number of ESCAPE statements which reference that label. The escape label and all ESCAPE statements that reference the label must be contained in the same module; it is not valid to escape across module boundaries. Also within any module, ESCAPE statements may not reference any other module declared at the same level; but may refer to its direct ancestors. An ESCAPE <process name> or an ESCAPE <program name> is only legal if the <process name or <program name> specified is that of the innermost process or program.

Example:
```
       LOOP:    WHILE I <= N DO
                  BEGIN
                    IF EOF
                      THEN ESCAPE LOOP;
                    READ(VAL);
                    SUM := SUM + VAL;
                    I := I +1;
                  END;
```

If an escape label and a statement label are used on the same structured statement, the statement label must appear before the escape label.

8.6.1.5 GOTO Statement. The GOTO statement transfers control to the statement having the specified label. The form of a GOTO statement is:

```
                GOTO <label>
```

where the <label> must be an unsigned integer value. The <label> must be explicitly declared in the LABEL declaration. A statement label is an unsigned integer which can be prefixed to any statement within its declared scope. If the <label> is not declared or does not appear as a statement label in the system, a syntax error occurs.

Example:
```
       PROGRAM SAMPLE;
       LABEL 2;
```

```
    . . .
    BEGIN
      . . .
    2:  I := I + 1;
        IF VECTOR[I] < 100
          THEN GOTO 2;
      . . .
    END.
```

It is not legal to jump into or out of a module, and it is not recommended to jump into a FOR or WITH statement. Labeled statements within FOR and WITH statements are flagged as possible locations of errors; this does not effect the execution of the system, provided that a jump does not occur into the FOR or WITH statement. However, if control is passed from outside the FOR or WITH statement to the labeled statement, unpredictable results will occur.

If a statement label and an escape label are both necessary on the same structured statement, the statement label must appear first.

GOTO statements are seldom necessary to use. The use of other Pascal control structures has been shown to be beneficial in terms of basic software engineering principles.

8.6.1.6 ASSERT Statement. The ASSERT statement allows the programmer to specify, using a BOOLEAN expression, a condition that should be true at a given point in the system. The form of an ASSERT statement is:

```
ASSERT <expression>
```

where expression must be of the type BOOLEAN.

If the ASSERTS compiler option is turned on, the <expression> in the ASSERT statement is evaluated when encountered; if it is TRUE, execution continues; otherwise a run-time error occurs.

Examples:

```
      ASSERT X <> 100;
      ASSERT FOUND;
      ASSERT LIMIT <= MAX;
```

The ASSERT statement is useful in system testing since it can be included anywhere in the system where a certain condition or relation should evaluate to true during the execution of the system.

8.6.2 Structured Statements

Structured statements contain other statements and are used to control the sequence of execution of these statements. Structured statements are used to form the language control structures such as loops, branches, and transfers. Structured statements include the compound statement, the conditional statements IF and CASE, the repetitive statements FOR, REPEAT, and the WITH statement.

8.6.2.1 Compound Statement. A compound statement is a sequence of statements enclosed by the keywords BEGIN and END which are treated as a single statement. The form of a compound statement is:

```
BEGIN <statement list> END
```

where the <statement list> is a list of zero or more statements, simple or structured, separated by semicolons. BEGIN/END act as statement delimiters in a compound statement.

The sequence of statements that make up the <statement list> are executed one by one in the order in which they appear, but the entire sequence is treated as a single statement.

Example:

```
BEGIN
  EXCHANGE := X1;
  X1 := X2;
  X2 := EXCHANGE
END;
```

The semicolon is used to separate statements and is not part of any individual statement. Therefore a semicolon need not follow the last statement in the statement list . If one does occur, it is assumed that a statement exists between the semicolon and the symbol END. This is an example of an empty statement which specifies that no action is to be taken. Most empty statements do not alter the flow of statement control. However, the user must be aware of empty statements because in some context the flow of control is altered.

Example:

```
BEGIN
  SUM := X + Y + SUM;
  X := X + 5;
  Y := Y - 2;
  WRITELN(SUM);
  {<empty>}
END;
```

8.6.2.2 IF Statement. The IF statement specifies that a <statement> is to be executed only if a given condition is TRUE; otherwise an alternative <statement> is executed, if present. The form of an IF statement is one of the following:

```
IF <expression> THEN <statement>
```

or

```
IF <expression> THEN <statement> ELSE <statement>
```

where the <expression> must be of type BOOLEAN.

If the <expression> evaluates to TRUE, the first <statement> alternative, the THEN clause, is executed; otherwise the second <statement> alternative, the ELSE clause, is executed if it is present.

Examples:

```
        IF COUNT >= 0 AND COUNT <= LENGTH
          THEN READ(X[I]);

        IF X < Y
          THEN MAX := Y
          ELSE MAX := X;
```

An ambiguity arises regarding multiple ELSE clauses in nested IF statements. The dangling ELSE problem is resolved by always associating an ELSE with the most recent unmatched THEN preceding it; any other desired interpretation requires either restructuring the IF statement, or adding a BEGIN/END to create a compound statement that could be used as the statement in a THEN or an ELSE clause.

Example:

```
        IF A > B
          THEN
            IF B > C
              THEN MIN := C
              ELSE MIN := B;
```

is equivalent to:

```
        IF A > B
          THEN
            BEGIN
              IF B > C
                THEN MIN := C
                ELSE MIN := B
            END;
```

Misplaced semicolons can cause syntax errors in an IF statement. For example, a syntax error always occurs whenever a semicolon immediately precedes the symbol ELSE. This would create two separate statements: an IF statement, followed by an unknown statement which begins with the keyword ELSE. The existence of the empty statement may cause some misplaced semicolons to remain undetected by the compiler since the resulting constructs may be syntactically correct. However, they can cause logical errors in the system which are not immediately apparent.

8.6.2.3 CASE Statement. A CASE statement allows a statement to be selected for execution depending on the evaluation of an <expression> at run-time. The form of a CASE statement is:

```
            CASE <expression> OF
              <case label list> : <statement>;
                      . . .
              <case label list> : <statement>
              OTHERWISE <statement list>
            END
```

where the <expression> must be of an enumeration type, the <case label list> is a list of one or more <case label>s separated by commas, and the <statement list> is a list of zero or more Pascal statements, simple or structured, separated by semicolons. The <case label list> : <statement> combination may be repeated zero or more times within the CASE statement, each occurence must be separated from the previous one by a semicolon. The OTHERWISE clause is optional.

The <case label> is either a constant value or a subrange value of the same enumeration type as the <expression> to be used as the selector. The <case label> is not a <statement label>, i.e. a <case label> may not be referenced by a GOTO statement. All <case label>s within a single CASE statement must be unique. The range limit of <case label>s within any CASE statement is 256.

The value of the <expression> at run-time is used as the selector for the CASE statement. If the <case label> indicated by the selector is present in the CASE statement, the corresponding component statement is executed. If the <case label> is not present and an OTHERWISE clause is included, the <statement list> following the OTHERWISE is executed. If the selected <case label> is not present and there is no OTHERWISE clause, a run-time error occurs.

Examples:

```
       CASE NUM OF
         0..3,8: TOTAL := TOTAL + NUM;
         4,6,7: TOTAL := TOTAL - NUM;
         5,9: TOTAL := TOTAL DIV 2
       END;

       CASE ALFA OF
         'A'..'M': CH := SUCC(ALFA);
         'N'..'Z': CH := PREC(ALFA)
         OTHERWISE WRITELN('NOT IN ALPHABET');
            INT := ORD(ALFA)
       END;
```

8.6.2.4 FOR Statement. A FOR statement allows for the repeated execution of a given statement for an increasing or decreasing progression of values which are assigned to the control variable of the FOR statement. A FOR statement is useful if the required number of repetitions is known beforehand. The form of a FOR statement is one of the following:

```
FOR <identifier> := <initial value> TO
    <final value> DO <statement>
                    or
FOR <identifier> := <initial value> DOWNTO
    <final value> DO <statement>
```

where the <identifier> is the control variable, and the <intial value> and <final value> must both be of an enumeration type -- a set type is not an enumeration type.

The control variable is implicitly declared by its appearance in the FOR statement and has scope only within the FOR statement. The value of the control variable may not be changed within the FOR statement either by assignment or by passing it as a reference parameter to a routine.

The control variable is assigned the <initial value> prior to the first execution of the <statement>. If the <intial value> is greater (less) than the <final value> in the TO (DOWNTO) case the <statement> is never executed. Otherwise, after each execution of the <statement> the control variable is incremented (decremented) by one until the value of the control variable is greater (less) than the <final value> in the case of TO (DOWNTO). Both the <initial value> and the <final value> are only evaluated once, prior to the first execution of the <statement>, so the total number of repetitions to be made is determined before the execution of the FOR statement.

Examples:

```
FOR I := N DOWNTO 1 DO
  SUM := SUM + A[I];

FOR DAY := MON TO FRI DO
  BEGIN
    READ(HRS,RATE);
    PAY := RATE * HRS;
  END;
```

8.6.2.5 WHILE Statement. A WHILE statement allows for the repeated execution of a given statement as long as the specified condition is true; the total number of repetitions is greater than or equal to zero. The form of a WHILE statement is:

```
WHILE <expression> DO <statement>
```

where the <expression> must be of type BOOLEAN.

The <expression> is evaluated before the execution of the <statement>. If the <expression> is initially false the <statement> is not executed at all; otherwise, the <statement> is executed repeatedly as long as the <expression> evaluates to true. The <expression> is evaluated before each execution of the <statement>.

Example:

```
        I := 1;
        WHILE I <= MAX DO
          BEGIN
            VALUE := AMT [I] + TAX [I+2] ;
            I := I + 1
          END;
```

8.6.2.6 REPEAT Statement. A REPEAT statement allows a sequence of statements to be repeatedly executed as long as the specified condition is false; the total number of repetitions is greater than or equal to one. The form of a REPEAT statement is:

REPEAT <statement list> UNTIL <expression>

where the <expression> must be of type BOOLEAN and <statement list> is a list of zero or more statements, simple or structured, separated by semicolons. REPEAT/UNTIL act as statement delimiters similar to BEGIN/END in a compound statement.

The <statement list> is executed once before the initial evaluation of the <expression>. If the <expression> initially evaluates to true, the <statement list> is executed only once; otherwise the <statement list> is repeatedly executed as long as the <expression> evaluates to false. The <expression> is evaluated after each execution of the <statement list>.

Example:

```
        I := 1;
        REPEAT
          IF A [I] > A [I+1]
            THEN
              BEGIN
                TEMP := A [I] ;
                A [I] := A [I+1] ;
                A [I+1] := TEMP
              END;
          I := I + 1
        UNTIL I >= LENGTH;
```

8.6.2.7 WITH Statement. A WITH statement can be one of two distinct forms or a combination of both of them. This statement is used to simplify references to components of record variables (Section 8.3.3.2.2). The first form of a WITH statement is:

WITH <record variable> DO <statement>

This form allows all components of the specified <record variable> to be denoted by field identifiers within the scope of the WITH statement. Nested WITH statements of this form are also useful and may be written using a shorthand notation by replacing the <record variable> with a list of <record variable>s separated by commas.

Example:

```
       WITH INITIAL DO                {VAR INITIAL:DATE}
         BEGIN
           MONTH := SEP;
           DAY := 9;
           YEAR := 1978
         END;

       is equivalent to

       INITIAL.MONTH := SEP;
       INITIAL.DAY := 9;
       INITIAL.YEAR := 1978;
```

A more relible form of the WITH statement is provided with which an <identifier> representing a synonym for the <record variable> may be defined. This form of a WITH statement is:

```
        WITH <identifier> = <record variable> DO <statement>
```

where the <identifier> used as the synonym is implicitly declared within the WITH statement and is only accessible within the scope of the WITH statement.

This form of the WITH statement may also be expanded to create synonyms for more than a single <record variable> by giving a list of synonym assignments, separated by commas.

Example:

```
       WITH I = INITIAL, F = FINAL DO      {VAR INITIAL,FINAL:DATE}
         BEGIN
           I.MONTH := AUG;
           F.MONTH := MAY;
           I.DAY := 28;
           F.DAY := 20;
           I.YEAR := 1975;
           F.YEAR := 1978
         END;

       is equivalent to

       INITIAL.MONTH := AUG;
       INITIAL.DAY := 28;
       INITIAL.YEAR := 1975;

       FINAL.MONTH := MAY;
       FINAL.DAY := 20;
       FINAL.YEAR := 1978;
```

This is more reliable since variables in the scope of the WITH statement must be spelled in their entirety. For example, in case a local variable has the same name as a field of a record, using this form of the WITH statment there is no confusion as to which one is meant.

8.7 INPUT AND OUTPUT

Files are manipulated via standard procedures or functions which are described in this section. Files are accessed by means of READ procedure statements, and are written to by means of WRITE procedure statements. In addition, READLN and WRITELN statements apply to textfiles.

8.7.1 Sequential File Operations

Before values may be read from a file, it is necessary to execute a RESET statement which positions to the beginning of the file and opens it for reading. READ returns the next value from the file. For a sequential file, reading may proceed until the last component is read. Then the sequential file is in the end-of-file state which is indicated by the function EOF. When applying READ to a file F with a variable V of a type compatible with the component type of F, the value of the next component of the file F is assigned to the variable V. V must not be an element of a packed structure. If F is positioned at the end-of-file mark, nothing is read, and an error exception occurs.

The general form of READ is as follows:

```
READ (<file variable>, <variable>,..., <variable>)
```

where each <variable> must be compatible with the component type of the <file variable> and each <variable> is read from successive components of the file.

Before writing to a file, it must be opened for writing by the procedure REWRITE. When applying WRITE to a file F with an variable V of a type compatible with the component type of F, the value of V becomes the next component of the file F.

The general form of WRITE is as follows:

```
WRITE (<file variable>, <variable>,..., <variable>)
```

where each <variable> must be compatible with the component type of the <file variable> and each <variable> is written into successive components of the file.

Sequential files may be opened for reading or writing but not both simultaneously.

Example:

```
       PROGRAM COPYSEQ;
         TYPE REC =
                RECORD
                  NAME:PACKED ARRAY [1..10] OF CHAR;
                  ID_NUM:INTEGER;
                END;
         VAR EMPLOYEE:REC;
             OLDCOPY,NEWCOPY:FILE OF REC;
       BEGIN
         RESET(OLDCOPY);
         REWRITE(NEWCOPY);

         WHILE NOT EOF(OLDCOPY) DO
           BEGIN
             READ(OLDCOPY,EMPLOYEE);
             WRITE(NEWCOPY,EMPLOYEE);
           END;
       END.
```

8.7.2 Text File Operations

Input and output data for many devices such as card punches, card readers, line printers, and CRT terminals is in the form of characters. The physical properties of these devices naturally divide files of characters into lines. A file of characters which is divided logically into lines by end-of-line markers is called a textfile.

Both input and output to and from text files are two-stage operations.

Writing to a file proceeds line by line. The WRITE statement writes the file component values into a line buffer. NOTE: WRITE does not write directly to the file. A WRITELN statement causes any values specified to be written to the line buffer, followed by output of the entire line buffer to the file. The line buffer is ONLY output to the file when the buffer becomes full (80 characters) or when a WRITELN is encountered.

Similarly, reading from a file is a two-stage operation. The READ statement takes a component from the line buffer and places it in a variable. NOTE: READ takes a component from the line buffer, puts it in a variable and then fetches a new line buffer from the file.

A RESET command must be sued prior to the first READ or READLN statement to open a file for reading. NOTE: The RESET does an automatic READLN. If this is not noted, it is possible for input to become shifted by one line from that which is expected.

When the last nonblank character of a line is read, the standard function EOLN when applied to the textfile returns the value TRUE. At this point, a READLN operation changes the value to FALSE. For more details concerning the textfile oeprations see Section 10.2.6.

Two standard textfiles are predefined in conventional Microprocessor Pascal System programs: INPUT and OUTPUT. If they are used in the Microprocessor Pascal system, they may be passed as parameters to a program via the parameter list; but they must be declared if they are to be used within that program.

Example:

```
PROGRAM COPYTEXT;
  VAR CH:CHAR;
      ORIGINAL,COPY:TEXT;
BEGIN
  RESET(ORIGINAL);
  REWRITE(COPY);

  WHILE NOT EOF(ORIGINAL) DO
    BEGIN
      WHILE NOT EOLN(ORIGINAL) DO
        BEGIN
          READ(ORIGINAL,CH);
          WRITE(COPY,CH);
        END;
      WRITELN(COPY);
      READLN(ORIGINAL);
    END;
END.
```

Note: This example works well when both the ORIGINAL and the COPY files are associated with a VDT device because one line is written before another line is read.

8.7.2.1 Text File Read Operation. A READ procedure statement has the form:

```
READ (<file variable>, <variable>,..., <variable>)
```

where <file variable> must be a text file and <variable> must be of type INTEGER, LONGINT, BOOLEAN, CHAR, REAL, or string (PACKED ARRAY 1..n OF CHAR). The variable must not be an element of a packed structure.

The file INPUT is assumed to be the default <file variable> if one is not explicitly given in the READ statement.

Examples:

```
READ(<variable>, <variable>)

is equivalent to

READ(INPUT, <variable>, <variable>)
```

similarly

```
READLN
```

is equivalent to

```
READLN(INPUT)
```

and

```
READLN(<variable list>)
```

is equivalent to

```
READ(<variable list>);
READLN
```

For each of the types mentioned above, the following action is performed for a READ operation:

If V is of type CHAR, the value read is the next character in the textfile (blanks are significant characters).

If V is of type BOOLEAN, the character T or F is read, or the standard identifier TRUE or FALSE is read.

If V is of type string with length L, the next L characters are read from the text file (blanks are significant characters).

If V is of type INTEGER, LONGINT, or REAL, the next series of digits which correspond to the definition of a constant of that type are read from the text file. This includes hexadecimal constants for INTEGER and LONGINT types.

Values to be read may not cross line boundaries. For the types INTEGER, LONGINT, REAL, and BOOLEAN all leading blanks are skipped; entirely blank lines are also ignored.

Note:

Although formatted read parameters are not allowed in READ statements, it is possible to perform formatted input using DECODE. Read parameters passed to DECODE may be of the form

```
V:M
```

where M is the field width.

8.7.2.2 Text File Write Operation. A WRITE procedure statement has the form:

```
WRITE (<file variable>, <parameter>,..., <parameter>)
```

where <file variable> must be a text file and <parameter> may have one of the following forms:

```
  <expression>
or
  <expression> : <field width expression>
or
  <expression> : <field width expression> : <decimal digits>
```

where <expression> represents the value to be written, <field width expression> specifies the minimum field width into which the value is to be written, and <decimal digits> specifies the number of digits to be output after the decimal point for REAL values.

The file OUTPUT is assumed to be the default <file variable> if one is not explicitly given in the WRITE statement.

Examples:

```
       WRITELN

       is equivalent to

       WRITELN(OUTPUT)

  similarly

       WRITE(<parameter>)

       is equivalent to

       WRITE(OUTPUT, <parameter>)

  and

       WRITELN(<parameter list>)

       is equivalent to

       WRITE(<parameter list>);
       WRITELN
```

The value to be written may be of type INTEGER, LONGINT, CHAR, BOOLEAN, REAL, or string. The value written is never split across two lines so WRITELNs may be performed implicitly. If the specified field width is greater than the line length an error occurs. If the value is less than one, at least one space is used. When no field width is given, a default field width is supplied by the compiler.

The following table gives the default values.

Type	Default Field Width
INTEGER	10
LONGINT	15
BOOLEAN	5
CHAR	1
REAL	15
String	Length of string

The specified field width is the minimum field width. If the value requires less than the specified width, an adequate number of preceding blanks are written such that the specified number of characters are written. If the value requires more than the specified width, the necessary additional space is allocated and used.

For each of the types mentioned above, the following action is performed for a WRITE operation:

If the value is of type CHAR, the character value is written with possible leading blanks.

If the value is of type BOOLEAN, the standard identifier TRUE or FALSE is written, preceded by an appropriate number of blanks. If the specified field width is less than five, either the character T or F will be written.

If the value is of type string, the entire string is written with possible leading blanks. If the specified field width is less than the length of the string, the entire string is written.

If the value is of type INTEGER or LONGINT, the value is written as a string of decimal digits. If the WRITE <expression> is followed by the identifer HEX, the value is written as a string of hexidecimal digits.

If the value is of type REAL, the value is written as a string of characters either in fixed point notation or in floating point notation with a coefficient and scale factor. Fixed point notation is only used if the <decimal digits> expression is specified.

specifies which element in the file is to be accessed. The EOF function returns a value of TRUE if a nonexistent file element is referenced. Note that if record N is written, the records 0 through N exist, even though values may not have been written to all of these records. A major difference between a RANDOM file and a sequential or TEXT file is that only a single parameter is allowed in both the READ and the WRITE statements.

8.7.3 RANDOM File Operations

The general form of READ for a RANDOM file is:

```
   READ (<file variable>, <record number>, <variable>)
```

where the <file variable> must be a RANDOM file, the <record number> must be an integer expression, and the <variable> must be compatible with the components of the specified file. The <record number> specified is treated as a LONGINT value.

The value of the <record number> is used to determine which record of the file is to be accessed. Therefore an error occurs if the resulting value is less than zero or if the referenced component does not exist. This integer value is not automatically incremented after each READ statement is executed.

The general form for a RANDOM file WRITE statement is:

```
   WRITE (<file variable>, <record number>, <variable>)
```

where the <file variable> must be a RANDOM file, the <record number> must be an integer expression, and the <variable> must be compatible with the component type of the file.

The value obtained from the <record number> represents the position in the file which is to be accessed. If this value is less than zero, an error occurs. The execution of a WRITE statement does not automatically increment the value of the specifed <integer expression>.

8.7.4 Binding of File Names

The default name associated with a file is the first eight characters of the file variable. This name can be changed by calling the routine SETNAME specifing the new file name.

Example:

```
      SETNAME(OUTPUT,'PRINTER');
```

8.7.5 Passing Files as Parameters

File variables may be passed to modules by reference and by value. When a procedure or function requires a file as a parameter, the parameter must be passed by reference. If a program or process requires a file to be passed as a parameter it must be passed by value.

When a program or process requires a file as a parameter, it must be passed by value. If a file is passed by value, the called routine operates on its local file variable which is initialized to have the same characteristics as the one passed.

The standard identifiers INPUT and OUTPUT may be declared as parameters of programs since READ and WRITE operations use those file variables as defaults. If INPUT and OUTPUT are to be used within a program of the system, they must be declared.

The function FILENAMED may be used when invoking a program or process with a file parameter. In this case only the name of the file is passed.

Example:

```
        START COPY(FILENAMED('CARDREADER'),FILENAMED('PRINTER'));
```

8.7.6 Encode and Decode

The procedures ENCODE and DECODE function similar to the textfile procedures WRITE and READ respectively, except a memory array is used instead of a file. This allows binary values to be converted to ASCII strings and also parts of strings to be converted to binary values. These procedures also allow a primitive form of string manipulation whereby substrings may be extracted and appended to other strings.

The form of the ENCODE and DECODE procedures are as follows:

```
   ENCODE (<string>, <index>, <status>, <parameters>)
and
   DECODE (<string>, <index>, <status>, <parameters>)
```

where <string> is the variable of type string into which values are encoded or from which values are decoded. <Index> is the initial index into the string where values are to be encoded or decoded. This index may either be a integer constant or a variable which is incremented by the number of characters encoded or decoded. <Status> must be a variable which is given a value indicating the status of the encode or decode operations. The possible error codes returned by <status>, and their associated meanings are listed below:

ERROR CODE	MEANING

1	Bad parameter passed to I/O routine
2	Field width too large for logical record
3	Incomplete data (READ operation)
4	Invalid character in field (READ only)
5	Data value too large (READ only)
6	Attempt to READ past end-of-file
7	Field larger than logical record size.

<Parameters> for ENCODE may be of the form discussed in Section 8.7.2.2 for the WRITE procedure statement, and <parameters> for DECODE are similar except that <variable>s must be used instead of <expression>s.

Examples:

```
VAR STRG8:PACKED ARRAY [1..8] OF CHAR;
    STRG4:PACKED ARRAY [1..4] OF CHAR;
    CH:CHAR;
    STAT,N,NUM:INTEGER;
```

Using the variable definitions given above,

```
ENCODE(STRG8,4,STAT,SRTG4:4);
ENCODE(STRG4,N,STAT,CH:4-N);    {1 <= N <= 4}

DECODE(STRG8,N,STAT,NUM:2);     {1 <= N <= 8}
DECODE(STRG4,3,STAT,CH:1);
```

SECTION IX

PROCESS SYNCHRONIZATION AND PROCESSOR MANAGEMENT

The Executive Run Time Support (RTS) supports multiprogramming which allows more than one process to be executing in the same system concurrently. These concurrent processes must share CPU time, through a scheduling policy; and may have to be synchronized through the use of semaphores. This Section describes the routines that are available to the Executive RTS user for management of theses activities.

9.1 SCHEDULING POLICY

If all multiprogrammed processes executing in a system are viewed over a time frame of several seconds, all appear to be progressing at a non-zero rate. However, if the state of a single processor is followed for several milliseconds, the switching of the processor among many processes is apparent. The switching is done according to the scheduling policy of the Executive RTS which determines the assignment of a processor to one of several processes. Repetitive switching of processes causes interleaving of the processes' progression through their algorithms and gives the illusion that the processes execute concurrently. In a system with one sequential program, successive instructions in the program's algorithm are executed in order by the single process. In contrast to this, the multiprogramming of several concurrent processes causes the following phenomenon. Instructions that are successive in code space are not necessarily executed by the processor in succession without the unexpected intervention of other instructions elsewhere in code space. The unexpected intervention of other instructions causing the CPU to branch, occurs when servicing hardware interrupts or when time-slicing. An interrupt may occur at any time and cause the currently active process to be delayed while another process services the interrupt. In many systems interrupts from a clock or timer are used to periodically change the assignment of the processor so no single process executes longer than its "time slice". Over several seconds this forced interleaving of execution causes all active processes to appear to progress. In either of these two cases, the state of an active process (a process to which a processor is asssigned) is saved and restored later.

The current Executive RTS schedules one processor among many processes. Future versions will support multiprocessing or more than one processor. However the number of physical processors is irrelevant to the design of a multitasking system and only affects its performance (throughput or response to real-time events). Once more than one process is made active in an Executive RTS system (even on one processor), the scheduling of concurrent processes introduces a new dimension to the design of a software system. Even on a single physical processor, the interleaving of processes is supported, so the concurrent activity of processes should be considered in the design of any process.

Each process is a separate entity with respect to execution on the host processor (or processors). The scheduling policy of the Executive RTS determines which of several concurrent processes in a system will be in execution at any instant; selection is based on process readiness and process priority.

Each process in a system is in one of two states: it is either ready for execution (but not necessarily executing) or suspended (typically waiting for some change in the state of the system). A suspended process is also called blocked. An active process is assigned to a processor and is executing its instructions; an active process is considered to be in the ready state.

The relative priorities of processes determine which of several ready processes become active. Each process has a priority that is represented by a user-assigned, non-negative integer. Zero indicates the greatest urgency, 32766 indicates the least urgency, and the range zero to fifteen is reserved for device processes that are usually associated with interrupt handlers. The priority of a process will often be described in terms of "urgency" to avoid potential ambiguity between the numeric value of the scheduling parameter "priority" and the relative priority of one process to another; confusion can arise since numerically small values of priority (e.g., interrupt handlers) correspond to great urgency. The fundamental characteristic of the Executive RTS scheduling policy is that an active (executing) process is never less urgent than any of the ready processes.

A scheduling decision is made by the Executive RTS whenever a blocked process becomes ready or an active process becomes blocked (goes into a suspended state). In the future, the Executive RTS will support more than one processor and will assign each processor to an active process. Currently one processor is managed, and there exists only one active process assigned to that processor. The following scheduling algorithm is used for one processor:

1) All processes that are ready for execution reside in an ordered structure called a scheduling queue. The ordering of the queue is based upon priority: a more urgent process (one with lower arithmetic value of priority) precedes those with less urgency. The first process in the scheduling queue is the active (executing) process.

2) The Executive RTS creates a process called the idle process with the least possible urgency, priority 32767; this priority is reserved for use by the Executive RTS. The idle process is always ready and is the last member of the scheduling queue. If this process ever becomes active, it places the

processor in an idle state (executes the IDLE instuction) in which it remains until an interrupt occurs.

3) If the active process is suspended, the next process on the scheduling queue becomes the active process. Since the scheduling queue is ordered by priority, the most urgent process that is ready is given the processor.

4) If a process is changed from the suspended state to the ready state, it is inserted into the scheduling queue based on its priority. If the process is a device process, i.e., has priority in the range zero to fifteen, it is inserted in front of processes with the same priority; if it is not a device process, it is inserted after processes with the same priority. (The reason for this distinction between device and non-device processes is discussed in Section 9.5.)

5) If the process which just became ready is inserted in front of the active process, then the processor is preempted, and the new process becomes active. Otherwise, the previously active process is as urgent as all other ready processes, and it remains active.

This algorithm is presented so the user can predict the behavior of his system when debugging. Priority should be viewed as a scheduling parameter of the Executive RTS that indicates the relative urgency associated with the execution of each process. It is strongly recommended that the system designer not design algorithms in such a way that their correct behavior depends implicitly upon the relative priorities of processes. It is a good practice to design a multiprogramming system in such a way that it works correctly regardless of the relative rates at which individual processes progress. With such a design, each process can be generally understood in terms of its sequential code; global interactions are explicitly indicated by points of interprocess synchronization. Implicit use of relative process priorities leads to designs that are tightly coupled to a particular executive; such a system will probably not transport to a time-slicing or multi-processor version of the executive.

Figure 9-1 illustrates the application of the Executive RTS scheduling algorithm to a series of process activations and suspensions. The scheduling queue is depicted by a series of boxes containing the name and priority of the associated process. The box labeled IDLE corresponds to the idle process that is described above. The left column of the figure indicates which process is activated or suspended to cause a change in the scheduling queue. The first transition occurs when process C with priority 16 is made ready. Since C is not a device process, it is placed in the scheduling queue behind process A which also has priority 16. When process A is suspended, C becomes the active process; similarly process B becomes active when C is suspended. When process D is made ready, it becomes the active process (preempts B) because it has greater urgency than process B. Since process E is a device process, it preempts the active process even though it has the same priority. The remaining transitions show that the processor becomes idle if processes E, D, and B are suspended.

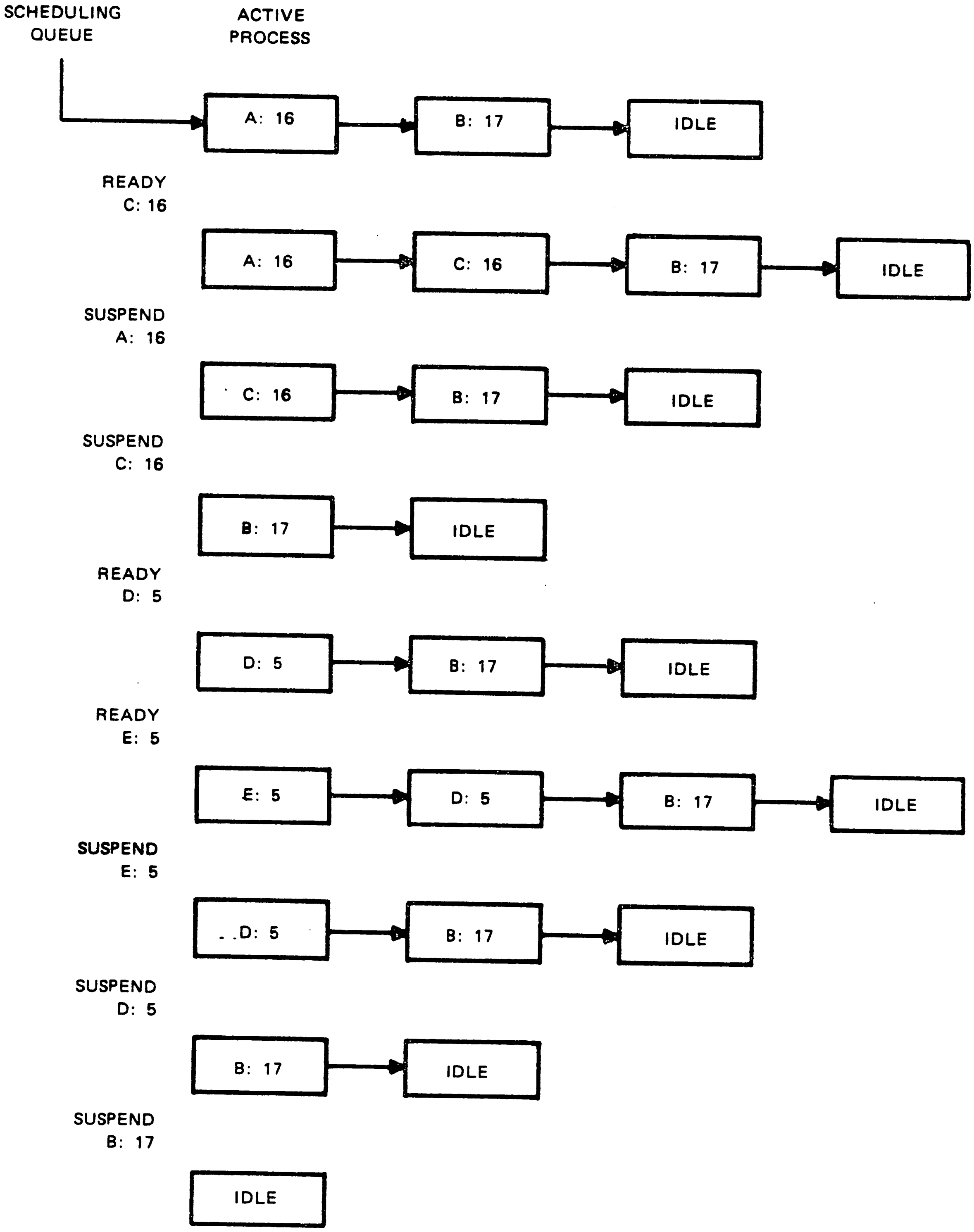


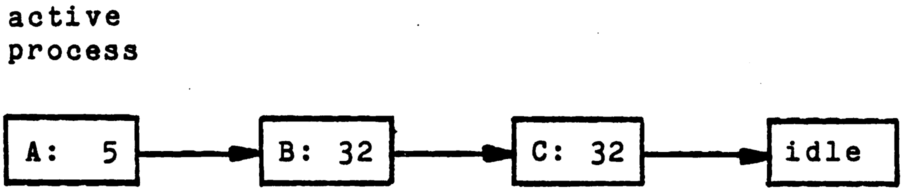
FIGURE 9-1. EXAMPLES OF THE EXECUTIVE RTS SCHEDULING POLICY

Note in the previous example there is no guarantee that a particular ready process ever becomes active. A process with equal or greater urgency could always be in front of the particular process in the scheduling queue. The Executive RTS scheduling policy interrupts the execution of a process temporarily to allocate its processor to a more urgent process and continue the interrupted process later when the more urgent one blocks or terminates. This is called preemptive scheduling with resumption and is designed for event driven systems that are composed of a large number of processes that execute in bursts until they block while waiting for a resource. Preemption makes it possible to achieve guaranteed response to service urgent processes.

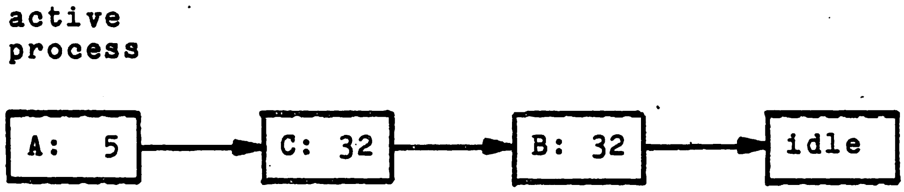
It is possible to alternate the processor among a collection of computation-bound processes by use of the RTS utility

```
     procedure swap; external;
```

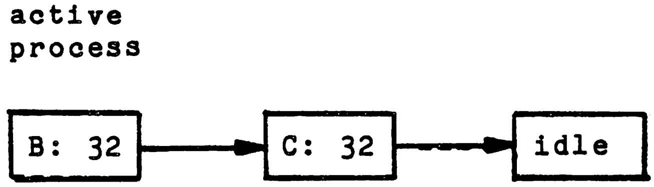
which has the effect of removing the first non-device process from the scheduling queue and then inserting it behind the last process with the same priority. (A non-device process has a priority between 16 and 32766 inclusive.) For example, if the scheduling queue looks like



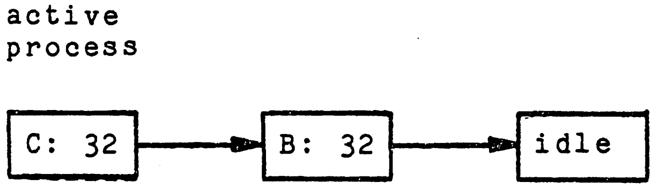
and SWAP is called from process A, then process B is moved behind process C:



Had process B been active and called SWAP, then process C would have become active, and B would become its successor in the scheduling queue:



becomes



A typical usage of SWAP is to implement time-slicing: a device process that is associated with a clock interrupt can force a non-device process to relinquish the processor by calling SWAP periodically. If the first non-device process is more urgent than all other ready processes, then SWAP does nothing and leaves the most urgent non-device process in its location in the scheduling queue.

The following RTS utility routine

```
type non_device_priority = 16..32766;

procedure setpriority(var oldvalue: non_device_priority;
  newvalue: non_device_priority); external;
```

changes the priority of the first non-device process in the scheduling queue; the position of that process in the queue of ready processes may have to be modified in order to remain consistent with the Executive RTS scheduling policy. Notice that SETPRIORITY cannot be used to modify the priority of a device process or to change a non-device process into a device process.

9.2 EVENTS

Real-time programming is concerned with the controlling of events in the physical world that do not wait for the computer. The central problem in a real-time environment is that the computer must be able to receive data and react to them as fast as they arrive; otherwise, the computer falls behind the real world events. The stimuli of events generated externally to the computer are passed to processes within the computer by means of hardware interrupts.

Event mechanisms are also useful in controlling the actions of concurrent processes within a computer. An event may be awaited by a process such that when the event happens, the process may service it, or proceed, or do some other action. Until the event happens, however, the waiting process is considered by the Executive RTS to be suspended and does not compete for the processor. The event upon which a process waits may be generated by another process. For example, consider processes that have related responsibilities; such processes often must communicate among one another, either through shared memory or some form of message passing protocol. Successful communication from one process to another requires some synchronization between the processes to ensure that they do not interfere with one another. For example, if one process wants to place an item into the next available space in a buffer, it must ensure that the receiving process does not modify pointers to the buffer until the transfer is complete. In general a system designer cannot determine a priori the rates at which individual processes progress. Semaphores provide a low-level primitive structure for the synchronization of otherwise asynchronous processes.

Synchronization of the action of a process with other actions is achieved by means of an event, which causes actions to concur or agree in time. The Executive RTS provides event mechanisms for external events through interrupt handling and for internal events through semaphores.

9.3 SEMAPHORES

9.3.1 Introduction

The fundamental tool in the Executive RTS for the synchronization of processes is the semaphore. (The term semaphore was apparently suggested originally by the semaphores of railroading that are used to synchronize access to sections of track.) A semaphore can be envisaged as representing some event on which processes wish to synchronize. A process may ensure that an event has occurred before proceeding by performing a WAIT operation on the associated semaphore. If the event has already occurred, the process continues execution; otherwise, the process is suspended until the event occurs. A process may signal the occurrence of an event by performing a SIGNAL operation on the associated semaphore. If some process is waiting for the event, then that process is made ready for execution; otherwise, the occurrence of the event is recorded in the semaphore until a subsequent WAIT operation occurs for that event. In either case the caller of SIGNAL remains in the ready state. The semaphores of the Executive RTS are counting (general) semaphore in the sense that an occurrence of an event is not lost even if no process is waiting when an event occurs -- a count is kept in a semaphore of the number of events that have occurred (by SIGNAL) but have not been received (by WAIT).

9.3.2 Abstract Operations on Semaphores

A semaphore may be viewed as having two components: a counter indicating the number of unserviced occurrences of the associated event and a (possibly empty) queue of processes waiting for the event to occur. The operations WAIT and SIGNAL have the abstract algorithms:

```
procedure wait(event: semaphore);
{Ensure that an occurrence of EVENT
 has happened before proceeding.}
begin
  {start indivisible operations}
  event.count := event.count - 1;
  if event.count   0
    then {suspend the calling process on EVENT.QUEUE};
  {end indivisible operations}
end {wait};

procedure signal(event: semaphore);
{Signal the occurrence of EVENT}
begin
  {start indivisible operations}
   event.count := event.count + 1;
   if event.count  = 0
    then {make the first process on
           EVENT.QUEUE ready for execution}
  {end indivisible operations}
end {signal};
```

These operations must be indivisible in the sense that they have exclusive access to the semaphore EVENT. If two processes had simultaneous access to the same semaphore, then they might modify EVENT.COUNT or EVENT.QUEUE in such a way that an occurrence of the associated event would be lost. Note that WAIT and SIGNAL may cause a process to be taken out of or inserted into the scheduling queue. When a signaled process is made ready, the Executive RTS scheduling algorithm determines whether the active process (executing in a processor) is the SIGNALer or SIGNALee.

Since a semaphore acts as an event counter, it must be initialized to the proper count before it is ever used. Otherwise the value of an uninitialized semaphore is interpreted to be an erroneous count value and/or queue value. The INITSEMAPHORE operation initializes a semaphore with an event count and has the following abstract algorithm:

```
type nonneg = 0..32767;

procedure initsemaphore(var event: semaphore;
  count: nonneg);
{Initialize EVENT with an event count.}
begin
  event.queue := {empty queue, i.e. no waiters};
  event.count := count;
end {initsemaphore};
```

Notice that a semaphore may be initialized with a count of zero. The maximum event count which a semaphore may hold by successive SIGNAL operations without corresponding WAIT operations is 32767. If the event count overflows, the process which causes the overflow by a SIGNAL fails.

INITSEMAPHORE may not be able to acquire the RTS-maintained data structure for a semaphore if memory is full. If the heap$term flag of the process calling INITSEMAPHORE is TRUE, then the process fails, else the routine returns and the semaphore is not initialized. Any operation on an uninitialized semaphore causes the calling process to fail. One can check that a semaphore is valid and initialized by calling

```
function oksemaphore(sema: semaphore): boolean;
  external;
```

which returns TRUE for a valid SEMA.

NOTE

> Every semaphore in a system must be initialized at least once, by calling INITSEMAPHORE, before the semaphore is used. If this is not done, operations on an uninitialized semaphore cause a fatal exception for the process operating on the uninitialized semaphore.

Processes that await the same event are delayed in the same semaphore queue. They are managed by the Executive RTS in a first-in, first-out queuing strategy (FIFO) which favors the longest delayed process. That is, the next time the awaited event occurs by a SIGNAL operation, the process which had been awaiting the event for the longest time is activated. So the Executive RTS orders a semaphore queue by the sequence in which processes are delayed in the queue.

As is discussed in detail in Section 9.3.5, each semaphore is implemented by a data structure that is allocated within a data area that is managed by the Executive RTS. For that data structure to be reclaimed by the Executive RTS,

```
procedure termsemaphore(var event: semaphore); external;
```

must be called by the last process that uses the associated semaphore.

9.3.3 Usage of Semaphores

A semaphore typically represents an event that is used to synchronize execution of several processes. The occurrence of an event is indicated by a call to the procedure SIGNAL with the associated semaphore as a parameter. Care must be taken when designing a collection of processes to ensure that the process that signals the event cannot do anything to nullify the event until receipt has been acknowledged. For example, consider two processes, one of which is placing into a cell in memory a value to be processed by the other. A semaphore DATA_AVAILABLE could be used to synchronize this transaction:

```
system test;

  type
    nonneg = 0..32767;

  procedure initsemaphore(var sema: semaphore;
    count: nonneg); external;
  procedure signal(sema: semaphore); external;
  procedure wait(sema: semaphore); external;

  program synchronize;
  var
    data: integer;
    data_available: semaphore;

  process producer;
  begin
    while true do begin
      {produce DATA}
      signal(data_available);
      end .loop forever ;
  end {producer};

  process consumer;
  begin
    while true do begin
      wait(data available);
      {consume DATA}
      end {loop forever};
  end {consumer};

  begin
    initsemaphore(data_available, 0);
    start producer;
    start consumer;
  end {synchronize};

begin
  start synchronize;
end {test}.
```

FIGURE 9-2. EXAMPLE OF INCORRECT SEMAPHORE USAGE

In general this is not a correct solution to the problem since there is no guarantee that CONSUMER has finished processing DATA when PRODUCER gets ready to place another value in DATA. Such a guarantee could be made if CONSUMER had a greater urgency than PRODUCER and would process DATA without having to relinquish the processor to wait for some other event. As was discussed in Section 9.1, it is unwise to structure a system so the scheduling policy is an implicit requirement for the success of an algorithm. The general solution requires a semaphore with which the receipt of DATA is acknowledged:

```
system test;

  type
    nonneg = 0..32767;

  procedure initsemaphore(var sema: semaphore;
    count: nonneg); external;
  procedure signal(sema: semaphore); external;
  procedure wait(sema: semaphore); external;

  program synchronize;
  var
    data: integer;
    data_available: semaphore;
    data_received: semaphore;

  process producer;
  begin
    while true do begin
      wait(data_received);
      {produce DATA}
      signal(data_available);
      end {loop forever};
  end {producer};

  process consumer;
  begin
    while true do begin
      wait(data_available);
      {consume DATA}
      signal(data_received);
      end {loop forever};
  end {consumer};

  begin
    initsemaphore(data_available, 0);
    initsemaphore(1ata_received, 1);
    start producer;
    start consumer;
  end { synchronize} ;

begin
  start synchronize;
end { test} .
```

FIGURE 9-3. EXAMPLE OF CORRECT SEMAPHORE USAGE

The semaphore DATA_RECEIVED is initialized to "1" so the first activation of PRODUCER can proceed from the WAIT operation at the beginning of its loop.

A potential problem that can occur with semaphores is deadlock, the state that occurs if two or more processes are waiting for conditions that can never hold because of a circular dependency. Each process is waiting on a condition that cannot occur because some other process is not active to cause it. As an example, suppose processes A and B both wish to perform a computation that requires access to resources X and Y. If process A executes

```
     wait( x ); wait( y ); . compute   signal( x ); signal( y )
```

and process B executes

```
    wait( y ); wait( x ); . compute   signal( y ); signal( x ) ,
```

then it is possible that each process passes the first call to WAIT and becomes deadlocked at the second. To avoid deadlock, each process must request the resources in the same order and must release them in the reverse of the order in which they were acquired:

```
   wait( x ); wait( y ); . compute   signal( y ); signal( x ) .
```

9.3.4 RTS Semaphore Routines

Microprocessor Pascal has a standard (predefined) type called SEMAPHORE. A variable of type SEMAPHORE may only be passed as a parameter to routines which implement semaphore operations; arithmetic operations are not permitted on SEMAPHOREs.

Semaphore routines are supplied by the Executive RTS and are declared by the user as EXTERNAL routines. The following types are assumed to be declared by the user:

```
        type nonneg = 0..32767;
          semaphorestate = ( awaited, zero, signaled );
```

RTS semaphore operations follow.

```
        procedure signal( sema: semaphore ); external;
```

This procedure performs the SIGNAL operation on semaphore SEMA: if a process is waiting for a signal to be sent through SEMA, then activate the first member of the waiting queue for SEMA; otherwise, increment the semaphore count of signals sent but not received. If the event count of SEMA overflows, then the caller of SIGNAL fails.

```
procedure wait( sema: semaphore ); external;
```

This procedure performs the WAIT operation on semaphore SEMA: if a signal has been sent to the semaphore SEMA but not received, then decrement the event count of SEMA and return; otherwise, suspend the calling process in the event queue of SEMA.

```
procedure initsemaphore( var sema: semaphore;
  count: nonneg ); external;
```

This procedure initializes the semaphore SEMA to have no waiters for the SEMA event and to have an event count equal to COUNT. The SEMA semaphore does not receive interrupt events.

```
procedure termsemaphore( var sema: semaphore );
  external;
```

This procedure notifies the Executive RTS that the SEMA semaphore is no longer in use. If the SEMA semaphore has a queue of waiting processes at the time this routine is called, an error is assumed, the calling process fails, and the semaphore is not terminated.

```
function semavalue( sema: semaphore ): integer;
```

This function returns the initial count of the SEMA semaphore plus the cumulative total of the number of SIGNAL operations minus the cumulative total of the number of WAIT operations on SEMA. Notice that a positive value of SEMAVALUE(SEMA) is the number of occurrences of the SEMA event that have happened but have not been serviced at the time of the call to SEMAVALUE. In other words, a positive value is the excess of SIGNAL operations over WAIT operations. A negative value of SEMAVALUE(SEMA) is the number of processes waiting for the SEMA event to occur. That is, a negative value is the excess of WAIT operations over SIGNAL operations. A zero value of SEMAVALUE(SEMA) indicates there is no process waiting for the SEMA event, and no SEMA events have occurred which have not been serviced. The result returned by this function must be used with care: it accurately reflects the state of SEMA at the time SEMAVALUE was called, but there is no reason to assume that the state does not change immediately thereafter. SEMAVALUE can be safely used if it is known that the system is in an invariant state (e.g., interrupts are masked) or if there is additional knowledge about SEMA (e.g., the caller of SEMAVALUE is the only process to wait on SEMA).

```
function semastate( sema: semaphore ): semaphorestate;
  external;
```

This function returns the state of SEMA semaphore. The AWAITED state indicates SEMA has processes waiting for an event which has not occurred. The SIGNALED state means that the event has occurred but not been serviced. And the ZERO state holds if the number of SIGNAL and WAIT operations are equal. Note that AWAITED, ZERO, and SIGNALED are equivalent to SEMAVALUE returning negative, zero, and positive values, respectively. The same caution must be taken in interpreting the result of SEMASTATE as was described above for SEMAVALUE. An application of SEMASTATE is to activate all processes waiting on a semaphore:

```
while semastate( sema ) = awaited do
  signal( sema );
```

For this code to truly empty the queue of SEMA and not send unwanted signals, the test on the state of SEMA and the following SIGNAL must be performed as an indivisible operation (e.g., interrupts should be masked).

```
procedure waitsignal( waitfor, signalthe: semaphore );
  external;
```

Two operations are performed by this routine in one indivisible step: a WAIT operation on the WAITFOR semaphore and a SIGNAL of the SIGNALTHE semaphore. This routine is useful to receive an event (by WAIT) and to issue an event (by SIGNAL) in one indivisible step without intervening instructions. This routine is not equivalent to

```
signal( signalthe ); wait( waitfor );
```

since the signal to SIGNALTHE could activate a process with a great enough urgency that the active process would be preempted before WAIT could be called.

```
procedure csignal( sema: semaphore;
  var waiter: boolean ); external;
```

This procedure peforms a SIGNAL operation only if a waiter on the SEMA event exists (the value of SEMASTATE(SEMA) is AWAITED). If there is at least one waiter for the SEMA event, then the first member of the waiting queue is activated and WAITER is returned TRUE. Otherwise, no SIGNAL operation is done, and WAITER is returned FALSE. The test and signal are performed in one indivisible operation. One application of this routine is to activate all the processes that are waiting on an event:

```
        repeat
          csignal( sema, waiter )
        until not waiter;
```

CSIGNAL is not equivalent to

```
        if semastate( sema ) = awaited then signal( sema )
```

since this statement is not indivisible: an interrupt could occur after SEMASTATE determines a waiting process exists, and the interrupt handler could activate that process before SIGNAL could be called in the THEN clause above.

```
        procedure cwait( sema: semaphore;
          var received: boolean ); external;
```

This procedure performs a WAIT operation only if an unserviced event has been recorded by SEMA but has not been previously received (the value of SEMASTATE(SEMA) is SIGNALED). That is, the caller of CWAIT ensures that a SEMA event has already happened, or the CWAIT call does nothing. RECEIVED is returned TRUE if a signaled event was received and FALSE otherwise. The test and wait are peformed in one indivisible operation. This routine is useful for a process to receive events which have already happened without possibility of the calling process being suspended. A usage of CWAIT is in an interrupt-handling process for which it is necessary to accept a signal if it has already been sent but it is not possible to wait for the signal to occur since the handler must remain in a state in which interrupts can be accepted.

The CSIGNAL and CWAIT operations allow one to implement polling for the occurrence of events rather than suspension and activation. Polling is repetitive testing for a condition or until a change is noted.

9.3.5 Implementation of Semaphores

When a variable of type SEMAPHORE is declared in Microprocessor Pascal System item that is allocated in the user's stack frame is not the structure that implements a semaphore but a reference (e.g., pointer) to that structure. Procedure INITSEMAPHORE must be called to allocate the semaphore structure (in an area managed by the Executive RTS) and initialize the reference to it; TERMSEMAPHORE must be called to deallocate the semaphore. This treatment permits the Executive RTS to manage all semaphores in a system and to ensure that semaphores can be addressed by all processes, even in the presence of memory mapping.

With this explanation of the implementation of semaphores, it should be clear why each of the semaphore utility routines (except INITSEMAPHORE and TERMSEMAPHORE) has as a parameter a semaphore passed by value. What is being passed is a reference to the semaphore, not a copy of the structure that implements it. Similarly, a semaphore can be declared in one process and passed (by value) to another, and the first process can then terminate. What is allocated in the first process is a reference to the semaphore structure that is in the data space managed by the Executive RTS. If the first process terminates (without calling TERMSEMAPHORE), the actual semaphore still exists.

9.4 INTERRUPT HANDLING

9.4.1 Introduction

An hardware interrupt is a stimulus from the external environment of a processor to pass an event to a process executing within the processor. The technique for interrupt handling uses the event mechanism of semaphores. A correspondence is established between an interrupt and a semaphore; when the interrupt occurs, a SIGNAL operation is performed on the semaphore, thus activating a device process to respond to the interrupt. A device process waits for the event associated with a semaphore; that event can be caused directly or indirectly by the occurrence of an interrupt. Indirect activation can be used to demultiplex an interrupt level. One process receives the interrupt and determines the nature of the interrupt and the event to which it corresponds; the associated semaphore is then signaled to activate a device process to handle the interrupt.

9.4.2 Interrupts Treated as Events

An interrupt may be viewed as an event which is externally generated; a semaphore event may be viewed as an internally generated event. A semaphore may be designated to the Executive RTS to be the event mechanism by which an external interrupt activates a process which awaits an interrupt of a fixed urgency level.

The event technique of handling interrupts is used as follows. A process which is to service all interrupts at level N executes a WAIT(SEMA) and is activated from the semaphore WAIT operation when an interrupt occurs. The priority of the process must be at least as urgent as the level N of the interrupt. The SEMA semaphore must previously have been initialized by calling the INITSEMAPHORE routine and must have been designated to the Executive RTS to receive all interrupt signals by calling EXTERNALEVENT(SEMA,N). This latter routine "attaches" the SEMA semaphore to the level N interrupt. After the process has serviced the interrupt, it again executes a WAIT(SEMA) operation and is suspended until the next level N interrupt. Figure 9-4 below illustrates a clock device service program which is passed a semaphore that has been attached to an interrupt level.

```
system example;
 const level_5 = 5;
 type interrupt_level = 0..15;
  nonneg = 0..32767;

 procedure initsemaphore( var sema: semaphore;
  count: nonneg ); external;
 procedure wait( sema: semaphore ); external;
 procedure externalevent( sema: semaphore;
  level: interrupt_level ); external;

 program clockdevice( clockinterrupt: semaphore;
  level: interrupt_level );
  { service interrupts from a clock device}
  begin
  {# priority=level; stacksize=200; heapsize=0}
   while true do { forever} begin
    { enable interrupts from clock device} ;
    wait( clockinterrupt );
    { disable interrupts from clock device} ;
    {       .
            .   service clock interrupt
            .

    end { forever loop} ;
  end { clockdevice} ;

 procedure initclockdevice( level: interrupt_level );
  { create clock device service program}
  var clockinterrupt: semaphore;
  begin
   initsemaphore( clockinterrupt, 0 );
    { first WAIT on CLOCKINTERRUPT causes suspension}
   externalevent( clockinterrupt, level_5 );
   start clockdevice( clockinterrupt, level_5 );
  end { initclockdevice} ;

 begin . example
 {# priority=1; stacksize=200; heapsize=0
  initclockdevice( level_5 );
 end { example} .
```

FIGURE 9-4. EXAMPLE OF SERVICING INTERRUPTS AS EVENTS

The Executive RTS actually permits the association of an alternate event with an interrupt by means of procedure ALTEXTERNALEVENT, which is described below. Such an event is intended to be used to handle unexpected or spurious interrupts. If an interrupt occurs and the (primary) external event semaphore associated with the interrupt does not have a waiting process, then the alternate event (if any) associated with the interrupt level is signaled. Such a capability is useful for devices whose interrupts cannot be disabled by software but whose device process might have to suspend itself while awaiting the availability of a resource. For example, consider a printer that generates an interrupt when it is taken off-line. If the device handler for the printer is waiting for some process to place a line in a buffer and thus is not suspended on the primary semaphore associated with the printer interrupt, then a spurious interrupt process could be invoked via the alternate semaphore to respond if the printer goes off-line. Alternate interrupt events permit the user to respond in such a situation that would otherwise have to be treated as a system design error.

9.4.3 Interrupt Routines

The current implementation of the Executive RTS does not allow level 0 interrupts to be awaited by user processes. Level 0 is the system restart interrupt and is used by the Executive RTS to restart the complete RTS system, not a single process.

The following types are assumed to be declared by the user.

```
        type interrupt_level: 0..15;
          interrupt_result: -1..15;
```

The following routines are available in the Executive RTS if the user declares the following calling sequences as EXTERNAL.

```
        function intlevel: interrupt_result; external;
```

This function returns the interrupt level of the hardware interrupt currently in service (zero through fifteen) or returns -1 if no interrupt is in progress.

Each of the following routines affect an attribute of a semaphore. The state of the semaphore with respect to its waiting processes or unreceived signals is not changed by the call of the routine.

```
procedure externalevent( sema: semaphore;
  level: interrupt_level ); external;
```

This procedure attaches the SEMA semaphore to the LEVEL interrupt as the primary receiver of an interrupt event. The same semaphore may be attached to more than one interrupt level as primary or secondary receiver. Note that the priority of any process waiting on such a semaphore must be set in such a way that only one interrupt associated with the semaphore can be active at a given instant. That is, the priority of the waiting process must be at least as urgent as the most urgent interrupt. But one interrupt level may be attached to none or one semaphore which is the primary receiver of an interrupt event. If the LEVEL interrupt had been attached to a primary semaphore before the call of EXTERNALEVENT, then that semaphore is no longer attached, and the SEMA semaphore becomes the primary semaphore.

```
procedure altexternalevent( sema: semaphore;
  level: interrupt_level ); external;
```

This procedure attaches the SEMA semaphore to the LEVEL interrupt as the alternate receiver of an interrupt event. The same semaphore may be attached to more than one interrupt level as primary or secondary receiver. But one interrupt level may be attached to none or one semaphore which is the secondary receiver of an interrupt event. If the LEVEL interrupt had been attached to a secondary semaphore before the call of ALTEXTERNALEVENT, then that semaphore is no longer attached, and the SEMA semaphore becomes the secondary semaphore.

```
procedure noexternalevent( level: interrupt_level );
  external;
```

This procedure detaches any semaphore which had been designated as the primary receiver of interrupt events for the LEVEL interrupt. If no semaphore had been so designated, this routine does nothing.

```
procedure noaltexternalevent( level: interrupt_level );
  external;
```

This procedures detaches any semaphore which had been designated as the secondary receiver of interrupt events for the LEVEL interrupt. If no semaphore had been so designated, this routine does nothing.

Figure 9-5 illustrates the use of one program that waits on more than one event. Program SPURIOUS waits on the alternate external event for interrupt levels zero through fifteen. If an unexpected (spurious) interrupt occurs, then a diagnostic message is printed, and then the system is terminated. Note that the priority of SPURIOUS must be 0 since the program must have at least as great an urgency as any interrupt with which it is connected. A program such as SPURIOUS is useful primarily while a system is being initialized. In general each interrupt level should have its own spurious interrupt handler so it is possible to attempt recovery. Since it has no knowledge about the particular devices associated with each interrupt, SPURIOUS cannot clear any interrupt that is handles; there is no alternative but to terminate the system.

```
system test;

  type
    interrupt_level = 0..15;
    interrupt_result = -1..15;
    nonneg = 0..32767;

  procedure initsemaphore( var sema: semaphore;
    count: nonneg ); external;
  procedure wait( sema: semaphore ); external;
  procedure altexternalevent( sema: semaphore;
    level: interrupt_level ); external;
  function intlevel: interrupt_result; external;

  program spurious;
  var
    spurious_interrupt: semaphore;
    msg: packed array[1..30] of char;
    n, stat: integer;
  begin
  {# priority = 0; stacksize = 200; heapsize = 0}
    initsemaphore( spurious_interrupt, 0 );
    for i := 0 to 15 do
      altexternalevent( spurious_interrupt, i );
    wait( spurious_interrupt );
    msg := 'Spurious interrupt at level ??'
    n := 29;
    encode( msg, n, stat, intlevel: 2);
    message( msg );
    { terminate execution}
  end { spurious } ;

begin
  start spurious;
  { start other user programs}
end .{test} .
```

FIGURE 9-5. SPURIOUS INTERRUPT PROGRAM

9.4.4 General Features of Interrupt Handling

9.4.4.1 General Routines. All hardware interrupts may be disabled by calling

```
        procedure mask; external;
```

except for level zero interrupt which is always enabled. This routine causes the interrupt mask in hardware to be set to zero. By using this routine an algorithm can ensure that it finishes a series of steps without being interrupted until the algorithm calls

```
        procedure unmask; external;
```

This routine enables interrupts which are more urgent than the priority of the calling process. The interrupt mask is set to the greater of zero and the calling process's priority minus one; this value is the default mask at which any process executes.

The interrupt mask of the processor is always set according to the mask of the process that is executing. If a process has called MASK, then it cannot be interrupted; however it can relinquish the processor by waiting for an event (either interrupt or semaphore) or by signaling a process with greater urgency. The priority of the new process determines the mask of the processor.

9.4.4.2 Techniques of Code Style. The following figure illustrates several stylistic features of interrupt handling and interfacing. (This is an abstraction of the example in Figure 9-4.)

```
system example;
  const level_5 = 5;
  type interrupt_level = 0..15;
    nonneg = 0..32767;

  procedure initsemaphore( var sema: semaphore;
    count: nonneg ); external;
  procedure wait( sema: semaphore ); external;
  procedure externalevent( sema: semaphore;
    level: interrupt_level ); external;

  program device( interrupt: semaphore;
    level: interrupt_level );
    { service interrupts from a device}
    begin
    {# priority=level; stacksize=200; heapsize=0}
      { perform local initialization}
      while true do { forever}  begin
        { enable interrupts from device}
        wait( interrupt );
        { disable interrupts from device}
        { service device interrupt}
        end { forever loop} ;
    end { device} ;

  procedure initdevice( level: interrupt_level );
    { create device service program}
    var interrupt: semaphore;
    begin
      { initialize device}
      initsemaphore( interrupt, 0 );
        { first WAIT on INTERRUPT causes suspension}
      externalevent( interrupt, level_5 );
      start device( interrupt, level_5 );
    end { initdevice} ;

  begin { example }
  {# priority=1; stacksize=200; heapsize=0}
    initdevice( level_5 );
  end { example } .
```

FIGURE 9-6. EXAMPLE OF STYLE FOR INTERRUPT HANDLING

The code to interface to the device is partitioned into two modules. Program DEVICE actually responds to interrupts. It has two parameters: INTERRUPT, the semaphore on which it will wait for an interrupt to occur, and LEVEL, the interrupt level associated with the device which must be used to specify the concurrent characteristic PRIORITY (and hence the mask of DEVICE). Since the semaphore that is associated with the interrupt is passed as a parameter instead of being allocated and identified as an external event within DEVICE, this handler can be activated either directly by an interrupt or indirectly by an interrupt demultiplexer. The body of DEVICE has the structure of a block of initialization code followed by an perpetual loop in which individual interrupts are serviced. The loop should begin with code that manipulates the device so that interrupts are enabled. Then a call is made to the Executive RTS to suspend the process until the next interrupt. After the interrupt occurs, further interrupts are disabled, and code is executed to handle the last interrupt (which generally involves some type of acknowledgment to the hardware). Finally a branch is made back to the top of the loop to re-enable interrupts and start the sequence again.

Note that there are two senses in which interrupts can be disabled within a device process. One is via the processor mask. If an appropriate priority has been chosen for the process, the processor mask of the device process protects it from further interrupts at the same level; however such an interrupt is held pending until the mask is raised. The other way to disable interrupts is to command the device interface not to permit them. In general it is wise, while servicing an interrupt, to disable further interrupts via the device interface since the processor mask could be raised if the handler had to suspend itself while waiting for a resource and a process with lesser urgency was activated.

The other module of the device handler is the procedure INITDEVICE which can be called to create an instance of the device process. The parameter to INITDEVICE is the interrupt level to be associated with the device. A semaphore INTERRUPT is declared, initialized, and connected to the interrupt at level LEVEL. Then program DEVICE is started with INTERRUPT and LEVEL as parameters. In general it is a good practice to isolate the creation of a device process within a routine such as INITDEVICE that has as its parameters the externally known characteristics of the device. Such an initialization routine can be invoked as needed without the user having to be aware of the details of process creation and initializtion.

It may appear incorrect for INITDEVICE to declare the semaphore INTERRUPT, pass it (by value) to program DEVICE, and then return, thus deallocating the variable INTERRUPT. As was explained in Section 9.3.5, a variable of type SEMAPHORE is actually a reference to an RTS-managed data structure. A semaphore passed by value is implemented by passing a reference to the associated structure. When a procedure returns in which a semaphore is declared, the data cell that contained a reference (pointer) to the structure is deallocated; the structure is deallocated only if TERMSEMAPHORE is called.

9.5 SCHEDULING OF DEVICE AND NON-DEVICE PROCESSES

The Executive RTS scheduling policy that is presented in Section 9.1 treats differently the scheduling of device and non-device processes having the same urgency: a device process (one with priority 0-15) preempts a process with the same priority; a non-device process does not. Such a distinction may appear to be inconsistent. However, as was discussed in that section, it is a good design practice to structure a system of processes so they interact correctly regardless of their relative priorities. Priority should affect only the urgency that is associated with the execution of process, not its algorithmic behavior. The choice for the treatment of processes with the same priority was based primarily upon the efficiency of the resulting implementation. The architecture of the TI 990/9900 and the data structures used within the Executive RTS make it significantly easier to place a non-device process in the scheduling queue than to preempt the active process; the converse is true for a device process. Such considerations of efficiency are important since it will not be uncommon for one process to activate another with the same priority. For device processes in particular, interrupt multiplexing results in one process signaling (scheduling) another process with the same priority, so preemption is appropriate.

SECTION X

PROCESS COMMUNICATION

A process must communicate with its external environment to perform any practical function. A process can communicate with other processes and with devices, which behave similarly to processes. The vehicle of the communication can be very simple such as shared memory or it may be very sophisticated such as the Executive RTS interprocess file system. This section describes the different mechanisms available to the user for communication and is divided into two parts. The first part discusses low-level mechanisms such as CRU, memory mapped I/O and shared memory. The second part discusses a more sophisticated form, the Executive RTS logical file system which is used to communicate with both devices and processes in an independent manner using standard Microprocessor Pascal System input/output operations.

10.1 SIMPLE COMMUNICATION MECHANISMS

Simple communication mechanisms are easily implemented and usually require very little overhead. They are, however, the most primitive forms available and should be used to implement more flexible communication systems. The mechanisms to be discussed here include:

- Device communication using CRU
- Device communication using memory-mapped I/O
- Interprocess communication using shared variables
- Interprocess communication using message buffers

10.1.1 Device Communication Using CRU

The Communications Register Unit (CRU) is the general-purpose, command-driven hardware interface of the TI 990/9900 family and is used to communicate with many supported devices. The CRU is addressed as a data space consisting of 4K consecutive bits which are addressed by the CRU input/output commands as independent bits or in groups of up to 16 bits. The CRU is addressed only by these input/output commands. The CRU bus is totally separate from the memory bus. Furthermore, input and output bits may be separate and unrelated so the CRU is best visualized as a 4096-bit input register and a 4096-bit output register.

Most devices are interfaced through 16 consecutive bits of the CRU space. Therefore, the CRU commands access the CRU space using a base address and, in some cases, a bit displacement; the base is a variable and the displacement is a constant. This design facilitates the implementation of a reentrant device handler which services identical devices having unique CRU bases.

The following standard procedures and function are provided for CRU access in Microprocessor Pascal. These routines are pre-declared within the compiler and should not be declared by the user. The responsibility for manipulation of the CRU base is given to the user.

```
    type
      base_range = 0..#1FFE;
      width_range = 1..16;
      displacement_range = -128..127;

    procedure crubase(base: base_range)

    procedure ldcr(width: width_range; value: integer)

    procedure sbo(displacement: displacement_range)

    procedure sbz(displacement: displacement_range)

    procedure stcr(width: width_range; var value: integer)

    function tb(displacement: displacement_range): boolean
```

The parameters DISPLACEMENT and WIDTH must be compile-time constants; permitting non-constant parameters would complicate in-line expansion of these routines.

10.1.1.1 Procedure CRUBASE

```
    procedure crubase(base: base_range)
```

This procedure establishes a routine-local CRU base. The value of the single parameter BASE is twice that of the actual hardware CRU address. This is the value that will be placed in R12 and allows a more efficient in-line expansion. The actual hardware address is used to bias CRU displacements. This value will be used as the CRU base in subsequent CRU operations done in the current routine. Each routine doing CRU operations has a routine-local CRU base which can be manipulated only by that routine. It is possible for CRUBASE to be called more than once in a particular routine to modify the current CRU base.

10.1.1.2 Procedure LDCR (Load CRU).

```
procedure ldcr(width: width_range; value: integer)
```

This procedure outputs the WIDTH least significant bits of VALUE to consecutive CRU bits beginning at the established CRU base. The parameter WIDTH must be a compile-time constant.

10.1.1.3 Procedure SBO (Set Bit to One).

```
procedure sbo(displacement: displacement_range)
```

This procedure outputs a "1" to the CRU bit whose address is the sum of the established CRU base and DISPLACEMENT. The value used for the CRU base is the hardware address which is half the value of the parameter to CRUBASE. The parameter DISPLACEMENT must be a compile-time constant.

10.1.1.4 Procedure SBZ (Set Bit to Zero).

```
procedure sbz(displacement: displacement_range)
```

This procedure outputs a "0" to the CRU bit whose address is the sum of the established CRU base and DISPLACEMENT. The value used for the CRU base is the hardware address which is half the value of the parameter to CRUBASE. The parameter DISPLACEMENT must be a compile-time constant.

10.1.1.5 Procedure STCR (Store CRU).

```
procedure stcr(width: width_range; var value: integer)
```

This procedure inputs the WIDTH CRU bits at the established base into the least significant bits of VALUE. All other bits of VALUE are cleared. The parameter WIDTH must be a compile-time constant.

10.1.1.6 Function TB (Test Bit).

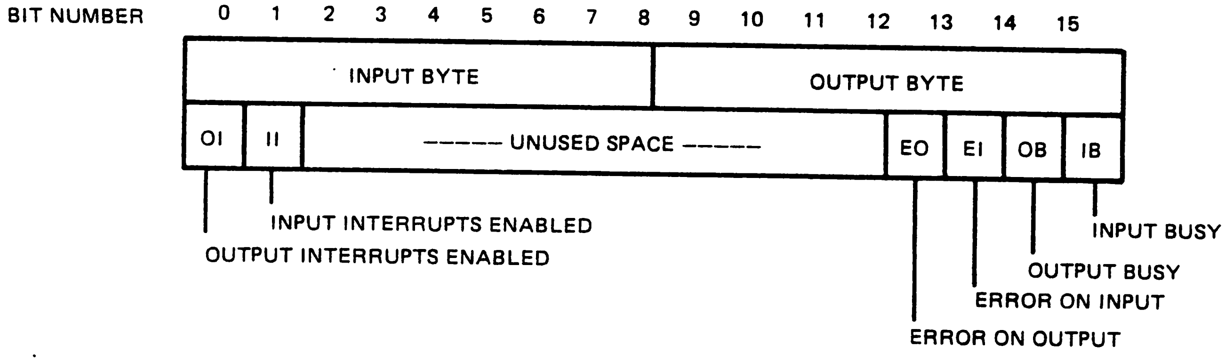
```
function tb(displacement: displacement_range): boolean
```

This function returns the value of the CRU bit whose address is the sum of the established CRU base and DISPLACEMENT. The value used for the CRU base is the hardware address which is half the value of the parameter to CRUBASE. The value TRUE is returned if the CRU bit is "1" and FALSE is returned if the CRU bit is "0". The parameter DISPLACEMENT must be a compile time constant.

10.1.2 Device Communication Using Memory-Mapped I/O

Some devices are interfaced to the processor on the memory and address busses. Communication to these devices is done by reading and writing into "memory locations" dedicated to the device. This type of I/O is referred to as memory-mapped I/O and is supported by the Executive RTS. The user describes the structure of the device's dedicated memory space in the type declaration of a packed record, referred to as a control record. Following in Figure 10-1 is an example device control record represented first as a diagram and then as a Pascal type declaration. The MAP option of the compiler can be used to check that the template formed by the type declaration of the packed record matches the bit placements required.

A variable must be declared as a pointer to a control record of type DEVICE_INSTANCE and initialized to the address of the control space. A type transfer will probably be necessary to assign an integer to the pointer. It is suggested that the address of the control space be passed as a process parameter allowing the process to be reentrant. Manipulation of the control space is done indirectly through a local pointer of type DEVICE_INSTANCE. Such a technique facilitates both the implementation of reentrant device handlers and migration from a prototype system to the target system. A device handler is reentrant because multiple instances of it may exist, each manipulating a different control space of identical structure. The migration is simplified because of the ease in selecting the location of the control space which may be different in the prototype and target systems. Following is an example of control space manipulation.



```
TYPE
 DEVICE–INSTANCE = @CONTROL–RECORD;
 CONTROL–RECORD = PACKED RECORD
    INPUT–BYTE, OUTPUT–BYTE: 0..=FF;
    OUTPUT–INTERRUPTS–ENABLED, INPUT–INTERRUPTS–ENABLED: BOOLEAN;
    FILL: 0..=3FF ( UNUSED SPACE );
    ERROR–ON–OUTPUT, ERROR–ON–INPUT: BOOLEAN;
    OUTPUT–BUSY, INPUT–BUSY: BOOLEAN;
    END ( CONTROL–RECORD );
```

Figure 10-1. Interface to Memory-Mapped I/O Device

```
process device_handler(address: integer);
var control_record_ref: @control_record;
  ch: char;
begin
control_record_ref::integer := address;
with control_record_ref@ do begin
  while true do begin
    while input_busy and output_busy do .poll ;
    if not input_busy then begin
      ch := chr(input_byte);
      input_busy := true {start input of next byte};
      {do something with ch}
      end {if};
    if not output_busy then begin
      output_byte := ord( '*' );
      output_busy := true {start output of byte};
      end {if}
    end {while true}
  end {with control_record_ref@}
end {process};
```

Figure 10-2. Manipulation of Memory-Mapped I/O Device

It should be noted that if the same control space were to be manipulated by more than one process concurrently, synchronization would be necessary to guarantee exclusive access to the control space.

10.1.3 Interprocess Communication Using Shared Variables

The simplest form of interprocess communication is accomplished through the sharing of variables such as integers or simple record structures. The Microprocessor Pascal scope rules allow processes to share variables with other processes within which they are nested. Also, variables in a heap can be shared among processes since pointers may be passed as parameters to processes. However, only a single process should be allowed to operate on a variable at a time. Therefore, a semaphore is used to guarantee exclusive access to each shared variable while it is being manipulated. This is done by treating each shared variable as a resource and allocating this resource to a single process at a time. The most convenient way to represent a shared variable is as a record structure containing a semaphore (initialized to "1") used to guarantee mutual exclusion with respect to the record. Code accessing a shared variable must be bracketed within a WAIT and a SIGNAL to ensure exclusive access. Figure 10-3 is an example of declaring and modifying a shared variable.

```
var
  b: {shared  record}
    mutex: semaphore;
    next: 1..10;
    end {b};
...
with b do begin {initialize shared variable}
  initsemaphore(mutex, 1);
  next := 1
  end {with b};
...
with b do begin wait(mutex) {guarantee mutual exclusion};
  next := next mod 10 + 1;
  signal(mutex) {release exclusive access} end {with b};
...
with b do begin {data no longer shared}
  termsemaphore(mutex); end { with b } ;
```

FIGURE 10-3. EXAMPLE OF SEMAPHORE CONTROL OF SHARED VARIABLES

10.1.4 Interprocess Communication Using Message Buffers

A message buffer is a shared data structure through which messages are transferred and buffered among processes. A message is any structure which can be copied from one process to another; examples are a string of characters, an integer, an array, a record or a pointer. A process may send a message through a message buffer without waiting for the message to be copied by the consuming process. The producing process is suspended only if the buffer space required is not available.

Since message buffers are implemented by the user, they may be viable alternatives in applications where the Executive RTS interprocess file system is not adequate. Figure 10-4 is an example implementation of a message buffer.

```
const
  max_messages = 10 {maximum messages to buffer};
type
  polar_coordinates = record r, theta: real end;
  message = polar_coordinates;
  message_index = 1..max_messages;

  message_buffer = {shared} record
    mutex: semaphore {ensures mutual exclusion};
    not_empty: semaphore;
    not_full: semaphore;
    next_in: message_index;
    next_out: message_index;
    buffer: array  message_index  of message;
    end {message_buffer};
```

FIGURE 10-4. EXAMPLE IMPLEMENTAITON OF MESSAGE BUFFERING DATA

The declaration of the message buffer is in the form of a record. Its first element is a semaphore MUTEX that is used to ensure mutual exclusion of processes accessing the record. The semaphore NOT_EMPTY is used to ensure that the buffer is not empty when removing messages from it. The count field of the semaphore NOT_EMPTY is always the number of messages currently contained in BUFFER. Therefore, a process performing a wait on NOT_EMPTY is suspended if no messages are present; otherwise the count of messages is decremented. The semaphore NOT_FULL is used to ensure that there is an available element in BUFFER to contain a message. The count field of the semaphore NOT_FULL is always the number of elements of BUFFER not containing messages. Therefore, a process executing a wait on NOT_FULL is suspended if the buffer space is not available; otherwise the number of available elements is decremented. The variable NEXT_IN indicates the next element of BUFFER to contain incoming messages. The variable NEXT_OUT indicates the next element of BUFFER containing a message to be sent. The variable BUFFER is managed as a circular buffer of messages.

The operations on message buffers are INITIALIZE, SEND, RECEIVE, and TERMINATE and are shown in Figure 10-5 below.

```
procedure initialize(var b: message_buffer);
  begin with b do begin
    initsemaphore(mutex, 1) {allow 1 process to access at a time};
    initsemaphore(not_empty, 0) {number of messages present};
    initsemaphore(not_full, max_messages) {available buffers};
    next_in := 1 {index of first in-comming message};
    next_out := 1 {index of first out-going message};
    end {with b}
  end {initialize};

procedure send(var b: message_buffer; m: message);
  begin with b do begin
    wait(not_full) {wait until a buffer is available};
    wait(mutex) {get exclusive access to record};
    buffer next_in  := m {insert message in buffer};
    next_in := next_in mod max_messages + 1 .update index ;
    signal(mutex) {release access to record};
    signal(not_empty) {indicate another message present};
    end {with b}
  end {send};

procedure receive(var b: message_buffer, var m: message);
  begin with b do begin
    wait(not_empty) {wait until a buffer is available};
    wait(mutex) {get exclusive access to record};
    m := buffer next_out  {remove message from buffer};
    next_out := next_out mod max_buffers + 1 {update index};
    signal(mutex) {release exclusive access};
    signal(not_full) {indicate another available buffer};
    end {with b}
  end {receive};

procedure terminate(var b: message_buffer);
  begin with b do begin
    wait(mutex);
    termsemaphore(mutex);
    termsemaphore(not_empty);
    termsemaphore(not_full);
    end {with b};
  end {terminate};
```

FIGURE 10-5. EXAMPLE IMPLEMENTATION OF MESSAGE BUFFERING

Notice that a dangerous problem can occur if the WAIT(MUTEX) precedes WAIT(NOT_EMPTY) in procedure RECEIVE. Suppose a process executes RECEIVE and is suspended on NOT_EMPTY because no messages are present to receive. Then MUTEX is left in a locking state. Any other process that executes SEND or RECEIVE is suspended on the MUTEX mutual exclusion semaphore. Therefore, no other process is able to SEND a message for the first process to RECEIVE. All processes sharing the message buffer become suspended forever. Semaphores are low-level synchronization tools that must be used with great care. Interprocess files (Section 10.2) provide a much higher level interface and should be used when possible.

10.2 EXECUTIVE RTS FILES

Typically files are associated with storage media such as disc or magnetic tape. However, many operating systems also allow devices to be treated as files in a consistent manner. Therefore, the term logical file is used to indicate any communication medium with which programs can perform logical I/O (device independent I/O). A logical file can be a disc file in a directory, a VDT, a card reader, a line printer, a spooler, etc. Because of the uniform interface used in logical I/O, programs do not have to be aware of unique characteristics of devices or disc files when doing logical I/O.

The Executive RTS logical files are manipulated through variables of type FILE. A FILE type is a structure which consists of a sequence of components which are all of the same type. The number of components, called the length of the file, is not fixed and may grow to any size depending on the source or destination of the file components.

Allowing logical files to include interactive devices, such as video display terminals (VDTs), permits the sequence of components to be generated in real time by an intelligent source (a human at the keyboard) as opposed to simply reading previously generated components from a storage medium. The generation of these components may also be influenced by output of the program, which is produced in real-time and displayed on the screen of the VDT. Both the program and the human are probably influenced by each other in their real-time interactions. In an abstract sense, the program and the human interacting with each other form a system of two cooperating processes.

The Executive RTS approach to interprocess communication is to treat it as a form of logical I/O: cooperating processes may communicate with each other using the Executive RTS logical files. One process may read file components which are being written concurrently by another process. This is very similar to the interaction described above because the input is generated and the output consumed in real-time by the processes rather than being stored on or retrieved from some storage device. In the Executive RTS, a logical file can be associated with a device, a disc file, or another process. (However, the initial release of the Microprocessor Pascal System does not support disc file management.)

The Executive RTS file I/O provides a consistent logical interface between system components, which include both hardware devices and software programs. This allows systems to be constructed from modular components, each of which is understood in terms of its inputs and outputs. In this way, the interfaces between the components form a nearly complete definition of the system.

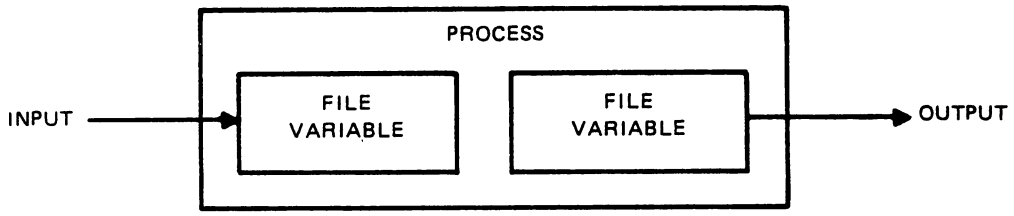


FIGURE 10-6. FILE VARIABLES AS PROCESS-LOCAL PORTS

Each component can be designed and implemented independently and can be tested in isolation from other units to verify that it performs its required function.

A system component can be replaced by a "plug-in-compatible" test component that injects test data into the system or monitors the data generated by other parts of the system.

10.2.1 Process-Local File Variables

An Microprocessor Pascal file variable is actually a process-local port which interfaces the process with its external environment (Figure 10-6).

Each file variable has a name, in the form of a character string, which indicates with which logical unit it is associated. A file variable declared in the VAR section is initially given the name of the identifier of the variable (truncated to eight characters). For example, a file variable declared as

```
    var
      printer: text;
```

has an initial name of "PRINTER". (Lower case letters are not significant.) The standard function FILENAMED can be used when passing files by value. Its calling sequence is:

```
    function filenamed(s: "any string"): anyfile;
```

The result of the function is a file with the initial name equal to the specified string. For example,

```
    program copy(input, output: text); forward;
    ...
    start copy( filenamed('reader'), filenamed('printer') );
```

causes the standard INPUT and OUTPUT files of COPY to be named "READER" and "PRINTER", respectively. The standard procedure SETNAME can be used to modify the name of a file variable to the specified string. Its calling sequence is:

```
    procedure setname(var f: anyfile; s: "any string");
```

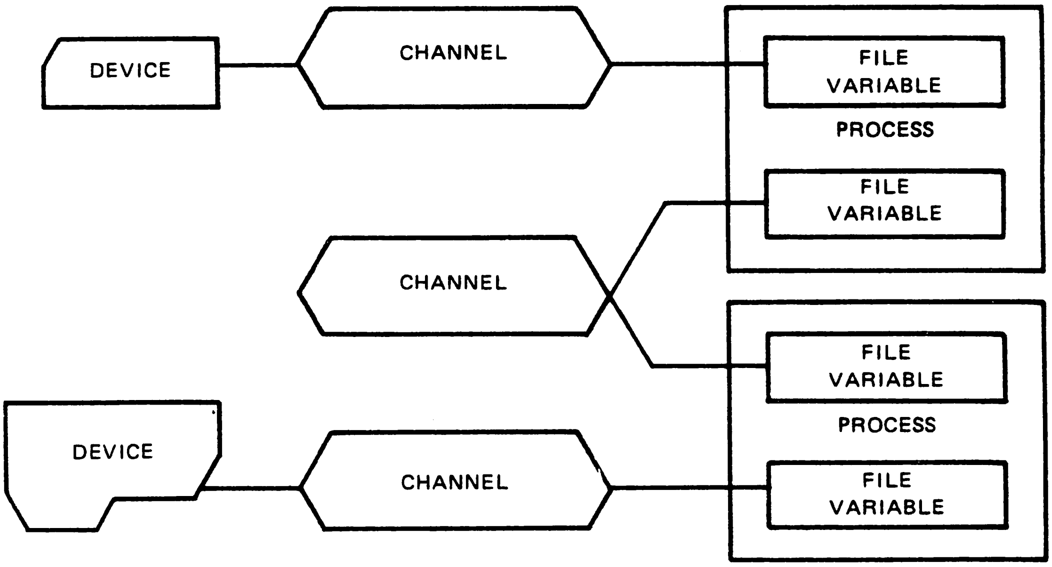


FIGURE 10-7. CHANNEL CONNECTIONS

Lower case characters in S are insignificant and trailing blanks are stripped. For example:

```
    setname(f, 'MagTape     ')
```

changes the name of F to "MAGTAPE".

10.2.2 Channels

Channels are shared data structures through which file variables are linked to devices and to other file variables. A channel conducts information among file variables and devices and synchronizes the execution of the participating processes. Channels may optionally have the capability to buffer components, allowing producers to proceed before components are consumed. Figure 10-7 illustrates the connections among file variables and devices with channels.

Each channel has a name, in the form of a character string, which is identical to the names of all file variables connected to it. Channels are automatically maintained by the Executive RTS and may be completely transparent to the user.

10.2.3 Device Channels

Each physical device in a system is identified by an alphanumeric name from one to eight characters in length and has a dedicated channel of the same name. Therefore, any logical I/O done with the channel "PRINTER" results in physical I/O on the device "PRINTER" (Figure 10-8).

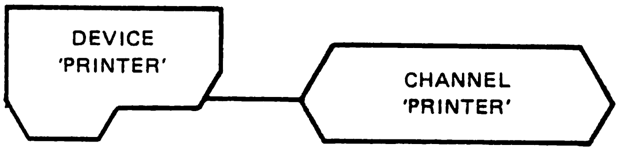


FIGURE 10-8. LOGICAL DEVICE AND ASSOCIATED DEVICE CHANNEL

This means that the user can dynamically select a device for I/O by calling the standard procedure SETNAME for a locally declared file variable. When the file is opened with a REWRITE, it becomes connected to the corresponding device channel.

10.2.4 Connection of File Variables to Channels

Before a file can be used for I/O, it must be opened. Microprocessor Pascal files are opened for writing and reading by the standard procedures REWRITE and RESET, respectively. Both of these procedures close the file first if it was previously opened and then open it in the appropriate mode. The standard procedure CLOSE simply closes the file if opened. This RTS procedure must be explicitly declared if used. its calling sequence is:

```
     procedure close( var f: anyfile ); external;
```

Exiting a routine in which a file variable is declared also causes an implicit close operation on the file.

When a file variable is opened, it is connected to a channel of the same name as the file. If no channel exists by that name, one is implicitly created and given the appropriate characteristics. A file variable is disconnected from a channel when it is closed. If no file variables are left connected to a particular channel as a result of a close, that channel is normally destroyed.

As an example, consider a pagination program which reads lines from its input file and formats them into pages with headings and page numbers (Figure 10-9).

```
program pagination(input, output: text);
  var ch: char; page_number, line_number: integer;
    heading: packed array  [1..72]  of char;
  begin .pagination  reset(input); rewrite(output);
  for i := 1 to 72 do begin {read first line into HEADING};
    if eoln then ch:=' ' else read(ch);
    heading i  := ch
    end {for i := 1 to 72};
  readln; page_number := 1;
  while not eof do begin
    writeln(heading, ' PAGE ', page_number: 1); writeln;
    line_number := 3;
    while line_number  =<56 and not eof do begin
      while not eoln do begin read(ch); write(ch) end;
      writeln; readln; line_number := line_number + 1
      end {while line_number <= 56 and not eof};
    page(output); page_number := page_number + 1
    end {while not eof}
  end   {pagination}.
```

FIGURE 10-9. PAGINATION PROGRAM

Assuming that there is a card reader named READER and a line printer named PRINTER, this program can be used to list a deck of cards by invoking it as follows:

```
    start pagination( filenamed('reader'), filenamed('printer') );
```

When the standard file INPUT is opened with RESET, it is connected to the device channel named READER. The standard file OUTPUT is implicitly opened with a REWRITE on entry to the program, and is connected to the device channel named PRINTER. The program then copies the text from the reader to the printer adding headings, page numbers, and page separations. When EOF is detected on INPUT, the program terminates causing the automatic closing (and disconnection) of both files.

Now consider the program in Figure 10-10 which reads polar coordinates from a text file and writes both the polar coordinates and equivalent rectangular coordinates to another text file.

```
program coordinate_conversion(input, output: text);
  var r, theta: real;
  begin {coordinate_conversion}
  rewrite(output);
  writeln('POLAR TO RECTANGULAR COORDINATE CONVERSIONS');
  writeln; writeln .skip two lines ;
  writeln(' ':20, 'R':10, 'THETA':10, 'X':10, 'Y':10);
  writeln .skip one line ;
  reset(input);
  while not eof do begin
    readln(r, theta);
    writeln(' ':20, r:10:2, theta:10:2,
      r*cos(theta):10:2, r*sin(theta):10:2)
    end {while not eof}
  end   {coordinate_conversion};
```

FIGURE 10-10. COORDINATE CONVERSION PROGRAM

This program can also be invoked to read from the card reader and print on the line printer as follows:

```
start coordinate_conversion
  ( filenamed('reader'), filenamed('printer') );
```

However, the output is not segmented into pages and is printed over perforations in the paper. It is possible to allow the program PAGINATION (Figure 10-9) to process the output of COORDINATE_CONVERSION (Figure 10-10) before it is printed. Figure 10-11 illustrates the relationship between the two programs that can be accomplished if the Executive RTS logical files are utilized for interprocess communication.

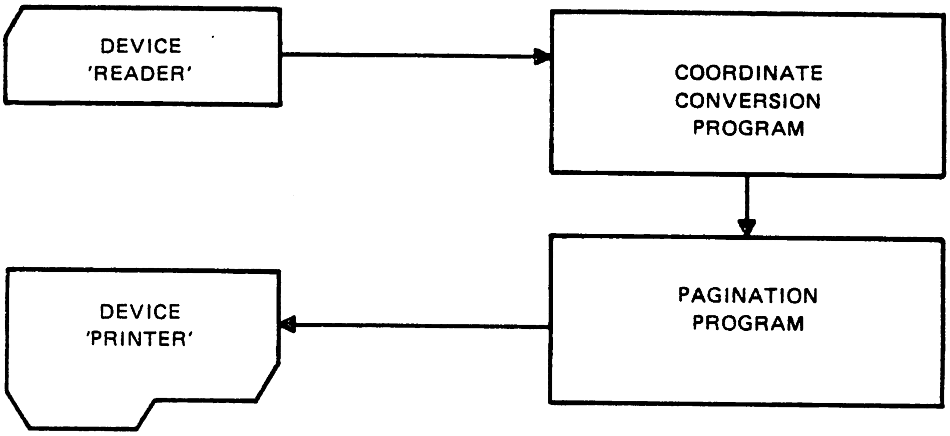


FIGURE 10-11. COMMUNICATION AMONG PROGRAMS AND DEVICES

The following invocations initialize this system:

```
    start coordinate_conversion
      ( filenamed('reader'), filenamed('private') );

    start pagination
      ( filenamed('private'), filenamed('printer') );
```

In this case, the output of COORDINATE_CONVERSION is transmitted as the input to PAGINATION through a channel named PRIVATE. This channel is created automatically by the first program that attempts to connect to it. When COORDINATE_CONVERSION detects EOF on its input file, it terminates, causing its output file to be disconnected from the channel. This causes an EOF indication on the input file of PAGINATION because it is connected to same channel. PAGINATION then terminates, causing both of its files to be closed, which causes the automatic destruction of the channel PRIVATE.

The Executive RTS logical files facilitate the production of general purpose utilities like PAGINATION which can be used in many different configurations to perform commonly needed functions.

A single channel may have more than one reading and/or writing file variable connected to it. However, it is possible to limit either or both for a particular channel, thus providing a way to implement both shared access and exclusive access devices.

10.2.5 Sequential (Non-Text) File Operations

Sequential files are used to transmit or receive values to or from their associated channels in "binary" format. There is no automatic data conversion when using sequential files. The standard procedure READ is used to receive into a variable the next component from the channel. The execution of this procedure may cause suspension of the process if the next component has not yet been produced. The reading process is activated again when the component becomes available. The standard procedure WRITE is used to transmit the value of a variable as the next component to the channel associated with the file. If no buffer is available for the new component and no other file variable connected to the channel is waiting for a component, the process is suspended. It is activated again when a buffer becomes available or another process reads from the channel.

10.2.6 Text File Operations

Text files are used to transmit or receive file components to or from their associated channels in "character" format. The components of a text file are lines of text which are encoded and decoded automatically when text file operations are performed. Associated with each text file is a line buffer, which contains the component being encoded or decoded, and a column index which indicates the current character position within the line buffer.

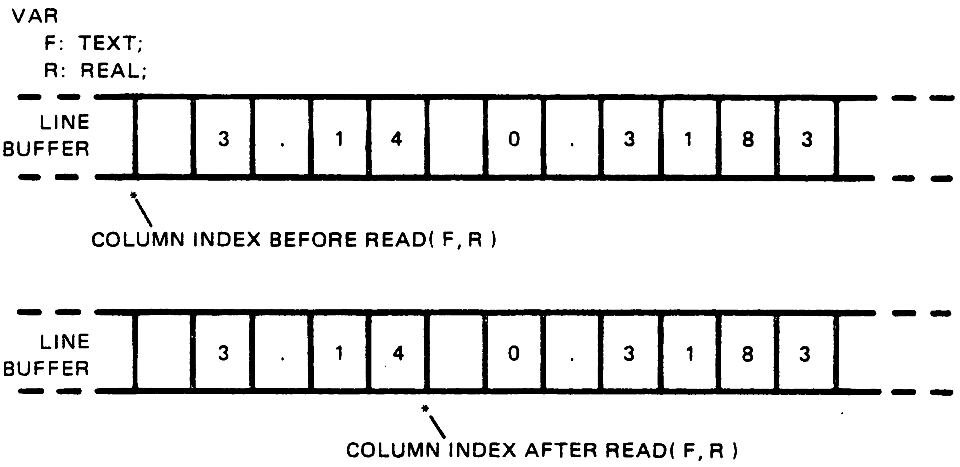


FIGURE 10-12. COLUMN INDEX IS INCREMENTED DURING TEXT READ

Operations on a text file cause the column index to be incremented as characters are produced or consumed in the line buffer. Figure 10-12 illustrates the effects on the column index during the read of a real number from a text file.

As the column index is incremented off the end of the line buffer, it is logically placed at the beginning of the next line. However, for a reading text file, the next component is not received from the channel until it is needed. The standard procedure READLN is used to logically advance the column index to the beginning of the next line, but does not cause the next component to be received from the channel. Figure 10-13 illustrates the effect of READLN on a text file.

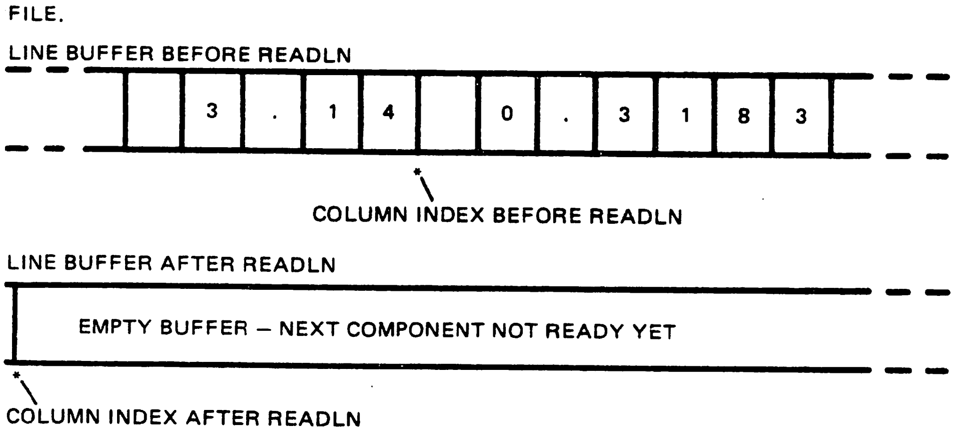

FILE.
LINE BUFFER BEFORE READLN
3
.
1
4
0
.
3
1
8
3
*
COLUMN INDEX BEFORE READLN
LINE BUFFER AFTER READLN
EMPTY BUFFER – NEXT COMPONENT NOT READY YET
*
COLUMN INDEX AFTER READLN

FIGURE 10-13. EFFECT OF READLN ON READING TEXT FILE

The standard procedure RESET leaves the text file with the column index at the beginning of the first line, but the line has not been received from the channel. A READ performed at this time will cause the next component to be received before any decoding is done (Figure 10-14.)

The standard function EOLN is used to detect the end of line when reading a text file. It returns TRUE if the last character on a line has been read. Logically, there is a blank character between lines, so when EOLN = TRUE, the column index is not at the beginning of the next line.

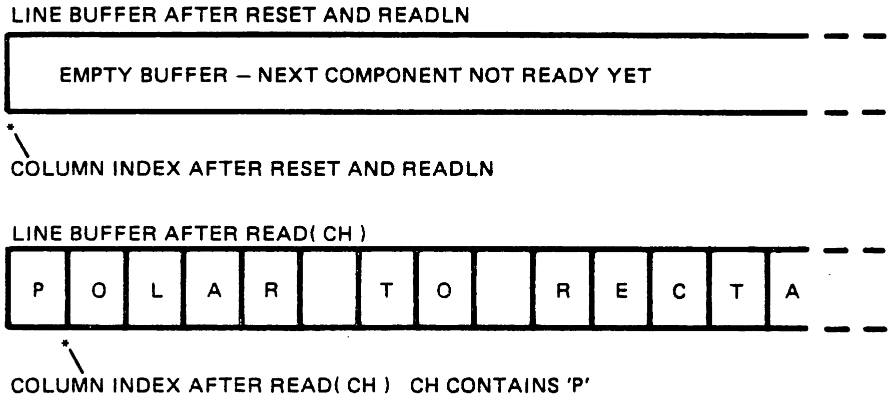

LINE BUFFER AFTER RESET AND READLN
EMPTY BUFFER – NEXT COMPONENT NOT READY YET
*
COLUMN INDEX AFTER RESET AND READLN
LINE BUFFER AFTER READ(CH)
P
O
L
A
R
T
O
R
E
C
T
A
*
COLUMN INDEX AFTER READ(CH) CH CONTAINS 'P'

FIGURE 10-14. EFFECT OF READING FIRST CHARACTER ON LINE

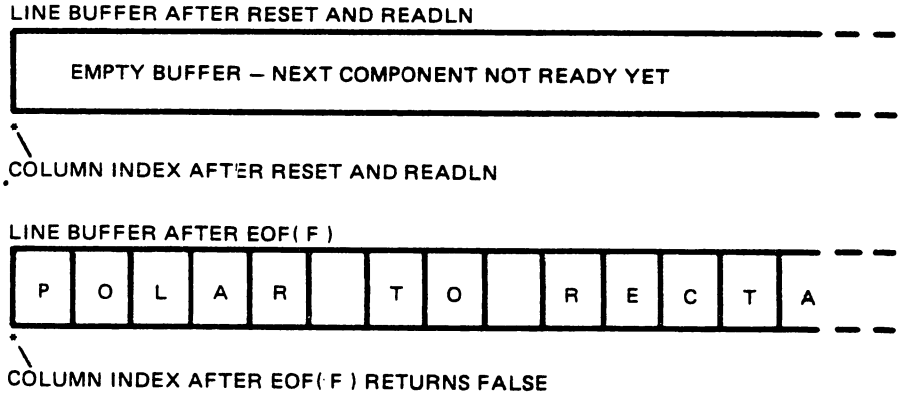

LINE BUFFER AFTER RESET AND READLN
EMPTY BUFFER – NEXT COMPONENT NOT READY YET
COLUMN INDEX AFTER RESET AND READLN
LINE BUFFER AFTER EOF(F)
P O L A R T O R E C T A
COLUMN INDEX AFTER EOF(F) RETURNS FALSE

FIGURE 10-15. EFFECT OF EOF(F) WHEN RESULT IS FALSE

However, reading the blank character or performing a READLN places it there.

The standard function EOF is used to detect the logical end of file for both sequential and text files. (The cause of logical end of file is discussed in a subsequent paragraph.) If EOF(F) is TRUE, then no information can be read from F. For reading text files, EOF(F) can only be TRUE if the column index is at the beginning of the line. If the column index is at the beginning of the line and the line buffer is empty and EOF is called, an attempt is made to receive the next component from the channel. If it is received, EOF returns FALSE. If the attempt fails, EOF returns TRUE. Figure 10-15 illustrates the effect of EOF when FALSE is returned.

The standard procedure WRITELN is used to transmit a writing text file's line buffer as the next component of the channel associated with the file. WRITELN also causes the column index to be placed at the beginning of the next line. An implicit WRITELN is performed if a WRITE causes the column index to go beyond the end of the line buffer.

10.2.7 Random File Operations

The initial release of the Microprocessor Pascal System does not support files of type RANDOM. If an operation is attempted on a random file, a run-time exception occurs.

10.2.8 Logical End of File

The standard function EOF is used to detect the logical end of file for files opened for reading. If EOF(F) is FALSE and F is a sequential file opened for reading, it is possible to read at least one more component. If EOF(F) is FALSE and F is a text file opened for reading, it is possible to read at least one more character. If EOF(F) is TRUE and a READ(F, ...) is attempted, a run-time exception occurs.

A logical end of file occurs on all reading files connected to a channel if end of transmission has occurred on the channel and all buffered components have been consumed. End of transmission means that all writing files connected to the channel have closed. Once end of transmission occurs on a channel, all reading files must close before the end of transmission status is removed. Files attempting to connect to a channel with an end of transmission status are suspended until the status is removed and are then connected.

10.2.9 Logical End of Consumption

A logical end of consumption occurs on a channel when all connected reading files become closed. Normally this is not considered an exception, and connected writing files may continue to write to the channel. However, attempts to write to the channel cause suspension until reading files become connected again.

The Executive RTS procedure F$STEOC(F) (read as "set end of consumption") may be called to indicate that end of consumption on channels associated with F is to be handled in a similar manner to end of transmission. In this case, when all reading files disconnect, no files are allowed to connect to the channel until all connected writing files close. The Executive RTS function F$EOC(F) returns a boolean indicating that end of consumption has occurred on the channel associated with the file F. The calling sequences for F$STEOC and F$EOC are:

```
    procedure f$steoc(var f: anyfile); external;

    function f$eoc(var f: anyfile): boolean; external;
```

10.2.10 Buffers Associated With File Variables

As discussed previously, each text file has a line buffer associated with it. A reading sequential file has a look-ahead buffer used when EOF is called. EOF returns TRUE if end of transmission has occurred on the associated channel and all buffered components are consumed. Otherwise, it returns FALSE and receives the next component into the look-ahead buffer. This ensures that another component is available to be read from the file which is contained in the look-ahead buffer. The first READ after a call to EOF retrieves the component from the look-ahead buffer rather than the channel.

These buffers are also used by the channel to allow producers to proceed before components are consumed. The Executive RTS procedure F$CHBUFFERS(F,N) may be used to ensure that any channels associated with F have the capability of buffering at least N components before producers are suspended. The calling sequence of F$BUFFERS is:

```
    procedure f$chbuffers(var f: anyfile; n: integer); external;
```

A call to F$CHBUFFERS may cause more buffers to be allocated to a channel. If so, these buffers remain the property of the channel until all files disconnect.

10.2.11 Connections of Files with Different Component Types

File variables connected to a common channel may have components of different types. However, the lengths of the components must be the same. Consider the following type declarations:

```
  type
    filetype1 = file of array [1..2] of integer;
    filetype2 = file of record field1, field2: integer end;
    filetype3 = file of integer;
```

FILETYPE1 and FILETYPE2 have components of the same size so variables of these types may be connected to the same channel. However, FILETYPE3 has components that are shorter than the components of FILETYPE1 or FILETYPE2. Therefore, a variable of this type cannot be connected to the same channel that variables of the first two types are connected.

One of the characteristics of a channel is the channel component length. This is initialized to the component length of the first file variable to connect to it. Any sequential file variables which subsequently connect to the channel must have an identical component length. Any text file variables which subsequently connect to the channel will automatically have a maximum line length which is the same as the channel component length. The Executive RTS function F$CLENGTH(F) returns the component length of the file F. If F is an open text file, then this is the maximum line length. The calling sequence of F$CLENGTH is:

```
  function f$clength(var f: anyfile); external;
```

If a text file is the first file variable to connect to a channel, the channel is given a component length equal to the default maximum line length of the text file. The initial default is 80 but may be modified by the Executive RTS procedure F$STLENGTH(F,LENGTH). For example, if F$STLENGTH(F,132) is executed and F is the first file variable to connect to a channel, the component length of that channel would be 132. The calling sequence of F$STLENGTH is:

```
procedure f$stlength(var f: anyfile; length: integer); external;
```

10.2.12 Conditional READs and WRITEs

Each sequential file has a conditional attribute, represented by a boolean, which indicates that each READ and WRITE is to be performed only on the condition that a buffer is available and it is not necessary to wait. If a buffer is not available, the READ or WRITE is not performed. The Executive RTS procedure F$CONDITIONAL can be used to alter the conditional attribute of a sequential file. (Text and random files may not have the conditional attribute.) The calling sequence of F$CONDITONAL is:

```
procedure f$conditional(var f: anyfile; conditional: boolean);
  external;
```

Calling this procedure causes the conditional attribute to be set to CONDITIONAL. The attribute defaults to FALSE so that normally, READs and WRITEs wait for buffers. The function F$LASTSUCCESSFUL(F) indicates that the last channel trnasfer made by F was successful. Its calling sequence is:

```
function f$lastsuccessful(var f: anyfile): boolean; external;
```

Figure 10-16 illustrates how conditional I/O can be used to poll several files for inputs.

```
program pollfiles(file1, file2, file3, file4: file of integer);
  var i: integer;
  begin {pollfiles}
  f$conditional(file1, true);
  f$conditional(file2, true);
  f$conditional(file3, true);
  reset(file1);
  reset(file2);
  reset(file3);
  rewrite(file4);

  while true do begin {poll each file}
    read(file1, i) {will not wait for component};
      if f$lastsuccessful(file1) then write(file4, i);
    read(file2, i) {will not wait for component};
      if f$lastsuccessful(file2) then write(file4, i);
    read(file3, i) {will not wait for component};
      if f$lastsuccessful(file3) then write(file4, i);
    end {while true do begin}
  end {pollfiles}.
```

FIGURE 10-16. POLLING FILES FOR INPUT

The conditional attribute does NOT ensure that a process proceeds without suspension during I/O operations. It is necessary for the Executive RTS to schedule access to shared channel control data structures. Therefore an I/O operation may cause suspension until the necessary structures are accessible by the calling process. This, however, should be a relatively short suspension.

As discussed previously, the standard function EOF normally causes a component from the channel to be received into the look-ahead buffer to ensure that one is available. However, if the file has the conditional attribute and EOF is called, the next component is received only if it has been produced. Otherwise, the look-ahead buffer is left empty and the result of EOF is FALSE, indicating that the end of transmission has not occurred on the associated channel. Therefore, for conditional files, EOF = FALSE indicates only that another READ attempt may be made without causing an exception, and EOF = TRUE indicates that another read attempt will unconditionally cause an exception.

Since EOF may cause a channel transfer, the function F$LASTSUCCESSFUL should be called after every EOF to indicate the success of the channel transfer to the look-ahead buffer. When EOF = FALSE and F$LASTSUCCESSFUL = TRUE, the READ(F,I) is guaranteed to be successful, as indicated by the ASSERT after the READ. Figure 10-17 illustrates the use of EOF to detect end of transmission on the channel associated with a conditional file.

```
program server(f: file of integer);
  var i: integer;
  begin {server}
  f$conditional(f, true) {establish F to be conditional};
  reset(f);
  while not eof(f) do begin {eof may be unsuccessful}
    if f$lastsuccessful(f) then begin .component in buffer
      read(f, i); assert f$lastsuccessful(f);
      "process i"
      end {if f$lastsuccessful(f) then ...};
    end {while not eof(f) do ...};
  end {server};
```

FIGURE 10-17. USE OF EOF WITH CONDITIONAL FILES

10.2.13 Channel Abortions

The Executive RTS provides the capability for a user to abort channels. This normally causes all file variables connected to the channel being aborted to become disconnected. Any subsequent READs or WRITEs of a disconnected file variable results in an exception until the file is opened again. Any files suspended on the channel being aborted are activated with an exception. The Executive RTS routine used to abort channels is

```
procedure f$chabort(var f: anyfile); external;
  .read as "channel abort"
```

which aborts all channels having the same name as F. The standard procedure SETNAME can be used to set the name of F to the name of the channel to be aborted. The file F does not have to be connected to the channel.

Channel abortions may be used to cancel I/O on a device by aborting the device channel. A device channel is not destroyed when aborted so that the device has the capability of restarting.

SECTION XI

PROCESS MANAGEMENT

Microprocessor Pascal System has the features of conventional Pascal plus more; several of the extended constructs were added to support concurrent processes. This section describes how concurrent processes are declared, invoked, and terminated.

11.1 SYSTEM DECLARATION

Microprocessor Pascal System defines the SYSTEM as the language construct within which all other constructs are nested; the SYSTEM is the outermost level of declarations and executable statements. Declarations at the system level are considered to be at lexical level zero; program declarations nested within the SYSTEM are at lexical level one; process declarations are nested within programs or other processes, starting at lexical level two.

The statements of the system body are executed before any other statements in a program or process. A process, called the system process, is created by the Executive RTS which executes the system body and terminates when it reaches the END statement of the system body. Properties of this process can be controlled with the concurrent characteristics clause of the system body definition. These properties are the PRIORITY, STACKSIZE, and HEAPSIZE of the system process.

The system body is considered to be a bootstrap program which is executed in a Pascal environment. Within the system body, COMMONs can be initialized and programs can be started to cause a system of several programs to begin execution concurrently. Typically each peripheral device is serviced by a program which is passed parameters that characterize the device. Figure 11-1 below illustrates how a system of two CRT devices and one copy of a main program are started from the system body.

```
System example;

  type
    alfa = packed array 1..8  of char;

  Program CRT( cru_base, interrupt_level: integer;
    device_name: alfa );
    begin
      { ...}
    end { CRT } ;

  Program main;
    begin
      { ...}
    end { main} ;

begin { example}
{# priority = 1; stacksize = 200; heapsize = 0}
  start CRT( #0C0, 3, 'CRT01    ' );
  start CRT( #0E0, 4, 'CRT02    ' );
  start main;
end { example} .
```

Figure 11-1. Example of a System Body which Starts Two Devices and a Main Program

Notice in Figure 11-1 that the CRT program is statically declared once but is dynamically invoked twice. Two unique copies of the CRT program are in concurrent execution. Also, since the system process does no NEW or DISPOSE operations, it needs no heap, and the HEAPSIZE characteristic of the system process is zero.

Another concurrent characteristic in Figure 11-1 sets the priority of the system process to one. The interrupt mask of the system process is zero, because the mask of any process is always the greater of zero and one less than its priority. Having an interrupt mask of zero means that no interrupts are recognized by the processor until the mask is lowered to some value between one and fifteen. This happens when some process of lower urgency executes. When the environment of the system process is created by the Executive RTS before the system process begins, the environment initialization executes a RSET instruction. (This instruction is in the USERINIT module; refer to Section 17.3.) This hardware instruction resets directly connected peripheral devices and those CRU devices that provide for reset in the interface with the CRU. However some devices are not reset, such as the TMS9902 asynchronous communication controller and the TMS9903 synchronous communication controller which are reset by software, or a perpetual clock which generates interrupts forever. Since all devices may not be reset by the environment initialization, and servicing interrupts from uninitialized devices could produce errors, the execution of the system process with all interrupts masked allows it to call routines to reset and initialize devices. The recommended technique of initialization and bootstrap by the system process is to set the priority of the system process to one and to call procedures from the system body which reset and initialize each peripheral device. These procedures are referred to as physical device interface initialization procedures in Section 14.2 on Physical Device Interface Systems.

The declarations in the SYSTEM include mainly programs but may also include routines (procedures and functions) which are globally available to the entire system (such as services provided by the run-time support). Constants declared at the system level could correspond to global values such as special characters (e.g. const line_feed = #0A;) or CRU addresses (e.g. const front_panel_lights = #1FE0;). Types declared at the system level could define run-time support data structures. Commons might correspond to interrupt and XOP transfer vectors or other global, fixed-address words in memory space. Programs must be declared at the system level, and this allows program identifiers to be within scope of any executable statement. A program or routine declared at the system level may have its body replaced by EXTERNAL or FORWARD to support separate compilation. Figure 11-2 illustrates declarations in a system.

```
System example;

   const { constants global to the entire system}
     line_feed = #0A;
     front_panel_lights = #1FE0;

   type { data structures implemented by
          the run-time support}
     processid = @ processid;
     alfa = packed array 1..8  of char;

   common { data structures at fixed addresses }
            in memory space}
     interrupt_trap_vectors: array 0..15  of
       record wp, pc: integer end;

   Program CRT( cru_base, interrupt_level: integer;
     device_name: alfa );
     external;

   Program main;
     begin
       { ...}
     end { main} ;

   function my$process: processid;
     external { provided by Executive RTS} ;

begin { example}
{# priority = 1; stacksize = 200; heapsize = 0}
  start CRT( #0C0, 3, 'CRT01   ' );
  start CRT( #0E0, 4, 'CRT02   ' );
  start main;
end { example } .
```

Figure 11-2. Example of System Body Declarations

There is no VAR section in the SYSTEM block, so no variables exist in the scope of the executable statements of the system body or in the scope of programs declared at level one.

11.2 PROGRAM DECLARATION

Microprocessor Pascal System allows more than one program to be declared in the SYSTEM construct. This is different than conventional Pascal which is oriented to one program that has one site of execution and one set of local data in one stack. A program is a natural encapsulation of an algorithm which is mostly independent of other concurrent activity in the same processing system. The fact that the Microprocessor Pascal System allows for multitasking causes the need for multiple programs in the same system.

A program is declared at lexical level one within the SYSTEM construct. There are no variables global to a program (no variables at the SYSTEM level), and program paUPmeters must be passed by value and must not be referenced by pointers. This means a program references no data space external to itself (except possibly for COMMONs which must be explicitly named in ACCESS statements). This data space address restriction better enables the Executive RTS to manage the resources of a program in an environment where there is more memory than the logical address space of a TI 990/9900 processor (65,536 bytes). A 16-bit memory reference in this environment is mapped to a unique physical memory location by hardware, and this capability is called memory mapping. The current version of the Executive RTS does not support this environment, but future systems will assign each program a unique logical address space that can be temporarily extended by the ACCESS statement to allow addressing a COMMON. Thus, the fact that a program limits data space addressability is consistent with the bounds of a logical address space enforced by memory mapping.

A program, like a process, has a site of execution, a machine context, and local data in a stack. In this sense, then, a program is no different than a process in that both independently execute with other concurrent activities. Except for the facts that a program must be declared at lexical level one and addresses no data space outside itself (except for commons), a program does not differ in capability from a process.

NOTE

> The word process will be used many times in this document in a context applicable to a system, program, or process. Both a program and a system are a special case of a process and do not differ in essential capability. When a distinction among system, program, or process is required, the distinction will be made clear.

The fact that a program shares no data space with other concurrent programs (except for commons) does not prohibit a program or a nested process from receiving data from another program or process in another data space. Data can be transmitted through interprocess files among processes (or programs) that do not share data space. This feature is used in building programs which service peripheral devices. The program construct is a convenient encapsulation of device handling algorithms which operate independently of other activities, receive or transmit data through interprocess files, and do not share data space with other programs. Section 10.2 gives more information on interprocess files, and Section 14 explains the construction of device handlers.

11.3 DECLARATION OF A CONVENTIONAL PASCAL PROGRAM

A conventional Pascal program, such as:

```
Program user;
{ declarations}
begin
  { executable statements}
end.
```

Figure 11-3. SIMPLE, CONVENTIONAL PASCAL PROGRAM

```
SYSTEM DUMMY;

  Program user;
  { declarations}
  begin
    { executable statements}
  end;

BEGIN
  START USER;
END.
```

Figure 11-4. EQUIVALENT of CONVENTIONAL PASCAL PROGRAM

is compiled without being nested within a SYSTEM body. The program cannot have nested PROCESS declarations and can neither declare program parameters nor declare the standard text files of INPUT and OUTPUT. The conventional Pascal program of Figure 11-3 is considered logically equivalent to that shown in Figure 11-4. That is, the user's conventional program, when placed in execution by the Executive RTS, is considered to be nested within a default system body which STARTs the user's program.

The standard files INPUT and OUTPUT may be referenced within a conventional Pascal program but must not be declared since they are predefined for the program (reference Section 8.7.2). The following shows a program which uses OUTPUT and is compiled as a conventional program:

```
Program user;
{ declarations}
begin
  writeln( 'START EXECUTION' );  { write to OUTPUT file}
  { other executable statements}
end.
```

Figure 11-5. CONVENTIONAL PASCAL PROGRAM WITH FILE I/O

```
  begin
    { initialize peripheral devices
      and start supporting processes}
  end;

  program user( output: text );
  { declarations}
  begin
    writeln( 'START EXECUTION' );  { write to OUTPUT file}
    { other executable statements}
  end;

begin
  init_system;
  start user( filenamed( 'PRINTER' ) );
end.
```

Figure 11-6. EQUIVALENT SYSTEM TO CONVENTIONAL PASCAL PROGRAM WITH FILE I/O

If the above program is the only user-written code present which executes on a processor, then no peripheral devices (physical devices) will have been initialized or enabled, and no other processes will have been started before the WRITELN statement. Therefore, the destination of the write to the OUTPUT file is not defined. A conventional program which uses files (including INPUT or OUTPUT) should never start file operations until peripheral devices have been initialized or other processes have been started.

If a program is compiled within a SYSTEM body and if it references the standard text files INPUT and/or OUTPUT, then they must be declared as program parameters. If the program in Figure 11-5 above is compiled within a SYSTEM, then it must declare OUTPUT since it references the file, and the SYSTEM code must pass the program parameter at the START of the program.

Notice that the procedure INIT_SYSTEM is called to initialize peripheral devices and start supporting processes before program USER is started. When USER executes the WRITELN statement, its output goes to the PRINTER channel instead of being lost.

In order to establish the environment for input/output of a conventional program using file operations, it is recommended that the source text of the program be transformed according to the following rules:

1. Add declarations for the standard files INPUT and OUTPUT as program parameters, for example

   ```
   CHANGE  program prog;
       TO  program prog( input, output: text );
   ```

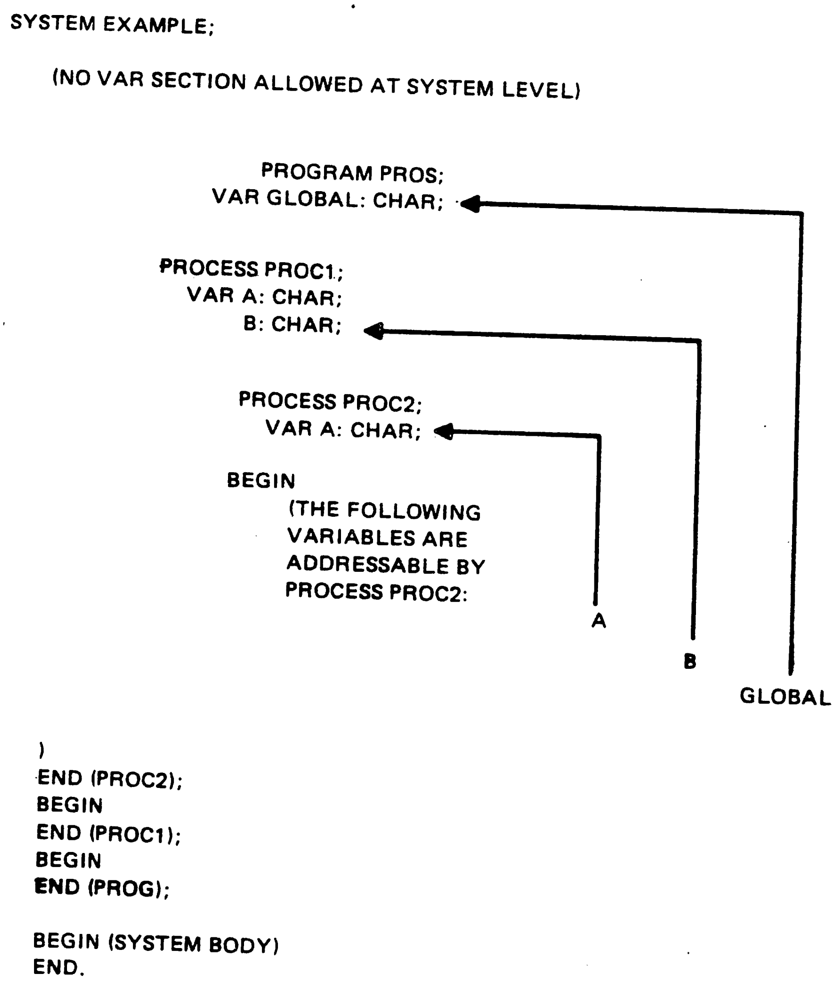
2. Compile the source text of the program as nested within a SYSTEM. The COPY statement (reference Section 5.4) in the declarations of the SYSTEM may be a convenient way to include the text of the program within the SYSTEM.

3. Add code to the SYSTEM body which causes peripheral devices to be initialized or supporting processes to be started.

4. Add a START statement which starts the program and passes its input and output files.

11.4 PROCESS DECLARATION

A process is declared within a program or another process starting at lexical level two. Scope rules of Pascal cause variables global to a process to be addressable by the process even if the variables are local to a program or process which is a lexical ancestor. Figure 11-7 illustrates the nesting of processes and their variables in scope.



```
SYSTEM EXAMPLE;

   (NO VAR SECTION ALLOWED AT SYSTEM LEVEL)

                  PROGRAM PROS;
               VAR GLOBAL: CHAR;

            PROCESS PROC1;
              VAR A: CHAR;
                  B: CHAR;

                 PROCESS PROC2;
                   VAR A: CHAR;

                BEGIN
                     (THE FOLLOWING
                     VARIABLES ARE
                     ADDRESSABLE BY
                     PROCESS PROC2:

 )
 END (PROC2);
 BEGIN
 END (PROC1);
 BEGIN
 END (PROG);

 BEGIN (SYSTEM BODY)
 END.
```

Figure 11-7. Nesting of Processes and Variables in Scope

A process declaration may not be nested within a procedure or function. This avoids a serious problem of addressability of variables: once the process is started executing, the variables which are local to the procedure or function would always be addressable to the process according to scope rules but would have existence only when the procedure or function is active. The extent of a computational quantity is the time during execution that the quantity may be considered to exist. The extent of the local variables of the procedure of function is shorter than the extent of the process. Passing parameters to a process by reference is likewise prohibited, because the extent of the parameters passed is independent of the extent of the process to which the parameters are passed.

Once a process (or program) begins execution by means of the START statement, its extent is independent of other processes. When a process terminates, the Executive RTS causes its process local variables to remain until all lexical descendants of the terminated process finish. Thus the Executive RTS ensures the extent of process local variables is at least as long as the extent of lexically descendent processes which can address the variables by scope rules. (Since a program has a self-contained data space, no other process must wait on a program to terminate.) In the case illustrated in Figure 11-7 above, the Executive RTS ensures that variables a and b (local variables of process proc1) exist at least until process proc2 terminates, and global exists at least until both proc1 and proc2 terminate. All variables within scope are guaranteed to exist by the Executive RTS.

A program and all its nested processes share data space addressability; on a processor with memory mapping, the data space of a program and all its nested processes are in the same logical address space. A program and its processes may share data space by means of variables in a stack (achieved by nesting of process declarations) or by means of variables in heap (achieved by sharing of pointers which reference the heap data). Process synchronization as explained in Section 9.3 on semaphores may be required to coordinate the sharing of data among concurrent processes.

11.5 CONCURRENT CHARACTERISTICS

Each system, program, and process has three concurrent characteristics: PRIORITY, STACKSIZE, and HEAPSIZE. These determine the execution and Pascal environment requirements of the corresponding block.

The PRIORITY characteristic is related to the urgency of a process relative to the urgencies of other processes. Priority is a non-negative number ranging from zero to 32766 where zero indicates the greatest urgency, and the range zero to fifteen is reserved for device processes. Concurrent processes which are ready (not blocked) compete to be assigned to a processor. The scheduling policy keeps the most urgent ready process assigned to the processor, such that no other more urgent process is ready but not given a processor. (On future systems which support more than one processor, each processor is assigned to a process which is not less urgent than any other ready process.) For example, a ready device process (priority from 0 to 15) always preempts any ready process of priority 16 or greater. On the other hand, a single process of priority 32766 never executes until all other processes have terminated or are blocked.

The priority of a process determines its interrupt mask which enables interrupts only of greater urgency than the process's priority. This means that non-device processes (with priority between 16 and 32766, inclusive) execute with an interrupt mask of 15. Device processes (with priority between zero and 15, inclusive) execute with an interrupt mask which is the greater of zero and the process's priority minus one.

The default priority of the system process is one. If no priority is stated for a program, then its default is 32766, which may not be adequate for the program to be responsive to real-time requirements. If no priority is stated for a process, then it inherits the priority of its lexical parent (which is either a process or a program). A negative PRIORITY or a PRIORITY of 32767 is considered illegal and causes an exception for the process attempting a START of a new process with an illegal PRIORITY.

The STACKSIZE concurrent characteristic is the number of words of storage which the process intends to use for its local variables and the variables associated with all subsequent dynamic routine calls. Space for this stack is contiguously allocated from the heap of the lexical parent.

If the user states a small stacksize or even zero, then a stack just big enough to execute the system, program, or process is allocated (holding its parameters and local variables) plus room to execute process termination.

A non-zero value of the HEAPSIZE concurrent characteristic indicates to the Executive RTS that the declared process requires a private heap upon which its heap requests NEW and DISPOSE operate. This private heap is nested within the heap of the declared process's lexical parent. This new nested heap is created by allocating a contiguous area of memory from the parent's heap and initializing it by the Executive RTS to be a heap structure. The non-zero HEAPSIZE parameter is the number of words of storage which the process intends to use in a nested heap. If HEAPSIZE is zero, then no nested heap is created, and the parent's heap is used as the child's heap.

A process references one and only one heap using NEW and DISPOSE. This heap (whether nested or the parent's) must be large enough to include all subsequent heap allocations of data, other heaps, and stacks of nested processes which are started. The default for an unstated HEAPSIZE is zero which is interpreted to mean use the parent's heap.

A process (or a program) may declare its concurrent characteristics to be constant or to be the value of a process parameter which is passed when the process is first started. Figure 11-8 below illustrates that program CRT has a STACKSIZE and HEAPSIZE which are constant and a PRIORITY which is passed as one of its parameters.

```
type
  alfa = packed array [1..8] of char;

Program CRT( cru_base, interrupt_level: integer;
  device_name: alfa );
  begin
  {# priority = interrupt_level;
     stacksize = 300; heapsize = 120}

    { executable statements}

  end { CRT };
```

Figure 11-8. Example of Concurrent Characteristics Which are Constant and Variable

Section 12.5 gives much more information on how to select the proper STACKSIZE and HEAPSIZE for a process.

11.6 PROCESS INVOCATION

A program or process is initially invoked with the START statement in which the program or process identifier is named (the identifier must be in scope) and process parameters are passed, if they exist, much like a procedure call. However, in a procedure call, the caller is resumed after the called procedure exits; in a process call, the called process continues execution concurrently with the calling process. A program or process is statically declared once but may be dynamically invoked more than once as required. Figure 11-9 below illustrates a program and two nested processes.

```
System example;

  Program prog;
    var global: char;

    Process proc1;
      var a: char;
        b: char;

      Process proc2;
      begin
      end { proc2} ;

    begin
      start proc2;
    end { proc1} ;

  begin
    start proc1;
    start proc1;
  end { prog} ;

begin
  start prog;
end { example} .
```

Figure 11-9. Multiple Dynamic Invocations of Processes

Notice in Figure 11-9 that the system process starts one copy of program PROG, PROG starts two copies of process PROC1, and each copy of PROC1 starts one copy of process PROC2. The single copy of PROG causes one instance of the variable GLOBAL to exist. Each of the two instances of PROC1 has a unique set of variables A and B. One instance of process PROC1 cannot address the A and B variables belonging to the other instance of process PROC1; each instance of PROC1 can address its own local copy of variables A and B. The variables of the first instance of process PROC1 are addressable by the single instance of process PROC2 which it starts. The same is true for the second instance of PROC1 and its single instance of PROC2 which it starts.

The START statement is not the only way to refer to a process by name. The Executive RTS maintains a process identification which is the dynamic "name" of a process and is assigned by the Executive RTS for each unique instance of a process. A process identification is declared by the user as follows:

```
type processid = @ processid
                 { i.e. a pointer to something} ;
```

The type PROCESSID is a pointer to an undefined data structure (undefined to the user) which is implemented by the Executive RTS. The following function:

```
    function my$process: processid; external;
```

returns the process identification of the calling process; it essentially answers the question, "Who am I?". Another useful function:

```
    function p$lastprocess( p: processid ): processid; external;
```

returns the identification of the last process successfully started by process P or returns NIL if the last attempted start of a process by P was unsuccessful. The initial value which this function would return is NIL. Therefore the following call:

```
    p$lastprocess( my$process );
```

returns the identification of the last process successfully started by the calling process. Once process identifications have been captured by using the MY$PROCESS or P$LASTPROCESS routines, the process identifications may be passed to other Executive RTS routines which manage processes, such as process abort.

The following function

```
    function p$successful( p: processid ): boolean; external;
```

returns a boolean status to indicate the success of the last process management operation done by process P. The initial status is FALSE. After a START statement the result of

```
    p$successful( my$process );
```

is TRUE or FALSE to indicate that a process was or was not created respectively. Process creation may fail if the resources of the candidate process could not be acquired, or if the priority of the candidate process was illegal.

The following procedure

```
        start$term( var oldvalue: boolean; newvalue: boolean);
```

is used to control the mode of exception handling when processes cannot be successfully started. If the START$TERM flag is TRUE when an unsuccessful START is encountered by a process, then the process which called START fails. If the START$TERM flag is FALSE, then a process which calls START and does not successfully start a process does not fail. The unsuccessful START is ignored. The default value of the START$TERM flag is TRUE for the system or a program, and a started process inherits the START$TERM flag of its lexical parent.

11.7 PROCESS TERMINATION

A process terminates normally when its execution reaches the END which closes the declaration of the process body. An ESCAPE statement which references the current process's statically declared identifier also causes the process to terminate normally.

A process can be involuntarily terminated by another process by means of the following procedure of the Executive RTS:

```
    procedure p$abort( p: processid ); external;
```

After calling P$ABORT(P) process P is marked to be aborted and is aborted when the process is again active (not suspended on a semaphore), has returned from all nested Executive RTS calls (e.g. has returned from a heap request), and is not nested within user-defined critical transactions of code. The Executive RTS causes process P to encounter an ABORTED exception which is considered abnormal. Section 13 gives more information on process aborting, user-defined critical transactions, and exception handling.

The following

```
    p$abort( my$process );
```

causes the calling process to terminate and is a useful technique for a routine, which is shared by more than one process, to cause the calling process to terminate. This is more general than an ESCAPE statement since an ESCAPE must specify a lexical parent, and this cannot be used in a routine that may be called from an arbitrary process. However, P$ABORT causes an abnormal exception and ESCAPE causes a normal termination.

When a process terminates, its resources (such as its stack, heap, and administration areas) are reclaimed by the Executive RTS. The local variables of a terminated process or program are retained until no lexical descendent processes exist, then the local variables are no longer addressable and are destroyed.

SECTION XII

MEMORY MANAGEMENT

This section describes the memory management features of the Executive RTS and shows how concurrent characteristics should be chosen for a process to satisfy stack and heap requirements.

12.1 MEMORY MANAGEMENT BY THE EXECUTIVE RTS

12.1.1 Dynamically Located Data Areas

System memory is the term used to describe all the programmable memory (RAM) available to the Executive RTS. All data space is managed by the Executive RTS in areas dynamically allocated from system memory. The only exception to this is that the static location of data areas on a target processor such as CSEGs (or COMMONs), DSEGs, and interrupt and XOP workspaces are chosen by the user. These static data areas are not part of system memory and are not directly manipulated by the Executive RTS.

System memory for the Host Debugger is determined from the user's reply to the "SYSTEM HEAP IN KBYTES" prompt. One contiguous area is created which is as large as the user's request. System memory on a target processor is specified by the user in a RAM configuration table in the "CONFIG" module explained in Section 17.2.1 covering the specification of RAM locations. Disjoint contiguous areas of RAM are allowed in the target processor.

Data space in programmable memory (RAM) that is available to a process is managed by a heap structure. Abstractly, a heap is an area of allocated (used) space and an area of unallocated (free) space. At any time (under program control), free areas may become allocated or allocated areas may be freed. Freed areas may be reused (reallocated).

Heaps are one of two types: program or nested. Program heaps are the heaps of programs or the system process, and are returned to system memory when the program and all its nested processes terminate or when the system process terminates. When a process begins which has a non-zero HEAPSIZE characteristic, nested heaps are created within another heap, called the parent. These are returned to the parent when the process which causes the nested heap to be created terminates. are returned to the parent when the process terminates which caused the nested heap to be created. Packets allocated out of a nested heap remain allocated even if the process terminates which caused the nested heap to be created. The packets which were in the nested heap are considered to be in the parent's heap and remain addressable.

In future versions of the Executive RTS which support memory mapping, the data space of any process is restricted to the program heap in which that process's data resides. Since all processes lexically nested within a program either use the program heap or a heap nested within the program heap, then a program heap is a logical division of data space for memory mapping. The Microprocessor Pascal System language enforces that no references or pointers may be passed outside of a program, so no referencing across map space boundaries is allowed. The data space may be temporarily extended within a routine which has an ACCESS statement of a COMMON. But the preferable technique to pass data among processes that reside in separate programs and data spaces is through the interprocess file mechanism provided by the Executive RTS.

12.1.2 Statically Located Data Areas

Statically allocated data areas on a target processor are not located by the Executive RTS. A COMMON declared in Microprocessor Pascal causes the Microprocessor Pascal compiler to generate a CSEG with a name which is the first six characters of the COMMON name. The user may use CSEGs and DSEGs as needed in assembly language modules and may need to control the static placement of these areas in memory space. The link editor PROGRAM, COMMON, and DATA commands or the use of modules with BSS directives to place succeeding modules at specific addresses allow the user to position these data areas. Interrupt and XOP trap workspaces are also located by the user anywhere he chooses.

12.2 HIGH-LEVEL USER INTERFACE TO MEMORY MANAGEMENT

The basic operations on heaps are allocation and deallocation done by the standard procedures NEW and DISPOSE, respectively. These routines manipulate packets of the process's heap.

12.2.1 Procedure NEW

```
new(ptr) .PTR is a pointer to some type
```

This routine returns a pointer in PTR to a new, allocated heap packet. This routine is pre-declared by the compiler.

12.2.2 Procedure DISPOSE

```
dispose(ptr) .PTR must be a pointer to a heap packet
```

This routine returns a heap packet referenced by PTR to the free area for reuse. The value of PTR is returned as NIL. This routine is pre-declared by the compiler.

12.3 LOW-LEVEL USER INTERFACE TO MEMORY MANAGEMENT

NEW$ and FREE$ implement the standard procedures NEW and DISPOSE, respectively, and can be called for unusual memory allocation requiring variable-size packets. The declarations to use these features are as follows:

```
     type pointer = @some_type;   " pointer to some type    .

          byte_length = 0..32767;

     procedure new$( var ptr: pointer; length: byte_length );
          external;

     procedure free$( var ptr: pointer ); external;

     procedure heap$term( var oldvalue: boolean; newvalue:
          boolean ); external;
```

12.3.1 Procedure NEW$

```
     procedure new$(var ptr: pointer; length: byte_length);
       external;
```

This procedure allocates a contiguous area from the current process's heap of LENGTH or more bytes and returns a pointer to the area in PTR.

12.3.2 Procedure FREE$

```
     procedure free$(var ptr: pointer); external;
```

This routine returns an allocated area to the free area for reuse. PTR is a pointer to the area to be freed and is set to NIL, and the packet is returned to the heap from which it was allocated.

12.3.3 Procedure HEAP$TERM

```
     procedure heap$term(var oldvalue: boolean; newvalue:
          boolean); external;
```

This routine allows user control over heap overflow, which occurs when available space cannot satisfy a request. The routine manipulates a process-local flag maintained by the Executive RTS that indicates whether heap exhaustion should cause error termination. The old flag is returned in OLDVALUE and NEWVALUE is the replaced value of the flag. If the flag is TRUE, a heap overflow causes fatal error termination for the process calling NEW or NEW$. If the flag is FALSE, no error occurs and NIL is returned as the value of pointers returned by NEW and NEW$. The default value of the HEAP$TERM flag is TRUE for the system or a program, and a started process inherits the HEAP$TERM flag of its lexical parent.

12.4 USE OF COMMONS

The Microprocessor Pascal System language enforces that no references or pointers may be passed outside of a program, and the Executive RTS keeps all data of a program and its lexically nested processes contained within the program heap. This allows the Executive RTS to dispose a program heap when the program and all lexically nested processes have terminated. The data space may be temporarily extended within a routine which has an ACCESS statement of a COMMON. However if a COMMON has a pointer, the user should be aware of the following problem. If the pointer in the COMMON is assigned a value by passing it to NEW or NEW$, then the heap packet referenced by the COMMON pointer is allocated from the heap of the process calling NEW or NEW$. This heap is a nested heap or the program heap of the calling process. When the calling process's parent program or the calling program itself terminates and all its nested processes terminate, the program heap and all the data contained therein are disposed. Therefore, the pointer in the COMMON still has a value but it references an area of data space which may be allocated again by the Executive RTS. This may potentially cause errors whenever the COMMON pointer is used, and its data structure is perturbed.

There are two ways to solve this problem. The first is to avoid the use of pointers in a COMMON and to store data structures directly in the COMMON rather than to use the indirect reference of the pointer. The other technique is to write a program (or a nested process) which dynamically allocates the data structures referenced by pointers in a COMMON and then not allow the program (or process) to terminate. Thus its program heap is never disposed, and the data structures referenced in the COMMON are forever allocated. The following routine is an example of one that the user may write which suspends the calling process forever:

```
procedure suspend;
  var
    forever: semaphore;
  begin
    initsemaphore(forever, 0);
    wait(forever);
  end;
```

12.5 PROCESS RESOURCES

The data structures created by the Executive RTS when a process is STARTed are in two classes: the process's stack and the process's heap. The following discussion applies also to a program or the system process.

12.5.1 Process Stack

A process has process parameters passed to it and global variables declared in the process module. These are stored along with a 30 byte administration area in one heap packet referred to as the process global frame. This packet is disposed when the process has terminated and all its lexically nested processes (if any) have terminated.

The process stack is allocated as another heap packet and is used to hold instantiations of routines which the process calls. The heap packet holding the process stack is disposed when the process terminates.

Each instantiation of a routine called within a process requires space in the process stack (called a stack frame) for the parameters passed to it and its local variables plus a 14 byte administration area which precedes its parameters and locals. The MAP option of the compiler produces a listing in which the STACK SIZE value is given for each routine. The stack requirements of a process can be determined by the user by summing stack frame sizes for the most deeply nested set of dynamic routine calls. When Executive RTS services are called, the user should consider that STARTing a process requires about 250 words of stack and file operations require about 250 words of stack.

When a system process, program, or process is STARTed, the STACKSIZE concurrent characteristic is used by the Executive RTS to allocate the stack for the new process. If STACKSIZE is not stated, is zero, or is a small number, then enough is allocated for the process's global frame to hold its parameters and global variables and the process stack is large enough for the process to execute the process termination code in the Executive RTS. This may not be enough space, however, for the process to execute the algorithms that the user has coded without encountering a stack overflow, which is a fatal error for the process. For a larger value of STACKSIZE, then STACKSIZE words is the sum of the words allocated for the process's parameters and global variables (plus 30 bytes in one heap packet for the process global frame) and the process stack. The latter is contained in one heap packet with 150 bytes of administration area added at the end. (The new process's process record is located at the end of the heap packet of the process stack.) A negative STACKSIZE is considered an error and causes an exception for the process attempting a START of a new process with a negative STACKSIZE.

12.5.2 Process Heap

The system process and any program are always given a new program heap which is allocated as one contiguous packet from system memory, even if the HEAPSIZE concurrent characteristic of the system or program is zero or not stated. If a HEAPSIZE is given which is greater than zero, then the new program heap is large enough for a packet of HEAPSIZE words to be allocated by the user by a call to NEW or NEW$. (Smaller packets may also be allocated up to HEAPSIZE words, but the "checkboarding" of available space must be considered.) For an unstated or zero HEAPSIZE, no available space is left in the new program heap after the system process or program is created.

When a process (lexical level of two or greater) is STARTed, it inherits the heap of its lexical parent if its HEAPSIZE concurrent characteristic is zero or not stated. That is, NEW or NEW$ called within the new process acquires packets from a program heap or a parent's nested heap which is nested in a program heap (or nested in another nested heap which is nested in a program heap, etc.). If the HEAPSIZE concurrent characteristic of a new process (lexical level of two or greater) is greater than zero, then the new process uses a new nested heap which is nested within its parent's heap. The nested heap appears to the parent's heap as one contiguous, allocated packet but appears to the nested process as a heap structure. Calls to NEW or NEW$ by the process acquire packets from the nested heap of the process.

The creation of heaps by the Executive RTS allows for the localization of the data space required by processes. The system process and each program are always given a new heap structure from which it allocates structures that are local to the system process or program. A process (lexical level of two or greater) may be restricted to a maximum heap usage by STARTing it with a new nested heap local to the process, or a process may be allowed to share the heap of its parent.

A new heap structure is allocated by the Executive RTS as one contiguous packet from a parent heap. The heap structure requires 26 bytes of administration areas plus HEAPSIZE words where an unstated HEAPSIZE is considered to be zero. A negative HEAPSIZE is considered an error and causes an exception for the process attempting a START of a new process with a negative HEAPSIZE.

12.5.3 Estimating Space Requirements of Process Resources

Since the resources of processes are dynamically allocated by the Executive RTS, the space requirements for a system of several processes is difficult to state precisely. When a system is initially debugged, it is wise to allocate more space for process stacks and process heaps than necessary in order for the system to execute without exhausting data space. After processes have executed and been debugged, the user may reduce STACKSIZE and HEAPSIZE parameters of processes.

The Host Debugger may be used to determine the stack requirements of a process. The process should have executed all paths of dynamic calling sequences to force it to use as much stack as possible. After the user is certain that a process has executed its most deeply nested set of routine calls, the DP (Display Process) command indicates how much stack has been used. Figure 12-1 gives an example of the DP command.

The process being debugged in Figure 12-1 was started with a STACKSIZE of 200 words. Notice that the maximum stack usage of 96 bytes and the current stack usage of 96 bytes are shown. This process could have been started with a STACKSIZE of 48 words without encountering a stack overflow exception.

The heap requirements of a process can also be determined with the Host Debugger. A process should have executed all NEW and NEW$ calls to allocate as much heap space as it needs. Then use the SDP command (Select Default process) followed by the SH command with no parameters (Show Heap). Figure 12-2 on the following page shows an example of this.

```
  DP(PROG)
Static/Dynamic Calling Order for Process PROG(2)

     Stack Size (bytes) = 400
     Stack Used (bytes) Maximum = 96  Current = 96

     Call Order          Name                 Statement
          1            PROG                       1
          2              A                        1
          3              B                        1
          4              C                        1
          5              D                        1
```

FIGURE 12-1. DETERMINING STACK REQUIREMENTS OF A PROCESS

```
 SDP(PROG)

 SH
     HEAP AT   C000 ROVER:   C010 HPMIN:   C010 HPMAX:   C0A4
      MAXUSED: 004E CURUSED: 004E MUTEX:   BFF8 PARENT:  B4D6
 C058 (0000) 0000 0000 0000 0000                        (........        )

 C062 (0000) 0000 0000 0000 0000 0000 0000 0000 0000    (................)
 C072 (0010) 0000 0000                                  (....            )

 C078 (0000) BFEA 0000 C000 E101 7FFE 0000 0000 0000    (................)
 C088 (0010) 0003 0000 0000 0000 0000 B076 0000 0000    (...........v....)
 C098 (0020) 0000 0000 0000 0000 C062 C058              (.........b.X    )
```

FIGURE 12-2. DETERMINING HEAP REQUIREMENTS OF A PROCESS

In Figure 12-2 the MAXUSED field of the SH command is shown as hexadecimal 4E bytes. Therefore, the process can have a HEAPSIZE of decimal 39 words (corresponding to 4E bytes) without exhausting its heap space.

The Target Debugger may also be used to determine the stack and heap requirements of a process. Figure 12-3 below shows an example of the SP command (Show Process).

The "STACK USED (MAX)" and the "HEAP USED (MAX)" fields are in hexadecimal bytes and allow the user to calculate the space requirements of the process.

```
  ?
SP

SHOW PROCESS "EX  AM  PL  " AT >3EC8
STACK BASE =>3D94  STACK LIMIT =>3EC8       STACK BOUNDARY =>3E9E
STACK SIZE =>0138  STACK USED (MAX) =>010E  STACK USED (CUR) =>0100
HEAP SIZE = >0E9E  HEAP USED (MAX) = >0516  HEAP USED (CUR) = >0516
PRIORITY  =  32766
NO OUTSTANDING EXCEPTIONS
NEXT PROCESS IN LIST =>3FA6     NEXT PROCESS IN QUEUE =>0000
QUEUE POINTER =>0000
CREATORS ID =>00    MY ID =>01
```

Figure 12-3. USE OF TARGER DEBUGGER TO DETERMINE
STACK AND HEAP REQUIREMENTS OF A PROCESS

12.5.4 Allocation of Process Resources

A precise statement of the allocation rules of process resources is given below for several cases.

12.5.4.1 Allocation of System Process or New Program. A program heap is created from system memory (all of data space managed by the Executive RTS). If HEAPSIZE is greater than zero, then the program heap initially has HEAPSIZE words of available space. If HEAPSIZE is zero or not stated, the program heap has no available space after the system process or program is created. The process global frame is allocated from the program heap. The stack of the new system process or program is allocated as one heap packet from system memory.

For a new program, note that lexically nested processes which it STARTs are allocated from the program's heap. Therefore the program should have a HEAPSIZE concurrent characteristic which is large enough for instantiations of nested processes to be allocated.

12.5.4.2 Allocation of New Process. A new process at lexical level two or greater is given resources according to the following rules. If HEAPSIZE is greater than zero, then a nested heap is allocated from the parent's heap. For a new process at lexical level two, the parent's heap is the program heap. For a new process at lexical level three, the parent's heap is the nested heap of the process at lexical level two or the program heap of the program at lexical level one. Thus the parent's heap for any new process is the program heap or a nested heap of a lexical parent process. The nested heap initially has HEAPSIZE words of available space. If HEAPSIZE is zero or not stated, the new process inherits the parent's heap and uses it for all its data resources. The process global frame is allocated from the new process's heap. The stack of the new process is allocated as one heap packet from the parent's heap.

12.5.4.3 Allocation of Conventional Pascal Program. A conventional Pascal program begins with the PROGRAM construct and is written and compiled without being nested within a SYSTEM. The Microprocessor Pascal System does not allow it to have program parameters, to be nested within a SYSTEM, to have nested PROCESS declarations, or to declare the standard INPUT or OUTPUT text files.

If HEAPSIZE is zero or not stated, then the program uses system memory for allocating its heap packets. If HEAPSIZE is greater than zero, then a new program heap is created with HEAPSIZE words of available space. If STACKSIZE is zero or not stated, then a process stack is allocated as a heap packet from system memory and has 250 words of space for routine calls within the conventional program. A non-zero STACKSIZE causes a stack to be allocated from system memory with the requested size.

12.6 EXAMPLE

The following program and discussion demonstrate how to determine concurrent characteristics for programs and processes.

```
system example;

   program copier;
   var line: packed array [1..134] of char;

      procedure transfer;
      var i,j: integer; c: char;
      begin
         "code which calls other routines
      end;

      process reader;
      var card: packed array [1..80] of char;
      begin {#priority = 100; stacksize = 340; heapsize = 15}
         transfer;
      end;

      procedure produce_data;
      begin
        "code which calls other routines
      end;

      process writer;
      var image: packed array [1..134] of char;
      begin {#priority = 100; stacksize = 170; heapsize = 0}
         produce_data;
      end;

   begin {copier}
     {#priority = 16; stacksize = 320; heapsize = 720}
      start reader;
      start writer;
   end;

begin {example}
  {#priority = 1; stacksize = 250; heapsize = 0}
   start copier;
end.
```

FIGURE 12-4. PROGRAM WITH CONCURRENT CHARACTERISTICS

Process READER needs 80 bytes for its global frame (for variable CARD) plus stack space for procedure TRANSFER and the routines which it calls (not shown). The user must sum the stack space required for each instantiation of routines called from TRANSFER. For example the instantiation of TRANSFER itself requires about 6 bytes (for variables I, J, and C) plus the 14 byte administration area in the stack for the instantiation of TRANSFER. In this example we assume the user has determined that TRANSFER and the routines which it calls require about 300 words. The STACKSIZE of READER is set to 300 words plus 80 bytes for the global frame of READER or 340 words.

For this example, READER needs a nested heap of 30 bytes, so a HEAPSIZE of 15 words is requested. From this nested heap the Executive RTS allocates the global frame of READER (holding variable CARD of 80 bytes). However the HEAPSIZE of READER does not include this size since the Executive RTS ensures that the nested heap of READER has 15 words of available space after READER is created. The packet which contains the heap of READER is allocated from the parent heap of COPIER. The size of this packet for the nested heap of READER is equal to 26 bytes of administration area for the heap, a packet allocated to hold the process global frame of READER equal to 30 bytes of administration area plus 80 bytes for CARD, and the requested available space of 15 words. The total size of the nested heap of READER is 26+30+80+30 bytes or 83 words. This 83 word packet is allocated from the heap of COPIER.

The stack of READER is allocated in one heap packet. The length of this packet is equal to the requested STACKSIZE of 340 words minus the space for the parameters and variables of process READER (80 bytes for CARD). A 150 byte administration area is added to the end of the stack. Therefore the size of the packet holding the stack of READER is 680-80+150 bytes or 375 words and is allocated from the heap of COPIER.

Process WRITER needs 134 bytes for its variables (variable IMAGE) and we assume about 200 bytes for the calls of PRODUCE_DATA and the routines it calls. So the STACKSIZE of WRITER is 134+200 bytes or about 170 words. WRITER needs no heap space so its HEAPSIZE is zero. Even if WRITER did need heap space, a zero HEAPSIZE parameter causes WRITER to use the heap of its parent which is COPIER.

The space requirements of WRITER is (1) a heap packet to hold its global frame of 134 bytes (for IMAGE) plus 30 bytes (administration area) or 82 words and (2) a heap packet for its stack of 206 bytes (170 words STACKSIZE minus 134 bytes for IMAGE) plus 150 bytes (administration area) or 178 words.

Program COPIER uses 134 bytes of stack (the size of variable LINE) plus the requirements to START READER and WRITER, hence 250 words for the STARTs and about 70 words for its global variable LINE. Therefore STACKSIZE of COPIER is 320 words. COPIER needs no heap for the algorithm of the program COPIER, but it does need heap space to allocate its nested processes, READER and WRITER. An instantiation of READER needs a packet of 83 words for its nested heap and a packet of 375 words for its stack. An instantiation of WRITER needs a packet of 82 words for its global frame (No nested heap is created.) and a packet of 178 words for its stack. Therefore the HEAPSIZE of COPIER is 83+375+82+178 words or about 720 words. If additional instantiations of READER or WRITER are STARTed, then HEAPSIZE of COPIER must be increased.

The program heap of COPIER is allocated as one heap packet from system memory. The packet has a size of 26 bytes of administration area for the program heap plus the global frame of COPIER equal to 30 bytes of administration area and 134 bytes for LINE plus 720 words of HEAPSIZE, which is 26+30+134+1440 bytes or 815 words. The stack of COPIER is allocated in one packet from system memory of size equal to 320 words of STACKSIZE minus 134 bytes for LINE plus 150 bytes of administration area. This calulates to 640-134+150 bytes or 328 words.

System EXAMPLE has no global variables and STARTs COPIER. The START call requires about 250 words, so the STACKSIZE of EXAMPLE is 250 words. EXAMPLE does not call NEW or NEW$ and the resources of program COPIER are allocated from system memory and not from the heap of EXAMPLE, so the HEAPSIZE of EXAMPLE is zero. With a zero HEAPSIZE, a program heap is still created for EXAMPLE but it holds only the global frame of EXAMPLE. The packet holding its global frame is 30 bytes of administration area plus zero bytes for the global data of EXAMPLE or 15 words. The program heap is allocated as one heap packet from system memory and is equal to 26 bytes of administration area plus the 15 word packet for the global frame plus zero words of HEAPSIZE, or 28 words. The stack of EXAMPLE is allocated as one heap packet from system memory and has a size of 250 words STACKSIZE minus zero bytes for the global variables of EXAMPLE plus 150 bytes of administration area, or 325 words.

Figure 12-5 illustrates the allocation of the memory resources for EXAMPLE in Figure 12-4 on the following page.

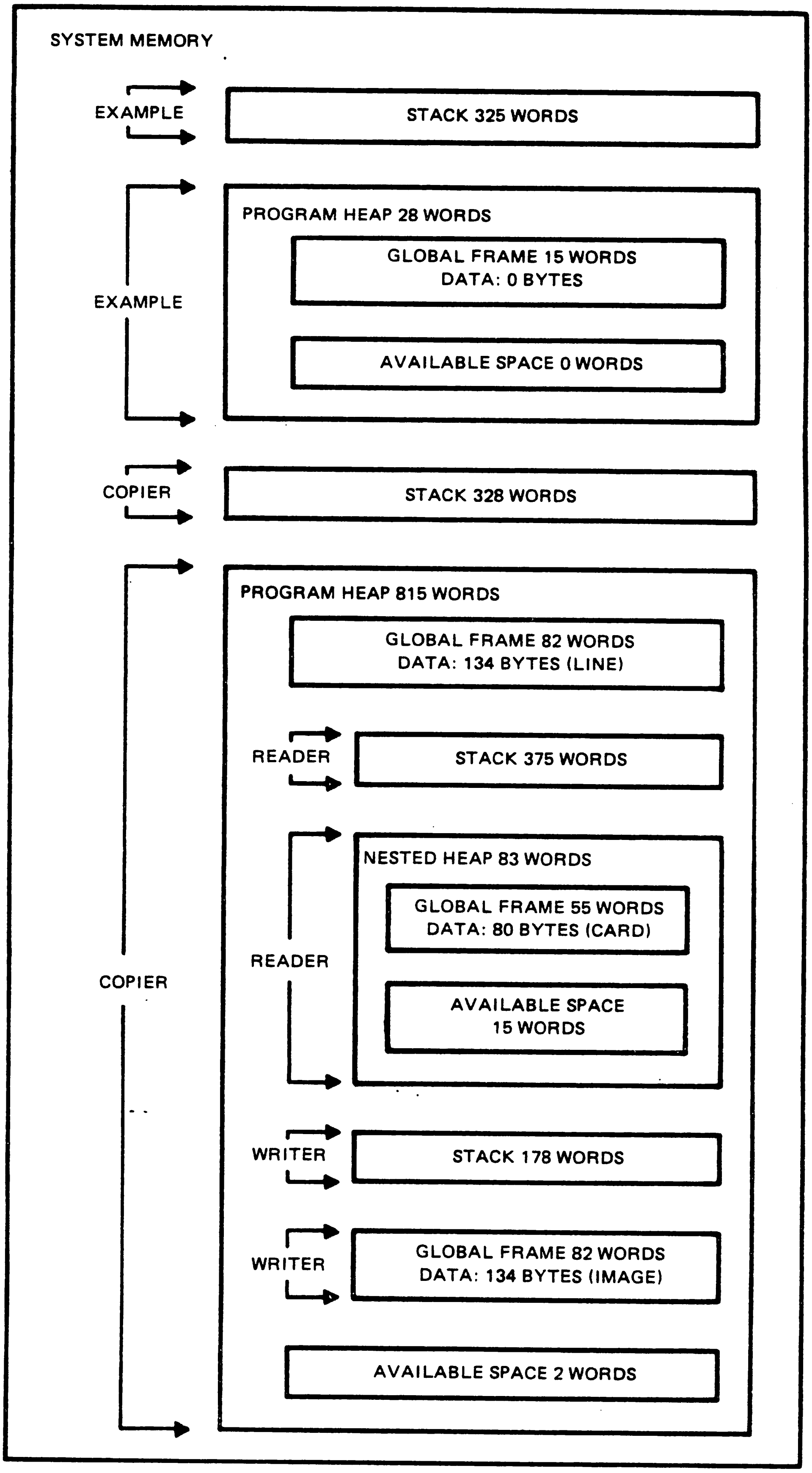


FIGURE 12-5. MEMORY LAYOUT OF STACKS AND HEAPS FOR EXAMPLE

SECTION XIII

ERROR RECOVERY AND EXCEPTION HANDLING

13.1 Introduction

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

13.2 Executive RTS Detected Errors

Errors detected by Executive RTS routines are classified according to the type of code which detected the error. Each error has a class code associated with it. Within each class code, errors are assigned a reason code. The error messages from the Host Debugger are of the following form:

Class Code	Error Message
(1)	User Error: <reason code for error>
(2)	Scheduling Error: <reason for error>
(3)	Semaphore Error: <reason for error>
(4)	Interrupt Error: <reason for error>
(5)	Process Mgmt Error: <reason for error>
(6)	Exception Error: <reason for error>
(7)	Memory Mgmt Error: <reason for error>
(8)	File Error: <reason for error>
(9)	Host File Error: <operating system error code>

Throughout the subsections of 13.2, the error reason codes are shown in parentheses to the left of each error message.

13.2.1 User Errors

A user error can be forced by calling the routine EXCEPTION as supplied in the Executive RTS Library. The process which executes this routine fails with some designated reason code (as specified by the user as a parameter to EXCEPTION).

13.2.2 Scheduling Errors

The following errors pertain to the scheduling of processes.

1) invalid queue
   This error should not be seen by the user. It indicates a system error which probably resulted from RTS code being accidently modified.

(2) 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.

13.2.3 Semaphore Errors

1) invalid semaphore
   This 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.

2) 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.

4) 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 waiters perform a wait.

13.2.4 Interrupt Errors

The following errors pertain to the handling of interrupts.

2) level invalid
   This error occurs when the priority passed to one of the routines ALTEXTERNALEVENT, EXTERNALEVENT, NOALTEXTERNALEVENT, or NOEXTERNALEVENT is not in the range 0 to 15.

3) semaphore invalid
   This error results from an attempt to use a semaphore before it has been initialized.

4) interrupt not handled
   This error occurs when an interrupt is signaled and there is no process waiting to service the interrupt.

6) handler priority error
   This error occurs when a waiting interrupt handler is less urgent than a signaled interrupt.

13.2.5 Process Management Errors

The following errors are detected in process management run-time support code.

1) not a process
   This error occurs when a run-time support routine is called which expects a process parameter and the parameter either points to something which is not a process or the process has terminated. Examples are the procedures P$LASTPROCESS and P$SUCCESSFUL, which both take an input parameter which must be a pointer to a process. Recall that a process pointer can be obtained by calling the the function MY$PROCESS.

2) aborted
   This error occurs in an aborted process after the user's system calls the procedure P$ABORT to abort the process.

3) 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.

4) not started - negative stacksize
   The "stacksize" given in the concurrent characteristics for the process must be non-negative.

5) not started - negative heapsize
   The "heapsize" given in the concurrent characteristics for the process must be non-negative.

6) not started - process is in assembly language
   User processes cannot be written in assembly language.

7) not started - no memory for semaphore
   This error indicates there was not sufficient memory for allocation of a semaphore used by the process.

8) not started - no memory for process heap
   This error indicates there was not sufficient memory for allocation of the process heap.

9) not started - no memory for process stack
   This error indicates there was not sufficient memory for allocation of the process stack.

10) not started - no memory for process frame
This error indicates there was not sufficient memory for allocation of the initial stack frame for the process.

13.2.6 Exception Errors

The following errors are those which can be encountered during exception handling.

1) handler not established from process
This error is received if the ONEXCEPTION routine is called from a user's procedure or function. The call of ONEXCEPTION must occur in the body of a process or program.

2) handler cannot have parameters
This error occurs if a candidate exception handler passed to the ONEXECPTION routine was defined to have parameters.

3) handler cannot be in assembly language
An exception handler must be written in Microprocessor Pascal, not in assembly language.

4) handler local variables too large for stack
This error occurs if a candidate exception handler passed to the ONEXECEPTION routine contains too many local variables.

13.2.7 Memory Management Errors

The following errors pertain to memory management problems which may occur.

1) 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.

2) heap overflow
This error indicates that the available heap space has been exhausted.

3) heap packet error
This error occurs when a heap packet is passed to a routine such as DISPOSE and the heap packet is invalid.

13.2.8 File Errors

The following errors pertain to file management problems.

1) 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.

2) text conversion, field width too large
This error occurs when a field width in a write statment is larger than the logical record length of the file.

3) 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.

4) text conversion, invalid character in text field
This error occurs when a field being read contained a character which was invalid for the particular data type, for example, the character "." when reading an integer value.

5) 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.

6) text read past end of file
This error occurs when an attempt is made to read past the end of a file.

7) text field exceeds record size
This error occurs when a specified field width is greater than the logical record size of the file.

8) file is not open for reading
This error occurs when a read attempt is made and the file was not opened for reading. A file must be opened for reading using RESET.

9) file is not open for writing
This error occurs when a write attempt is made and the file was not opened for writing. A file must be opened for writing using REWRITE.

10) sequential read past end of file
This error occurs when an attempt is made to read past the end of file for a sequential file.

50) no system memory for file descriptor
This error occurs when there is not sufficient memory space with which to allocate a file descriptor.

51) random files not implemented
Random files are not currently implemented.

52) file component length is incompatible with channel
This error occurs when a file is opened that has a logical record length that is smaller than the component length as declared in the user's system.

53) no system memory for descriptor of file parameter by value
This error occurs when there is not sufficient memory space with which to allocate a file descriptor being passed as a process parameter.

54) parameter to F$CHBUFFERS exceeds 255
The system procedure F$CHBUFFERS was called with a parameter greater than 255.

55) file parameter to F$CONDITIONAL is not sequential
A TEXT or RANDOM file was specified to be conditional. Only sequential files are allowed to be conditional.

56) file parameter to F$STLENGTH is not closed
A file variable must be closed before it can be specified as a parameter to F$STLENGTH.

57) F$STLENGTH component length is not in 1..8191
F$STLENGTH was called with a component length greater than 8191 or less than 1. Only the values 1..8191 are allowed.

58) F$STLENGTH component length greater than declared for file
The sequential file specified to F$STLENGTH has a declared component length which is less than that specified to F$STLENGTH.

60) RESET called for channel master before F$CREATECHANNEL
F$CREATECHANNEL must be called before a master file can be opened.

61) RESET called for channel master and master's mode is writing
F$STMODE was previously called to establish the mode of the file as WRITING. Only REWRITE may be used to open this file.

62) RESET called for channel master and master's mode is USERMODE
F$USERMODE was previously called to indicate that the user will establish the mode of this file. Therefore, the system routine F$WAIT must be called before the file is opened.

63) RESET called for channel master after F$WAIT and user's mode is reading
REWRITE must be called if user's mode is reading.

64) RESET called for channel master before CLOSE and F$WAIT
F$USERMODE was previously called to indicate that the user will establish the mode of this file. Therefore, once the file is open, CLOSE must be called to close it and F$WAIT called again to determine the mode of the next user.

65) REWRITE called for channel master before F$CREATECHANNEL
F$CREATECHANNEL must be called before a master file can be opened.

66) REWRITE called for channel master and master's mode is reading
F$STMODE was previously called to establish the mode of this file as READING. Only RESET may be used to open this file.

67) REWRITE called for channel master before F$WAIT
F$USERMODE was previously called to indicate that the user will establish the mode of this file. Therefore, F$WAIT must be called to determine the mode of the next user.

68) REWRITE called for channel master and user's mode is writing
F$USERMODE was previously called to indicate that the user will establish the mode of this file and the user's mode is writing. Therefore, a RESET must be done to open the master file for reading.

69) REWRITE called for channel master before CLOSE and F$WAIT
F$USERMODE was previously called to indicate that the user will establish the mode of the file. Therefore, once the file is open, CLOSE must be called to close it and F$WAIT must be called to determine the next user's mode.

70) F$MASTER called and file not closed
A file must be closed before F$MASTER can be called on it.

71) F$MASTER called twice for same file
F$MASTER can be called only once for a particular file.

72) no system memory for F$MASTER structures
There is not sufficient memory space with which to allocate structures needed by a master file.

73) F$EOC called and F$STEOC not called for file
A call to F$STEOC must be made before the function F$EOC can be called.

74) file parameter to F$STEOC is not channel master
F$STEOC may not be called unless F$MASTER is called first.

75) F$STEOC called after F$CREATECHANNEL
F$STEOC must be called before F$CREATECHANNEL.

76) parameter to F$STMODE is not in READING, WRITING, USERMODE
The parameter to F$STMODE is not of type MODE where MODE = (READING, WRITING, USERMODE).

77) file parameter to F$STMODE is not channel master
F$STMODE can only be called after F$MASTER and before F$CREATECHANNEL.

78) F$STMODE called after F$CREATECHANNEL
F$STMODE can only be called after F$MASTER and before F$CREATECHANNEL.

79) file parameter to F$ULENGTH is not channel master
F$ULENGTH can only be called after F$MASTER and before F$CREATECHANNEL.

80) F$ULENGTH called after f$createchannel
F$ULENGTH can only be called after F$MASTER and before F$CREATECHANNEL.

81) F$CREATECHANNEL called before F$MASTER
F$MASTER must be called before F$CREATECHANNEL.

82) F$CREATECHANNEL called before F$STMODE
F$STMODE must be called before F$CREATECHANNEL.

83) F$CREATECHANNEL called twice
F$CREATECHANNEL can only be called once for a particular file.

84) file parameter to F$WAIT is not channel master
F$MASTER and F$CREATECHANNEL must be called before F$WAIT for a particular file.

85) F$WAIT called and F$CREATECHANNEL not called
F$CREATECHANNEL must be called before F$WAIT for a particular file.

86) file parameter to F$WAIT is not closed
CLOSE must be called to close the file before F$WAIT is called.

87) file parameter F$XACCESS is not channel master
F$MASTER and F$CREATECHANNEL must be called before F$WAIT for a particular file.

88) F$XACCESS called after F$CREATECHANNEL
F$XACCESS must be called before F$CREATECHANNEL.

103) conditional read or write failed (nonfatal error)
A READ or WRITE was performed on a file which was established as conditional, and the attempt failed due to insufficient buffers. This is not a fatal error, and the READ or WRITE may be attempted again.

104) channel aborted
Some process has aborted the channel to which a user's file is connected. The file may be closed with CLOSE and reopened with REWRITE or RESET.

106) no system memory for channel buffers
This error occurs when system memory cannot be obtained to allocate the buffers for a channel.

200) no system memory for channel
This error occurs when system memory cannot be obtained to allocate a channel.

201) no system memory for pathname
This error occurs when system memory cannot be obtained to allocate space for the pathname of a channel.

202) invalid pathname
This error indicates that a file pathname is syntactically incorrect.

203) attempt to open device in an unsupported mode
This error occurs when an attempt is made to open a device in a mode that is not supported, for example, opening the printer for input.

204) device channel not initialized before user connected
This error indicates that a device handler did not create a device channel before user code attempted to connect to the channel.

205) attempt to initialize device channel with same name as existing user channel
This error indicates that a device handler attempted to create a device with the same name as some existing user channel.

206) attempt to open multiple device channels of same name with conflicting modes
This error indicates that a device handler attempted to create a device channel with the same name as some existing device channel.

13.2.9 Host File Errors

If a host file error is detected, a message is printed which contains a hexadecimal error code for either a DX or TX operating system (depending on which host system you are running). The appropriate file error message can be found in a DX or TX operating system manual.

13.3 Run-Time Execution Errors

Errors encountered at run-time cause a message of the following form:

Run-Time Error: reason for error

The reason for error message is one of the error messages described below.

1) invalid opcode
This error indicates that the interpreter encountered an illegal opcode during execution. This may have been caused by an error in the compilation of the program.

2) 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.

3) unresolved procedure call
This error occurs when a "call" instruction is encountered and the routine being called is not known to the interpreter. This kind of error is normally detected at compile-time. If a change is made to a system, the collect program should be executed to ensure that references to routines are not unresolved.

4) division by zero
This error occurs when division by zero is detected. The offending expression should be checked and corrected to avoid this error.

5) 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.

6) set element out of range
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.

7) 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 occurred at some point in the program.

8) 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.

9) array index error
or
12) LONGINT array index

This error occurs when a 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 (allocated) 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
or
13) LONGINT subrange 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.

13.4 CRITICAL TRANSACTIONS

Concurrrent processes which share data must synchronize their references to the data to avoid errors. Typically, the references to shared data are preceded and succeeded by semaphore WAIT and SIGNAL operations to prevent more than one process from simultaneously referencing shared data.

Consider that a process has executed a WAIT on a semaphore, has proceeded from that operation and is referencing the protected object (data), and then encounters an exception and aborts. Since the aborted process may not have finished its operations on the shared data, the data could be left in an inconsistent state. Another problem is that the aborted process is not able to SIGNAL the semaphore which protects the shared data upon which other sharing processes WAIT. The semaphore could be left in a state that it is never signaled again, and other sharing processes may be left suspended on the semaphore forever.

The problem outlined above shows that a section of code is sensitive and is designated here as a critical transaction. The following routine

```
procedure ct$enter; external;
```

indicates the entry of a critical transaction of the user's code. While a process is within a critical transaction as defined by the user, it is treated specially: process abort is remembered and not allowed until the process leaves the critical transaction. The entry and exit of critical transactions may be nested, that is, code within a critical transaction may call another critical transaction to implement it.

```
    procedure ct$exit; external;
```

The above routine indicates the exit of a critical transaction. Fatal errors encountered by a process, such as stack overflow, cause a process to fail even if it is within a critical transaction.

The Executive RTS manages many resources for concurrent processes using semaphores. When a user process is executing this RTS code, it is not apparent to the user that semaphores are being used to protect and to keep consistent shared data. As an example, a heap resource is managed with a semaphore in the heap administration record. If a process were aborted while it was executing a heap request (such as a NEW request), the heap resource is left inconsistent, and other processes will be suspended forever at their next operation on the same heap.

Critical transactions of code within the Executive RTS are bracketted by routines which are similar in function to CT$ENTER and CT$EXIT. Therefore, resources maintained by the Executive RTS are protected from the aborting of processes.

The concept of a critical transaction applies to more than a section of code which uses semaphores to synchronize concurrent processes that share data. A transaction can be defined to be a series of operations which may be executed by more than one process such that all of the operations must be completed (in possibly a strict order). An example is a command/response transaction in which a process sends a request to another process which in turn responds to the requestor. If a command is sent and the requestor process waits for a response, but the command servicing process fails, the requestor could wait forever. The recovery of this user-defined transaction is implemented with a protocol also implemented by the user.

13.5 EXCEPTION HANDLING

Within the text of a process (or program) a user may call the following run-time support routine:

```
        procedure onexception(handler_location: integer);
          external;
```

(read as "on exception") where HANDLER_LOCATION is derived by

```
        location(  procedure identifier  )
```

and PROCEDURE IDENTIFIER is the name of a procedure which is to be designated the exception handler for the process.

When an exception occurs, all routines which are currently active in the stack of the process are forced by the run-time support to immediately return and a call of the exception handler procedure is forced as though it were called from the body of the process. The exits of all routines which are currently active are done as though an ESCAPE routine name statement were called for each active routine. All data which had been local to the routines are lost. However, the parameters and variables declared in the process are left intact and in the same state as at the time of the exception. The process data are addressable to the exception handler and other routines which it calls to reprocess lost work.

The call of ONEXCEPTION must occur in the body of a process or program. The calling process fails if ONEXCEPTION is called from a procedure or function. The procedure which is designated as an exception handler cannot have any parameters and cannot be implemented in assembly language.

Once the exception handling procedure is invoked, it can fix up or reinitialize as appropriate. Then it can repeat the lost work of the process or can exit. Since the exception procedure was forcibly called by the run-time support and the return point from the procedure is undefined, an exit by the exception routine is interpreted to be an ESCAPE(process name); that is, the process terminates.

Notice that the instantiation of the exception handling procedure stays active throughout reprocessing after an exception. However, if another exception occurs, the instantiation of the exception handler is lost (by a simulated ESCAPE) and a new instantiation is caused by the forced call issued by run-time support. If variables are required to inform the exception handler that an exception occurred or how to reprocess, then these variables should be declared as process variables so they are left intact after all exceptions.

The Executive RTS allows a process repeatedly to fail for the same reason and restart ad infinitum. The user must be cognizant that a process which repeatedly fails should be allowed to terminate.

Following is an example sketch of exception handling.

```
process example {(...process parameters if appropriate...)};
  var "...process variables if appropriate...

  procedure accomplish_work;
  "This routine does the normal, main
   processing of process EXAMPLE.
   begin
      "...main processing...
   end {accomplish_work};

  procedure exception_handler {...no parameters allowed..} ;
    var "...routine variables if appropriate...
    begin
      "...handle exception...
      if "continuing work would be useful
      then accomplish_work "call routine which accomplishes
                              all the work of process EXAMPLE ;
      "otherwise exit EXCEPTION_HANDLER and process EXAMPLE
    end {exception_handler};

  begin
    onexception( location ( exception_handler ) );
    accomplish_work;
  end {example};
```

FIGURE 13-1. EXAMPLE SKETCH OF EXECUTION HANDLING

Code written by the user which services an exception can determine the current cause of the exception by calling the following two routines:

```
function err$class: integer; external;
function err$reason: integer; external;
```

Class codes and reason codes for each error are explained in Sections 13.2 and 13.3 and are listed in Appendix E.

The exception codes of the current process may be cleared by calling

```
procedure err$rset; external;
```

If a process terminates after an exception without clearing its exception codes with ERR$RSET, then the exception codes are available to the process termination servicing of the Executive RTS which reports abnormal process termination. If a process terminates with zero exception codes, because no exception ever occurred for the process or ERR$RSET had been called, then the process is considered by the Executive RTS to be terminating normally. The difference is only in reporting abnormal termination by the Executive RTS.

A user can force an exception with

```
        procedure exception(classcode, reasoncode:
          integer); external;
```

The process which executes this routine fails with the designated exception, and its exception handler is invoked. One class of errors is designated USER_ERROR and is never used by the Executive RTS. The user may call the EXCEPTION procedure with a class of USER_ERROR and any reason code.

A process (or program) that has not called ONEXCEPTION is considered not to have any exception handling code with which it can handle errors. The environment of every process includes a default exception handler which causes the process to abort. This default handler is invoked to service an exception otherwise not serviced by user-written code. If an exception occurs, the process is terminated as though an ESCAPE(process name) were called.

If reprocessing after an exception is not desired, the following routine

```
        procedure re$start; external;
```

causes the entire system to restart exactly as it did from application of power or from toggling an external reset switch. Low-level initialization is done to establish the Executive RTS data structures and the code declared in the SYSTEM is executed.

13.6 EXAMPLE

The following example illustrates exception handling for a process.

```
        const
          user_error       = 1;
          scheduling_error = 2;
          semaphore_error  = 3;
          interrupt_error  = 4;
            "....additional class error codes....

          invalid_queue    = 1;
          priority_error   = 2;
          not_a_process    = 1;
          aborted          = 2;
            "....additional reason error codes....
```

FIGURE 13-2. EXAMPLE OF EXCEPTION HANDLING FOR A PROCESS
(Sheet 1 of 2)

```
procedure onexception(handler_location: integer); external;
function err$class: integer; external;

process example;
  var i, j, k: integer;
    number_of_exceptions: integer;

  procedure accomplish_work;
  begin
    "Do main processing here.
  end {accomplish_work};

  procedure exception_handler;
  begin
    number_of_exceptions := number_of_exceptions + 1;
    if number_of_exceptions  = 3
    then
      case err$class of
        user_error: accomplish_work
                   "Try main processing again. ;
      "other cases as appropriate, i.e.
       semaphore_error: ......
       file_error: ...........
      end {case err$class of};
    "Exit EXCEPTION_HANDLER and process EXAMPLE.
  end {exception_handler};

begin {example}
{#priority=100; stacksize=400; heapsize=0}
  i := 0;
  "initialize and set up
  "This code is done exactly once.
  j := 1;
  number_of_exceptions := 0;

  onexception( location( exception_handler ) );
  accomplish_work;
end {example};
```

FIGURE 13-2. EXAMPLE OF EXCEPTION HANDLING FOR A PROCESS (Sheet 2 of 2)

13.7 RECOVERY OF FILES

A variable of type FILE may be declared local to a routine. If an instantiation of the routine is terminated by a user-called ESCAPE statement during normal processing or by a simulated ESCAPE during exception processing, the file variable is automatically closed. A process parameter or a variable declared local to a process is not affected by the Executive RTS during exception processing. The user may desire to declare files as process parameters or process variables to keep them in an open state during recovery processing by an exception handler.

13.8 PROCESS MANAGEMENT

A process identification is the dynamic "name" of a process which is assigned by the Executive RTS when a process is created. The user may define a process identification as follows:

```
    type processid = @ processid
                      "i.e. a pointer to something ;
```

The process identification of the current (calling) process is returned by

```
    function my$process: processid; external;
```

and the identification of the last process successfully started by process P is returned by

```
    function p$lastprocess(p: processid): processid; external;
```

A process can be involuntarily terminated by another process by means of the following run-time support procedure:

```
    procedure p$abort(p: processid); external;
```

If P$ABORT(P) is called, process P receives an ABORTED exception which causes it to fail. If process P is within a critical transaction defined by the user (by routines CT$ENTER and CT$EXIT) or by the Executive RTS modules, then the critical transaction is finished and the process is immediately caused to fail. Upon failure process P can terminate abnormally or recover using exception handling.

These routines are covered in more detail in Section 11 on Process Management.

13.9 SYSTEM CRASH

Non-recoverable errors are defined to cause a system crash. Under the Debugger, a message is issued indicating the crash. Executing stand-alone, the system crash code in the module 'USERINIT' will be entered (see Section 17.2). The errors for each crash condition are discussed below.

1) Interpretive RTS is unable to boot the system, probably because of insufficient memory.

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

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

(4) An unimplemented interrupt or XOP occurs and cannot be serviced.

(5) The scheduling queue has been destroyed; further scheduling is impossible.

(6) RAM made available to Interpretive RTS is found to be in error. An address specified to be RAM is either bad, ROM, or unimplemented memory.

(7) An interrupt has occurred for which the handler's priority is not urgent enough.

SECTION XIV

IMPLEMENTATION OF DEVICE HANDLERS

14.1 Introduction

The Executive RTS device management system interfaces supported physical devices to the Executive RTS logical file system. Conceptually, logical devices are processes executing concurrently and communicating with other processes through the Executive RTS logical file system. Each logical device has at least one dedicated channel through which it communicates with the rest of the system as illustrated in Figure 14-1. However, no physical device is capable of communicating with processes directly through an Executive RTS channel. Therefore, the implementation of a logical device requires an interface process which communicates with the channel through a file, interfaces to the physical device through CRU (or memory-mapped I/O) and synchronizes its execution with the device through interrupts (Figure 14-2).

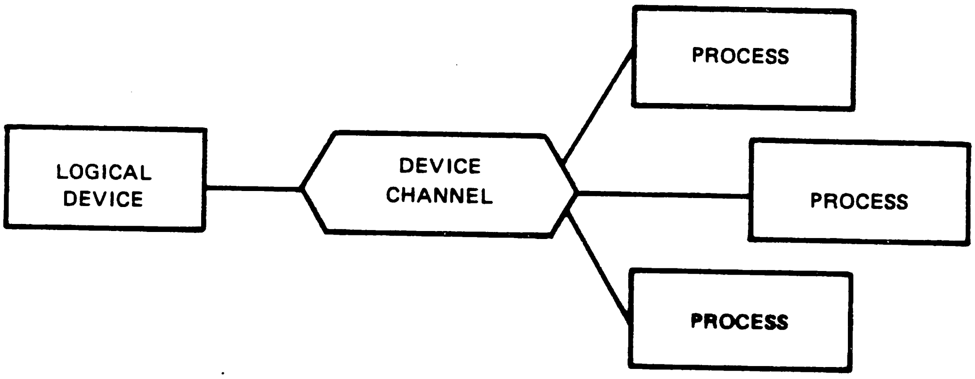


FIGURE 14-1. CONCEPTUAL VEIW OF INTERFACE TO A LOGICAL DEVICE

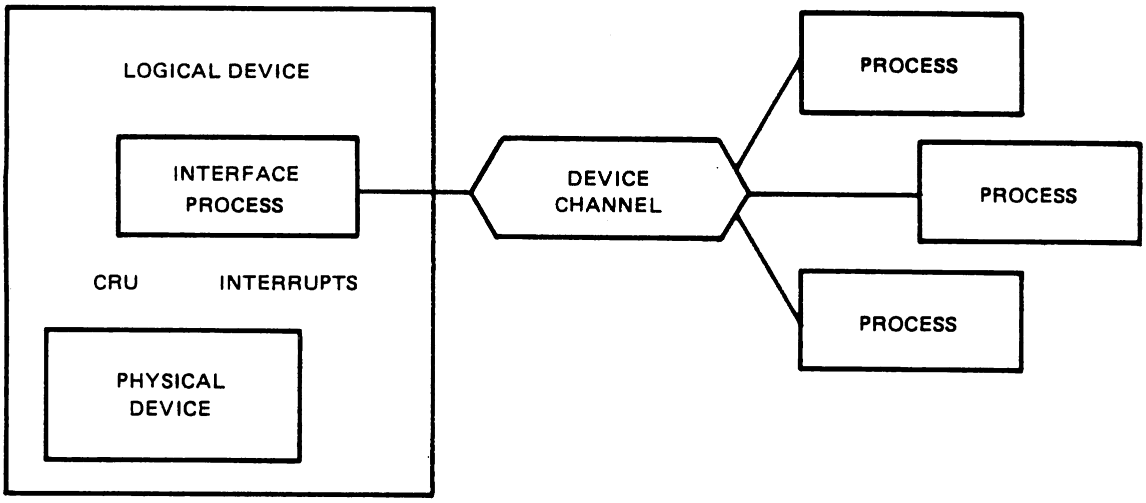


FIGURE 14-2. INTERFACE TO PHYSICAL DEVICE

Figure 14-3 illustrates the implementation of an interface process.

```
process interface_process(...);
  var
    f: file of some_type "file associated with channel ;
    component: some_type;
    ...
  begin {interface_process}
  ...
  reset(f);
  while not eof(f) do
    begin
    read(f, component) .receive component from channel ;
    "output component to device through cru"
    wait(completion_interrupt);
    end {while not eof( f )} ;
    ...
  end   {interface_process} ;
```

FIGURE 14-3. EXAMPLE SKETCH OF AN INTERFACE PROCESS

A single physical device may actually be more than one logical device. For example, an ASR 733 is a single physical device (it has a single device controller) but can be viewed as three logical devices: two cassette drives and a keyboard/printer. Outside of the interface processes, these logical devices appear to be independent. However, the interface processes are dependent on each other and must coordinate use of the single physical device among themselves. Figure 14-4 illustrates a physical device consisting of more than one logical device.

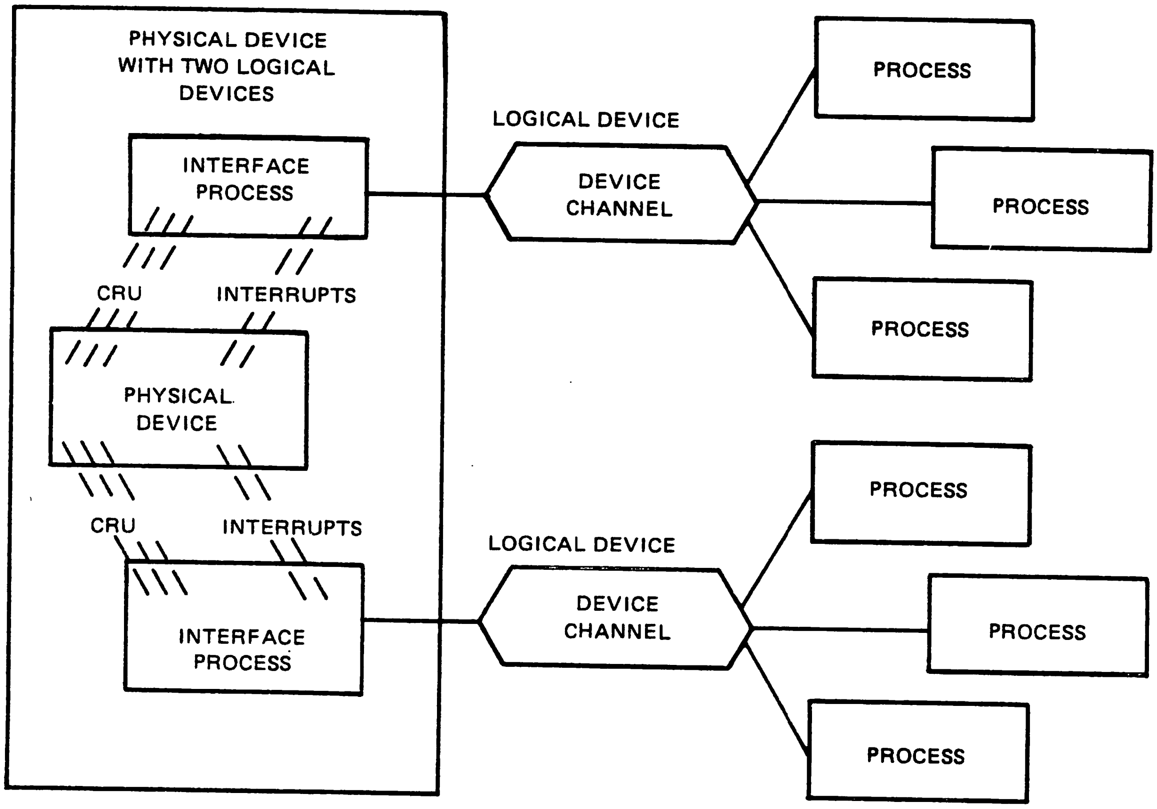


FIGURE 14-4. ILLUSTRATION OF MULTIPLE LOGICAL DEVICES ON A SINGLE PHYSICAL DEVICE

14.2 PHYSICAL DEVICE INTERFACE SYSTEMS

A physical device interface system is a collection of software modules which interfaces a particular type of physical device (such as a KSR 745) to the Executive RTS file system. These systems are reentrant allowing a single copy of the code to manage any number of supported devices. The components of a physical device interface system may include the following:

- an initialization procedure
- a supervisor program
- one or more logical device interface processes
- one or more logical device channels
- an optional interrupt demultiplexer process

```
type
  cru_address = 0..#1FFE;
  interrupt_level = 0..15;
  alfa = packed array [1..8] of char;

procedure asr$(base: cru_address;
               level: interrupt_level;
    printer_keyboard: alfa;
       left_cassette: alfa;
      right_cassette: alfa); external;
```

FIGURE 14-5. CALLING SEQUENCE OF EXAMPLE PHYSICAL DEVICE INTERFACE INITIALIZATION PROCEDURE

14.2.1 Physical Device Interface Initialization Procedure

A physical device interface initialization procedure is a user-callable, level-one procedure which is called to initialize an instance of the system for a particular physical device. The parameters to the procedure identify the CRU base address (or memory address for memory-mapped I/O), the interrupt level, and the names of each of the logical devices on the physical device. Some interface systems may require other information during initialization. An example calling sequence of a physical device interface initialization procedure for the ASR 733 is given in Figure 14-5.

A call to this procedure initializes one instance of the ASR 733 device interface system to service a physical device at CRU address BASE and interrupt level LEVEL. The names of each of the logical devices are also specified. This procedure is called from the SYSTEM body for each ASR 733 on the system. Figure 14-6 illustrates how four ASR devices are initialized on a single system.

```
system example;
  ...
  begin {system example}
  ...
  "configure physical device interface systems
    asr$(0000, 6, 'SYSLOG  ', 'INPUT   ', 'OUTPUT  ');
    asr$(0020, 7, 'ST01    ', 'CS03    ', 'CS04    ');
    asr$(0040, 8, 'ST02    ', 'CS05    ', 'CS06    ');
    asr$(0060, 9, 'ST03    ', 'CS07    ', 'CS09    ');
  ...
  end {system example}
```

FIGURE 14-6. INITIALIZATION OF FOUR ASR 733's

The only interfaces to a physical device interface system are 1) the Executive RTS logical file system via channels and (2) the physical device interface initialization procedure. Outside of these two isolated interfaces, the implementation of a physical device interface system is insignificant and can be modified without affecting the rest of the system. Therefore, a physical device interface system is a good example of a modular software component with an isolated, well-defined interface.

The initialization procedure starts the physical device interface supervisor program with appropriate parameters and waits for it to complete initialization. A semaphore should be initialized by the procedure and passed to the program to be signaled upon completion of initialization. The procedure can then execute a WAIT on the semaphore to ensure that channels associated with logical devices are created before the user's programs are started. A possible implementation of the ASR$ initialization procedure is illustrated in Figure 14-7.

```
program asr$supervisor(base: cru_base;
                           level: interrupt_level;
                printer_keyboard: text;
                   left_cassette: text;
                  right_cassette: text;
         initialization_complete: semaphore); forward;

procedure asr$(base: cru_address;
               level: interrupt_level;
    printer_keyboard: alfa;
       left_cassette: alfa;
      right_cassette: alfa);
  var
    initialization_complete: semaphore;
  begin {asr$}
  initsemaphore(initialization_complete, 0);
  start asr$supervisor(base, level,
    filenamed(printer_keyboard),
    filenamed(left_cassette),
    filenamed(right_cassette),
    initialization_complete);
  wait(initialization_complete);
  termsemaphore(initialization_complete);
  end   {asr$};
```

FIGURE 14-7. IMPLEMENTATION OF PHYSICAL DEVICE INTERFACE INITIALIZATION PROCEDURE

14.2.2 Physical Device Interface Supervisor Program

This program has the responsibility of the complete initialization of the interface system and reporting back to the initialization procedure that initialization has been completed. Other processes in the interface system are lexically nested within this program and can communicate with each other through the program's variables. Once initialization is complete, the program may terminate, or may function as a logical device interface process or an interrupt demultiplexer process. If it terminates, most of its resources are reclaimed; however, the global variables are preserved as long as there are active nested processes. Terminating the program after initialization allows the resources that are required for initialization only to be reclaimed. The shell of the deceased program also provides an encapsulated environment for the active processes in the system.

14.2.3 Logical Device Interface Process

A logical device interface process is a process which interfaces the logical device channel to the physical device. It has the responsibility of 1) communicating with the user's process through a channel, 2) editing the information to be communicated with the device (possibly to add or delete control characters or to respond to keyboard edit commands), 3) communicating with the physical device through CRU or memory-mapped I/O, 4) synchronizing its execution with the device through interrupts, 5) synchronizing with other interface processes in the same physical device interface system, 6) handle device errors, and 7) restart on channel abort. Logical device interface processes should have a priority equal to the interrupt level corresponding to the physical device. This ensures that no hardware interrupts from the device preempt the interface process.

14.2.4 Logical Device Channel

The Executive RTS provides file and channel routines which assist the implementation of device handlers. A device interface system needs more control over the characteristics and behavior of the associated channels than do user programs. Each file variable in a logical device interface process associated with a device channel is established as the channel master. This allows the interface process to manipulate the channel in ways prohibited to normal processes and identifies the process to handle channel abortions. A file is established as a channel master by calling the routine

```
     procedure f$master(var f: anyfile); external;
```

where F is the file variable. This does not cause channel creation or connection but does indicate that F is to be master of any channel that it creates. Each channel may have zero or one master, and a master must be the creator of the channel. The characteristics of a channel to be created by a master file are identified after F$MASTER is called and before the channel is created. These channel characteristics are also characteristics of the master file. The characteristics include the following.

- name - defaults to name of file
- component length - defaults to file component length
- mode (reading or writing)
- maximum number of users - defaults to 32767
- end of consumption handling

Once the characteristics of the channel have been established, the channel must be created by a call to the routine

```
     procedure f$createchannel(var f: anyfile); external;
```

If a channel already exists having the same name and mode, an exception occurs.

14.2.4.1 Channel Name. The standard procedure SETNAME can be used to assign the name given to the created channel.

14.2.4.2 Component Length. The component length of a channel be can dictated by the master file or can be implicitly initialized to the component length of the first connecting user file. The routine

```
     procedure f$stlength(var f: anyfile; length: integer);
       external;
      { read as "set length"}
```

is used to modify the component length of the file F to LENGTH. If F is a sequential file, LENGTH must be less than or equal to the length of its initial component and READ/WRITE operations will only affect the first LENGTH bytes of specified variables. Normally, LENGTH (or the default component length) is used for the component length of the channel. However, one can indicate that the component length of the channel is to be set to the component length of the first connecting user file by calling the routine

```
     procedure f$ulength(var f: anyfile); external;
      { read as "user defined length"}
```

In this case, the component length of F is used as an upper limit of the component length of the channel. If the first user file to connect is a text file, the component length of the channel is set to the component length of the master.

14.2.4.3 Channel Mode. The mode of a channel indicates which way information flows with respect to the master file. When the master is opened, its mode is always equal to the mode of the channel. However, it is possible for a master file to wait for the first user to open the channel to see which mode the user opens it. The mode of a channel is established by the routine

```
     procedure f$stmode(var f: anyfile; m: channel_mode);
       external;
      { read as "set mode"}
      { where channel_mode = (reading, writing, usermode)}.
```

If the mode of the channel is READING, the master file can only be opened by the RESET routine. If the mode is WRITING, then only REWRITE may be used. If the mode is USERMODE, the master file must wait until the first user connects to the channel by calling the routine

```
procedure f$wait(var f: anyfile; var m: mode); external;
```

This procedure suspends the calling process until a user connects to it and returns the mode of the user file in M. The master file should then be opened in the opposite mode. This is illustrated in Figure 14-13.

14.2.4.4 Maximum Number of Connected User Files. Normally, any number of user files are allowed to connect to a device. However, the master file may specify that users have exclusive access to the device while they have it opened by calling the routine

```
procedure f$xaccess(var f: anyfile); external;
```

Users attempting to open a device which is being used exclusively by another user are suspended until the device is released.

14.2.4.5 End of Consumption Handling. Normally, end of consumption on a channel is not significant as is end of transmission. However, it is possible to have end of consumption handled similarly to end of transmission. The routine

```
procedure f$steoc(var f: anyfile); external;
 {read as "set eoc"}
```

must be called to indicate that end of consumption is to be handled for the channel of which F is master. When end of consumption occurs on the channel, no other files are allowed to connect until all writing files are closed. The function

```
function f$eoc(var f: anyfile): boolean
```

returns a boolean which indicates that end of consumption has not occurred on the channel associated with F and that at least one more component can be written to the channel without being suspended forever. When end of consumption is detected on F, it is normally closed and some buffered components may be lost. The capability of handling end of consumption is necessary if the user is allowed to close a file open for reading before end of file is detected.

14.2.4.6 Device Channel Destruction. Device channels exist until termination of the stack frame in which the master file exists. Therefore, if the master file is a parameter to the logical device interface process, the associated channel will exist at least until that process exits.

14.2.4.7 Device Channel Abortions. As discussed in paragraph 10.2.13, a user may abort a device channel and cause all connected files to be disconnected. It also causes an exception to occur in the logical device interface process. If a device is to restart automatically, it should have an exception handler which detects the channel abortion and restarts processing of the logical device interface process. When the exception occurs, the channel still exists with the same characteristics. Processing should continue at the point that the file is open (or F$WAIT is called). If an exception handler is not provided, the logical device interface process terminates with the exception and the device channel is destroyed.

14.2.5 Interrupt Demultiplexer

An interrupt demultiplexer is a process which waits for an interrupt from a physical device, determines the logical device for which the interrupt is intended, and signals a semaphore corresponding to the logical device (Figure 14-8). An interrupt demultiplexer is only necessary when more than one logical device on a physical device needs to share a common interrupt. An alternative to demultiplexing interrupts is to allow each logical interface process to test the device interrupt until one of them claims it. This is a much slower method but does not require the overhead of the interrupt process.

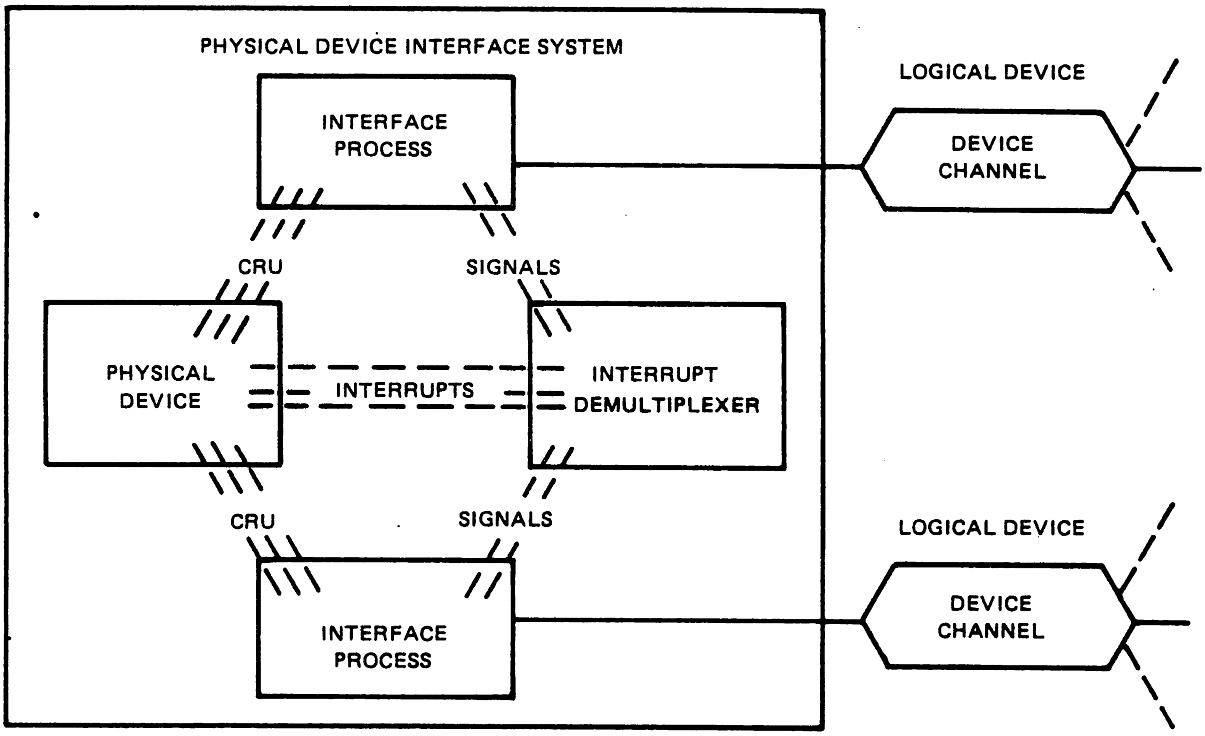


FIGURE 14-8. PHYSICAL DEVICE INTERFACE SYSTEM WITH INTERRUPT DEMULTIPLEXER PROCESS

An interrupt demultiplexer process should have a priority equal to the level of interrupt it services. This ensures that no hardware interrupts from the device being serviced can occur. However, when the logical device interface process is activated due to the signal, it preempts the interrupt process since both processes are device processes and their priorities are the same.

14.3 EXAMPLES

In this Section several cases of device handlers are studied. Partial designs and implementations are presented and are accompanied by discussions of the reasoning used to arrive at them. Since many different types of devices are considered, the examples should serve as a starting point in the implementation of most device handlers.

14.3.1 Physical Device Interface System for a Line Printer

A line printer is one of the simplest devices to handle. There is only one logical device and it can be opened in only one mode (output). Therefore, only one logical device interface process is required and no interrupt demultiplexer is required. Basically, the only processing required of the system is the addition of carriage control on output.

The initialization procedure for the line printer is given in Figure 14-9.

```
procedure lp$(base: cru_base;
  level: interrupt_level;
  name: alfa;
  line_length: integer;
  file_oriented: boolean)
```

FIGURE 14-9. CALLING SEQUENCE OF LINE PRINTER INITIALIZATION PROCEDURE

The integer LINE_LENGTH indicates the number of columns per line on the particular device. The boolean FILE_ORIENTED indicates that this particular device is to be used by a maximum of one user at a time. In other words, a user is granted exclusive access to the device as long as the file is opened. If the device is to be used as a report printer, it is important that lines of several users not be intermixed. Therefore, in this case FILE_ORIENTED should be specified as TRUE. If it is to be used as a log device, available to any number of users at a time, then FILE_ORIENTED should be FALSE. The parameter NAME is the name of the logical device channel to which user's files connect. Figure 14-10 illustrates the implementation of the initialization procedure.

```
program lp$supervisor(infile: text;
  base: cru_base;
  level: interrupt_level;
  line_length: integer;
  file_oriented: boolean;
  initialization_complete: semaphore); forward;

procedure lp$(base: cru_base;
  level: interrupt_level;
  name: alfa;
  line_length: integer;
  file_oriented: boolean);
  var
    initialization_complete: semaphore;
  begin {lp$}
  initsemaphore(initialization_complete, 0);
  start lp$supervisor(filenamed(name),
    base, level, line_length, file_oriented,
    initialization_complete);
  wait(initialization_complete);
  termsemaphore(initialization_complete);
  end   {lp$};
```

FIGURE 14-10. IMPLEMENTATION OF LINE PRINTER INITIALIZATION PROCEDURE

Since there is only one process required to service the device, it is convenient to allow the supervisor program to exist as that process after initialization is complete. Therefore, the supervisor program also takes the place of the logical device interface process.

```
program lp$supervisor(infile: text;
  base: cru_base;
  level: interrupt_level;
  line_length: boolean;
  file_oriented: boolean;
  initialization_complete: semaphore);

  var
    ch: char;
    interrupt: semaphore;

  procedure lp$put(ch: char); forward;

  begin {lp$supervisor}
 {#priority = level;
     stacksize = lp$stacksize;
     heapsize = lp$heapsize}
  initsemaphore(interrupt, 0);
  externalevent(interrupt, level);
  f$master(infile) "establish infile as channel master ;
  if file_oriented then f$xaccess(infile);
  f$stlength(infile, line_length) "establish maximum length ;
  f$ulength(infile)  allow user's sequential file of shorter
    component length to be used as the channel component length ;
  f$stmode(infile, reading) "establish mode of channel ;
  f$createchannel(infile) "create channel ;
  signal(initialization_complete) "allow procedure to continue ;

  "initialization is complete

  while true do {do forever} begin
    reset(infile) "wait for user to open ;
    while not eof(infile) do begin
      while not eoln(infile) do begin
        read(infile, ch);
        lp$put(ch) "output character to device ;
        end {while not eoln};
      if ch >= ' ' "if last character was not control character
        then lp$put(line_feed) "output carriage control ;
      readln(infile) "get next line from channel ;
      end {while not eof(infile)};
    lp$put(form_feed) "advance form to top of page ;
    end {while true do}
  end   {lp$supervisor};
```

FIGURE 14-11. IMPLEMENTATION OF LINE PRINTER SUPERVISOR PROGRAM

The program first associates the semaphore INTERRUPT with the interrupt level of the device. It then establishes INFILE as a master file, initializes the characteristics of the logical device channel and creates the channel. The semaphore INITIALIZATION_COMPLETE is then signaled to allow the initialization procedure to continue. At this point the program changes roles and becomes the logical device interface process. The WHILE TRUE DO loop is executed forever copying file sequences to the line printer. Within this loop it opens INFILE for reading and proceeds to read lines from it until the logical end of file occurs at which time it outputs a form feed to eject the last page printed, loops back, and RESETs INFILE waiting for the next user. Within each line, characters are read one at a time and output to the device by the procedure LP$PUT. At the end of each line a test is made to see if the last character in the line is a control character. If it is, it is assumed that the user is doing his own carriage control and no additional carriage control is added. If the last character is not a control character, a line feed is output to the device. (The implementation of PAGE(F) is WRITELN(F,FORM_FEED) so that the carriage control is at the end of the line.) Figure 14-12 illustrates an example implementation of LP$PUT.

This procedure is lexically nested within LP$DEVICE and can access its parameters and variables. This procedure waits until the device is ready to accept a character, outputs the character (actually the inverse of the character) with an LDCR, and returns. Interrupts are normally disabled on the device. The only time they are enabled is while an interrupt is being waited upon. Therefore, unsolicited interrupts will not occur when they are not expected.

The performance of this handler can be increased by placing the code for LP$PUT in-line, thus avoiding the overhead of one procedure call for each character.

```
procedure lp$put(ch: char);
  "output ch to device through CRU
  begin {lp$put}
  crubase(base);
  sbo(strobe);
  while tb(not_demand) do begin {wait for demand}
    sbo(interrupt_enable) "enable interrupts ;
    wait(interrupt) "wait for interrupt ;
    sbz(interrupt_clear) "clear interrupt ;
    sbz(interrupt_enable) "disable interrupts ;
    sbo(strobe);
    end; {while tb(not_demand)}
  ldcr( 7, -1 - ord(ch) )  output inverted character ;
  sbz( strobe );
  end   {lp$put};
```

FIGURE 14-12. EXAMPLE OF LINE PRINTER DEVICE MANIPULATION

SP 1 14.3.2 Logical Device Interface Process for a Cassette Drive

This example is chosen to illustrate the handling of a device that may be opened for either input or output but only one at a time. The logical device channel is initialized to have a mode of USERMODE which indicates that the first user file to connect to the channel establishes the mode. The routine F$WAIT causes suspension until the first user file is connected and then indicates the mode that the user's file is in. The process then goes into a read loop or a write loop depending on the mode of the user. Once the transmission is complete, CLOSE is called to close the file and F$WAIT is called again. (See Figure 14-13 on following page.)

```
process cassette_drive(f: text;
  initialization_complete: semaphore);

  var
    users_mode: file_mode;
    ch: char;

  begin {cassette_drive}
  f$master(f);
  f$xaccess(f) "allow single user at a time ;
  f$stlength(f, 80) "establish maximum length ;
  f$ulength(f) "allow user's sequential file of shorter
    component length to be used as the channel's component length ;
  f$stmode(f, usermode) "allow user to initialize mode ;
  f$steoc(f) "set end of consumption flag ;
  f$createchannel(f) "create channel ;
  signal(initialization_complete);

  while true do begin
    f$wait(f, users_mode) "wait for user to connect to channel ;
    case users_mode of
      reading: begin {user opened for reading}
        rewrite(f) "open channel for writing ;
        while not f$eoc(f) do begin
          getch(ch) "get first character on line ;
          while ch <> carriage_return do
            write(f, ch) "write character to file ;
            getch(ch) "get next character in record ;
            end {while ch <> carriage return};
          writeln(f) "send line to channel ;
          end {while not f$eoc(f)};
        end {reading};
      writing: begin {user opened for writing}
        reset(f) "open channel for reading ;
        while not eof(f) do begin
          while not eoln(f) do begin
            read(f, ch);
            putch(f, ch);
            end {while not eoln(f)};
          putch(carriage_return) "add carriage return ;
          readln(f) "get next component from channel ;
          end {while not eof(f)};
        putch(end_file_character); putch(carriage_return);
        end {writing};
      end {case};
    close(f);
    end {while true do};
  end   {cassette_drive};
```

FIGURE 14-13. IMPLEMENTATION OF CASSETTE LOGICAL DEVICE INTERFACE PROCESS

14.3.3 Implementation of a Video Display Terminal Handler

This example illustrates the complete implementation of a device handler for the 911 video display terminal. It is a very thorough example containing a high degree of technical detail. Therefore, several modules of the handler are presented separately with appropriate discussions. These modules may have much of the technical detail removed to preserve clarity. At the end of this example, the entire implementation is presented as a complete Microprocessor Pascal-compilable module with all technical detail present (Figure 14-22).

14.3.3.1 User Interface and Operation of VDT. The 911 VDT is a very versatile device which can be treated in many different ways by a device handler. The device handler implemented in this example is relatively simple, supporting only line-oriented I/O with automatic cursor control. (The user is not free to format the screen and control the cursor himself.) This handler does allow the device to be opened for both reading and writing at the same time. User programs communicate with the VDT through text files or sequential files having a component length of 80 bytes or less. Performing a RESET on a file having the same name as the VDT device will cause the file to be connected to the keyboard. A REWRITE on a file having the same name as the VDT device will cause the file to be connected to the screen. When the user's program requests input from the keyboard the entire screen is rolled up one line leaving the last line blank. The cursor then appears in the first column of the last line indicating to the keyboard operator that input is being requested. Characters input from the keyboard are then echoed in high intensity on this line and can be edited using control characters before the line is transmitted to the user's program. A carriage return will transmit the edited line to the user's program and will allow the displayed line to be rolled up the screen.

When a user's program writes a line to the screen, all available lines are rolled up and the output is displayed on the last available line in low intensity. (If a read is in progress, then all but the last line are rolled up and the output is displayed on the next to the last line of the screen. If a read is not in progress, all lines are rolled up and the output is displayed on the last line.) One very useful feature of this handler is that it allows output to continue while the operater is editing a line of input. Also, keyboard input is displayed in high intensity and is rolled up the screen with the output lines which are in low intensity.

The VDT is actually treated as two logical devices, the screen and the keyboard, having the same device name. The handler is implemented with two logical device interface processes, one for each of the logical devices. There are also two channels having the same name but opposite modes. User files attempting to connect are connected to the channel of the appropriate mode. Figure 14-14 illustrates the connections of user files to the device channels for a VDT named "VDT01".

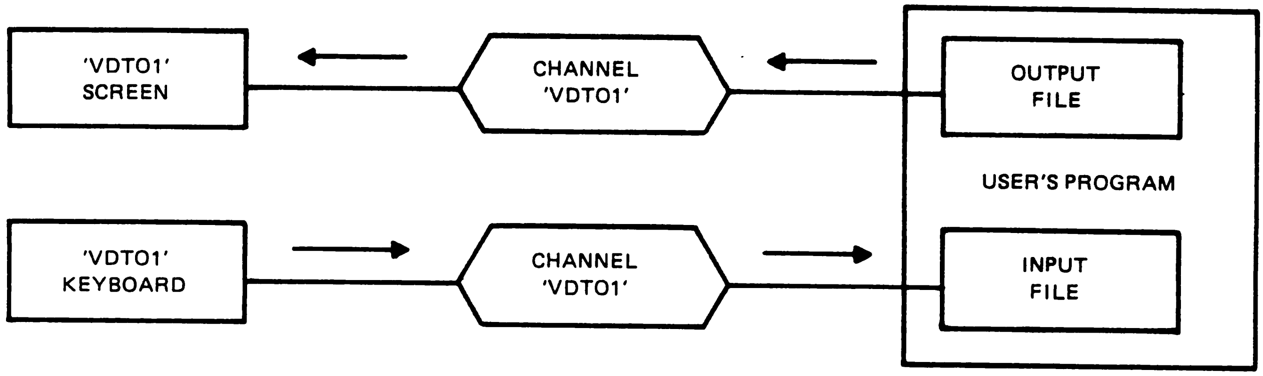


FIGURE 14-14. EXAMPLE OF CONNECTION OF USER FILES TO A VDT

The VDT physical device interface system is initialized by calling the procedure VDT$ which has the following calling sequence:

```
type
  cru_address = 0..#1FFE;
  interrupt_level = 0..15;
  alfa = packed array  [1..8]  of char;

procedure vdt$(
  base: cru_address;
  level: interrupt_level;
  name: alfa;
  file_oriented: boolean );
  external;
```

14.3.3.2 Implementation of Initialization Procedure. Figure 14-15 illustrates the implementation of the procedure VDT$ which is invoked to initialize the entire system.

```
procedure vdt$(
  base: cru_address;
  level: interrupt_level;
  name: alfa;
  file_oriented: boolean );
var
  initialization_complete: semaphore;
begin
initsemaphore( initialization_complete, 0 );
start vdt$supervisor( base, level, file_oriented, name,
  initialization_complete );
wait( initialization_complete );
termsemaphore( initialization_complete );
end;
```

FIGURE 14-15. VDT INTERFACE SYSTEM INITIALIZATION PROCEDURE

14.3.3.3 Implementation of Supervisor Program. The VDT physical device interface system supervisor program is responsible for initializing the device and processes in the system, and informing the initialization procedure that initialization is complete. Figure 14-16 illustrates the implementation of the VDT supervisor program.

The concurrent characteristics for this program specify that its priority is to be the same as the interrupt level of the device being serviced. The constants VDT$SUPERVISOR_STACK and VDT$SUPERVISOR_HEAP are constants defined at the system level. The first thing done by the program is the initialization of the physical device. Because a master reset is done when the Executive

```
program vdt$supervisor(
  base: cru_address;
  level: interrupt_level;
  file_oriented: boolean;
  name: alfa;
  initialization_complete: semaphore );
var
  partial_completion: semaphore;
  output_line: integer;
  exclusive_access_to_physical_device: semaphore;

  process vdt$keyboard(outfile: text; level: interrupt_level);
  ...
  end;

  process vdt$screen(infile: text; level: interrupt_level);
  ...
  end;

begin {vdt$supervisor}
{#priority = level;
 stacksize = vdt$supervisor_stack;
 heapsize = vdt$supervisor_heap}
crubase(base) "set cru base to initialize device ;
sbz(select_word_w0_w1) "select word zero ;
sbo(enable_display_w0) "enable display ;
output_line := 24 "last line available for output ;
initsemaphore(partial_completion, 0);
initsemaphore(exclusive_access_to_physical_device, 1);
start vdt$keyboard(filenamed(name), level);
  wait(partial_completion);
start vdt$screen(filenamed(name), level);
  wait(partial_completion);
signal(initialization_complete);
termsemaphore(partial_completion);
end   {vdt$supervisor};
```

FIGURE 14-16. VDT INTERFACE SYSTEM SUPERVISOR PROGRAM

RTS initializes, most of the initialization of the device has already taken place. The only thing left to be done is to enable the display on the screen.

Then the semaphore PARTIAL_COMPLETION is initialized to zero. This semaphore is used by the program to wait for completion of each process started. The semaphore EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE is initialized to one. This semaphore is used by the logical device interface processes (VDT$KEYBOARD and VDT$SCREEN) to synchronize access to the physical device.

The integer OUTPUT_LINE is then initialized to 24. This variable is used by the nested processes to indicate the last available line for output.

The logical device interface processes are then initiated with the START statement.

A WAIT(PARTIAL_COMPLETION) is performed after each initiation to wait until that process has initialized all necessary structures.

Then the semaphore INITIALIZATION_COMPLETE is signaled to indicate to the initialization procedure that it may proceed. The last thing done is the termination of the semaphore PARTIAL_COMPLETION since it is needed no longer.

14.3.3.4 Implementation of VDT Screen Logical Device Process. The VDT screen logical device process has the responsibility of transferring the information from the file INFILE to the VDT screen through the CRU. Figure 14-17 illustrates the implementation of this process.

```
process vdt$screen(infile: text; level: interrupt_level);

  procedure vdt$swork;
  ...
  end;

  procedure vdt$sexception;
  ...
  end;

begin {vdt$screen}
{#priority = level;
 stacksize = vdt$screen_stack}
f$master(infile) "establish INFILE as channel master ;
if file_oriented then f$xaccess(infile);
f$stlength(infile, screen_line_length) "set max line length ;
f$ulength(infile) "allow user's sequential file of shorter
  component length ;
f$stmode(infile, reading) "set mode of screen channel ;
f$createchannel(infile) "create screen channel ;
signal(partial_completion);
onexception( location(vdt$sexception) ) "establish exception
  handler ;
vdt$swork "invoke screen work procedure ;
end   {vdt$screen};
```

FIGURE 14-17. VDT SCREEN LOGICAL DEVICE PROCESS

The nested procedure VDT$SWORK does the majority of the work after the necessary structures are initialized. The nested procedure VDT$SEXCEPTION is the exception handler for this process. The channel characteristics of the screen device channel are initialized through the file INFILE. INFILE is established as a channel master by the routine F$MASTER. If the device is to be file oriented (FILE_ORIENTED=TRUE) the routine F$XACCESS is called to set the maximum number of users to one. The routine F$STLENGTH is then called to set the maximum line length for user's text files or maximum component length for user's sequential files. The call to F$ULENGTH indicates that the user's component length is to be used if it is shorter. The mode of the device channel is then specified by calling the procedure F$STMODE. The modes of the two identically-named channels are what distinguish them from each other. The call to F$CREATECHANNEL creates the channel (but INFILE is still not opened). The semaphore PARTIAL_COMPLETION is then signaled to indicated to the supervisor program that initialization of the device channel is complete. The Executive RTS procedure ONEXCEPTION is called to establish VDT$SEXCEPTION as the exception handler and the procedure VDT$SWORK is called to perform the transfer of information from INFILE to the screen. The implementaion of VDT$SWORK is illustrated in FIGURE 14-18.

```
procedure vdt$swork;
var
  ch: char;
  oldcursor: integer;
begin {vdt$swork}
crubase(base);
repeat reset(infile);"copy files forever
  while not eof(infile) do begin
    wait(exclusive_access_to_physical_device);
    sbo(select_word_w0_w1);
    stcr(11, oldcursor) "save original cursor address ;
    vdt$rollup(output_line);
    while not eoln(infile) do begin
      read(infile, ch);
      sbz(select_word_w0_w1);
      ldcr( 7, ord(ch) ) "load into character buffer ;
      sbo(set_low_intensity_w0) "low intensity ;
      sbo(write_data_strobe_w0) "strobe data to screen ;
      sbz(move_cursor_w0) "move cursor right ;
      end {while not eoln(infile)};
   "restore cursor address to oldcursor
    sbo(select_word_w0_w1);
    ldcr(11, oldcursor);
    signal(exclusive_access_to_physical_device);
    readln(infile);
    end {while not eof(infile)};
  until eternity;
end  "vdt$swork ;
```

FIGURE 14-18. IMPLEMENTATION OF VDT SCREEN WORKER PROCEDURE

The code within the outer-most REPEAT loop copies a single file sequence from INFILE to the screen. The RESET opens INFILE for reading and connects it to the device channel. The code within the WHILE NOT EOF(INFILE) copies a single line from INFILE to the screen. The process will be suspended each time through this loop at the EOF(INFILE) until a connected user file writes a line. Within the loop the process maintains exclusive access to the physical device while it rolls lines up and copies the characters from the line buffer of INFILE to the screen. Also, the cursor position is saved before each line is written and restored before exclusive access to the device is released. Therefore, except for the short amount of time that the line is being displayed, the cursor's position is maintained by the keyboard process to indicate which character is being edited in the edit line.

Aborting the logical device channel for the screen causes all file variables connected to the channel to become disconnected. Any subsequent READ of a disconnected user file variable results in an exception until that file is opened again. Any files suspended on the channel being aborted are activated with an exception and an exception will occur for the process VDT$SCREEN since INFILE is the master of the channel.

Any exception of VDT$SCREEN causes an implicit escape from VDT$SWORK and an implicit invocation of VDT$SEXCEPTION. The implementation of VDT$SEXCEPTION is illustrated in Figure 14-19.

The only exception handled by this routine is one of class F$$CLASS_FILE_ERROR (file errors) and reason F$$REASON_CHANNEL_ABORT (channel abortion). All other exceptions are ignored and cause a termination of VDT$SCREEN with the exception still present.

If the exception is a channel abort, the exception is reset (cleared) and VDT$SWORK is invoked to restart the transfer. Since the VDT screen device channel has a master, it is not destroyed when aborted. Therefore, it is not necessary to initialize the channel again. The RESET in VDT$SWORK will first close INFILE and will then

```
procedure vdt$sexception;
begin {vdt$sexception}
if err$class = f$$class_file_error
  and err$reason = f$$reason_channel_abort
  then begin
    err$rset "clear error ;
    vdt$swork "invoke work procedure again ;
    end {then};
{no other exceptions are handled.
 other exceptions will cause process termination with
 error code and no error recovery};
end   {vdt$sexception} ;
```

FIGURE 14-19. IMPLEMENTATION OF VDT SCREEN EXCEPTION HANDLER

connect INFILE to the channel again. At this time, user files are allowed to connect to the channel.

14.3.3.5 Implementation of VDT Keyboard Logical Device Process. The VDT keyboard logical device process has the responsibility of reading, editing and echoing characters input from the keyboard and transferring edited lines to users through the file OUTFILE. Figure 14-20 illustrates the implementation of this process.

The nested procedure VDT$KWORK does the majority of the work after the necessary structures are initialized. The nested procedure VDT$KEXCEPTION is the exception handler for this process. The channel characteristics of the keyboard device channel are initialized through the file OUTFILE. The initialization of this channel is identical to the initialization of the screen channel with the exception of the channel mode. The screen channel is given the mode READING and the keyboard channel is given the mode WRITING. After the channel is created, the semaphore PARTIAL_COMPLETION is signaled to indicate to the supervisor program that initialization of the device channel is complete. The procedure VDT$KEXCEPTION is established as the exception handler and the procedure VDT$KWORK is invoked to perform the transfer of information from the keyboard to the user through OUTFILE. A simplified version of VDT$KWORK is illustrated in Figure 14-21. The completed routine appears in Figure 14-22.

```
process vdt$keyboard(outfile: text; level: interrupt_level);
var
  interrupt: semaphore;
begin {vdt$keyboard}
{#priority = level;
 stacksize = vdt$keyboard_stack}
initsemaphore(interrupt, 0);
externalevent(interrupt, level) "associate semaphore with
  interrupt level ;
f$master(outfile) "establish OUTFILE as channel master ;
if file_oriented then f$xaccess(outfile);
f$stlength(outfile, screen_line_length) "set maximum line
  length ;
f$ulength(outfile) "allow user's sequential file of shorter
  component length ;
f$stmode(outfile, writing) "set mode of keyboard channel ;
f$createchannel(outfile) "create channel ;
signal(partial_completion);
onexception( location(vdt$kexception) ) "establish exception
  handler ;
vdt$kwork "invoke work procedure ;
end   {vdt$keyboard};
```

FIGURE 14-20. VDT KEYBOARD LOGICAL DEVICE PROCESS

```
procedure vdt$kwork;
var
  ch: char; cursor: integer;
begin {vdt$kwork} rewrite(outfile);
repeat "forever copying lines
  wait(exclusive_access_to_physical_device);
  output_line := 23;
  vdt$rollup(24) "reserve line at bottom for editing ;
  "enable cursor display"
  repeat "get character from keyboard
    "enable keyboard interrupt"
    signal(exclusive_access_to_physical_device);
    wait(interrupt);
    wait(exclusive_access_to_physical_device);
    "acknowledge and disable keyboard interrupt"
    "input character from keyboard into ch"
  "edit character
    if ch = backspace then
      if column(outfile) > 0 then begin "move cursor left"
        f$bspace(outfile);
        end {then if column(outfile) > 0 then begin}
      else {nothing to do}
    else if ch = rubout then
      while column(outfile) > 0 do begin
        "move cursor left"
        f$bspace(outfile);
        end {else if ch = rubout then)}
    else if ch <> carriage_return and ch <> dc3 then begin
      if ch = dc2  right arrow
        then "read ch from screen";
      if eoln(outfile)
        then f$bspace(outfile) "replace last character ;
      write(outfile, ch);
      if ch >= ' ' then begin "display character
        if not eoln(outfile)
          then "move cursor right";
        end . if ch >= ' ' then  ;
      end .else if ch <> carriage_return then ;
    until ch < ' ' or ch > '';
  "blank rest of line"
  output_line := 24 "include in next roll up ;
  signal(exclusive_access_to_physical_device);
  if ch = dc3 then rewrite(outfile) else writeln(outfile);
  until eternity;
 end  {vdt$kwork};
```

FIGURE 14-21. IMPLEMENTATION OF VDT KEYBOARD WORKER PROCEDURE

The REWRITE opens OUTFILE for writing and connects it to the device channel. The code within the outer-most REPEAT loop inputs a single line from the keyboard and transmits it to the user through the file OUTFILE. The code within the nested REPEAT inputs a character from the keyboard and edits it. There are only two places within the outer-most REPEAT that the process may be suspended for a significant amount of time. These are the WAIT(INTERRUPT) and the last statement of the loop (REWRITE or WRITELN). Therefore, the process has exclusive access to the physical device except at these places. Exclusive access is acquired with the first statement of the loop with the WAIT(EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE) and is released just before waiting on the interrupt. It is acquired again after the interrupt occurs and released again just before the WRITELN (or REWRITE). The Executive RTS procedure

```
    procedure f$bspace(var f: text); external;
```

causes the column index of F to be decremented if it is greater than zero. The function

```
    function column(var f: text): integer; external;
```

returns the current value of the column index of F. The standard function EOLN(F) can be used when F is open for writing. It returns FALSE if another character can be written to F without causing an implicit WRITELN(F). If it is TRUE, writing another character will cause an implicit WRITELN(F).

The exception handler for the keyboard process is very similar to the exception handler for the screen process. The entire VDT physical device interface system is presented in Figure 14-22.

```
system vdt$handler;
  {hierarchical relationships among modules in vdt$handler:

      vdt$handler. . . . . . . . .null-bodied system environment
        vdt$supervisor . . . . .physical device supervisor program
          vdt$rollup . . . . . .internally used procedure
          vdt$keyboard . . . . .keyboard logical interface process
            vdt$kwork. . . . . .keyboard worker procedure
            vdt$kexception . . .keyboard exception handler
          vdt$screen . . . . . .screen logical interface process
            vdt$swork. . . . . .screen worker procedure
            vdt$sexception . . .screen exception handler
        vdt$ . . . . . . . . . .physical device initialization procedure}

    const

      vdt$suprvisor_stack = 200;
      vdt$supervisor_heap = 400;
      vdt$keyboard_stack = 200;
      vdt$screen_stack = 400;

    {cru input bit assignments}    {cru output bit assignments}
     test_low_intensity_w0 = 7;     set_low_intensity_w0 = 7;
     keyboard_data_w0 = 8;          write_data_strobe_w0 = 8;
     keyboard_data_ready_w0 = 15;   move_cursor_w0 = 10;
     keyboard_data_msb_w1 = 11;     enable_blinking_cursor_w0 = 11;
     terminal_ready_w1 = 12;        enable_keyboard_interrupt_w0 = 12;
     previous_word_select_w1 = 13;  enable_hi_low_intensity_w0 = 13;
     keyboard_parity_error_w1 =14;  enable_display_w0 = 14;
     keyboard_data_error_w1 = 15;   select_word_w0_w1 = 15;
                                    enable_cursor_display_w1 = 12;
                                    acknowledge_keyboard_w1 = 13;
                                    enable_beep_w1 = 14;

    {control characters}
     backspace = '#08';
     carriage_return = '#0D';
     dc2 = '#12' {right arrow};
     dc3 = '#13' {end of file character};
     rubout = '#7F';

     maxint = 32767;
     screen_line_length = 80;
     f$$class_file_error = 8    {exception class for file errors};
     f$$reason_channel_abort = 104   {exception reason for channel
       abortions};
     cursor_of_24th_line = 23*80;
     last_cursor_on_screen = 24*80-1;
     eternity = false;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 1 of 7)

```
  type
    cru_address = 0..#1FFE;
    interrupt_level = 0..15;
    alfa = packed array  [1..8]  of char;
    nonneg = 0..maxint;
    channel_mode = (reading, writing, usermode);
    word = packed record case integer of
      0: (most_significant_char, least_significant_char: char );
      end {word};

{Executive RTS routines}
 function column(var f: text): integer; external;
 procedure f$bspace(var f: text); external;
 procedure f$createchannel(var f: anyfile); external;
 procedure f$master(var f: anyfile); external;
 procedure f$stmode(var f: anyfile; mode: channel_mode); external;
 procedure f$stlength(var f: anyfile; length: integer); external;
 procedure f$ulength(var f: anyfile); external;
 proceudre f$xaccess(var f: anyfile); external;

 function err$class: integer; external;
 function err$reason: integer; external;
 procedure err$rset; external;
 procedure onexception(handler_location: integer); external;

 procedure initsemaphore(var s: semaphore; n: nonneg); external;
 procedure termsemaphore(var s: semaphore); external;
 procedure signal(s: semaphore); external;
 procedure wait(s: semaphore); external;
 procedure externalevent(s: semaphore; level: interrupt_level);
    external;

 program vdt$supervisor(
    base: cru_address;
    level: interrupt_level;
    file_oriented: boolean;
    name: alfa;
    initialization_complete: semaphore);
  var partial_completion: semaphore;
    output_line: integer;
    exclusive_access_to_physical_device: semaphore;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 2 of 7)

```
procedure vdt$rollup(last_line: integer);
{ roll lines 1 through LAST_LINE up one row leaving line
  LAST_LINE blank}
  var chbyte: 0..255; cursor, cursor_at_last_line: integer;
begin {vdt$rollup  assert last_line > 0 and last_line <= 24;
crubase(base) .establish cru base};
sbz(select_word_w0_w1);
cursor_at_last_line := (last_line-1) * 80;
for cursor_of_next_column_in_last_line
  := cursor_at_last_line to cursor_at_last_line + 79 do begin
  cursor := cursor_of_next_column_in_last_line;
  chbyte := ord( ' ' );
  for row := last_line downto 1 do begin
    sbo( select_word_w0_w1 )  {select word 1} ;
    ldcr( 11, cursor )  {position cursor on screen} ;
    sbz( select_word_w0_w1 )  {select word 0} ;
    ldcr( 8, chbyte )  {write last byte into screen buffer} ;
    stcr( 8, chbyte )  {read last visible character} ;
    sbo( write_data_strobe_w0 )  {strobe buff byte to screen} ;
    cursor := cursor - 80  {set cursor to same column of
      previous line} ;
    end  {for row} ;
  end  {for cursor_of_next_column_in_last_line} ;
sbo( select_word_w0_w1 )  {select word 1} ;
ldcr( 11, cursor_at_last_line )  {leave cursor here} ;
 {leave word one selected}
end   { vdt$rollup } ;

process vdt$keyboard( outfile: text; level: interrupt_level );
var interrupt: semaphore;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 3 of 7)

```
procedure vdt$kwork;
var
  ch: char; cursor: integer;
begin { vdt$kwork }
crubase( base );
rewrite( outfile );
repeat { forever copying lines}
  wait( exclusive_access_to_physical_device );
  output_line := 23;
  vdt$rollup( 24 ) { reserve line at bottom for editing };
  sbo( enable_cursor_display_w1 );
  repeat { get character from keyboard}
    sbz( select_word_w0_w1 );
    sbo( enable_blinking_cursor_w0 );
    sbo( enable_keyboard_interrupt_w0 );
    signal( exclusive_access_to_physical_device );
    wait( interrupt );
    wait( exclusive_access_to_physical_device );
    sbz( select_word_w0_w1 ) { select word zero} ;
    sbz( enable_keyboard_interrupt_w0 );
    stcr( 15, ch::integer );
    ch := ch::word.most_significant_char;
    sbz( enable_blinking_cursor_w0 );
    sbo( select_word_w0_w1 ) { select word one} ;
    sbo( acknowledge_keyboard_w1 );
  { edit character}
    sbz( select_word_w0_w1 ) { select word zero} ;
    if ch = backspace then
      if column( outfile ) > 0
        then begin
          sbo( move_cursor_w0 ) { move cursor left} ;
          f$bspace( outfile ) { will not cause suspension} ;
          end { then if column( outfile ) >0 then begin }
        else { nothing to do}
    else if ch = rubout then
      while column( outfile ) > 0 do begin
        sbo( move_cursor_w0 ) { move cursor left} ;
        f$bspace( outfile );
        end { else if ch = then while column( outfile ) > 0}
    else if ch <> carriage_return and ch <> dc3 then begin
      if ch = dc2 { right arrow}
        then stcr( 7, ch::integer ) { read ch from screen} ;
      if eoln( outfile )
        then f$bspace( outfile ) { replace last character} ;
      write( outfile, ch );
      if ch >= ' ' then begin { display character }
        ldcr( 8, ord( ch ) );
        sbo( write_data_strobe_w0 );
        if not eoln( outfile )
          then sbz( move_cursor_w0 ) { move cursor right} ;
        end { if ch >= ' ' then };
      end { else if ch <> carriage_return then } ;
    until ch < ' ' or ch > '';
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 4 of 7)

```
  { blank rest of line}
    sbo( select_word_w0_w1 );
    sbz( enable_cursor_display_w1 ) { turn cursor off} ;
    ldcr( 11, cursor ) { get current cursor address} ;
    sbz( select_word_w0_w1 );
    if eoln( outfile ) then begin
      sbz( move_cursor_w0 ) { move cursor right} ;
      cursor := cursor + 1;
      end { if eoln( outfile )} ;
    ldcr( 8, ord( ' ' ) );
    for i := cursor to last_cursor_on_screen do begin
      sbo( write_data_strobe_w0 ) . strobe ' ' to screen  ;
      sbz( move_cursor_w0 ) { move cursor right} ;
      end { for} ;
    output_line := 24 { include in next roll up} ;
    signal( exclusive_access_to_physical_device );
    if ch = dc3 then rewrite( outfile ) else writeln( outfile );
    until eternity;
  end   { vdt$kwork} ;

  procedure vdt$kexception;
  begin { vdt$kexception}
  if err$class = f$$class_file_error
    and err$reason = f$$reason_channel_abort
    then begin
      err$rset { clear error} ;
      vdt$kwork { invoke work procedure again} ;
      end { then} ;
  { no other exceptions are handled.
    other exceptions will cause process termination with
    error code and no error recovery} ;
  end   { vdt$kexception} ;

begin { vdt$keyboard}
{# priority = level;
   stacksize = vdt$keyboard_stack}
initsemaphore( interrupt, 0 );
externalevent( interrupt, level ) { associate semaphore with
  interrupt level} ;
f$master( outfile ) { establish OUTFILE as channel master} ;
if file_oriented then f$xaccess( outfile );
f$stlength( outfile, screen_line_length ) { set maximum line
  length} ;
f$ulength( outfile ) { allow user's sequential file of shorter
  component length} ;
f$stmode( outfile, writing ) { set mode of keyboard channel} ;
f$createchannel( outfile ) { create channel} ;
signal( partial_completion );
onexception( location( vdt$kexception ) ) { establish exception
  handler} ;
vdt$kwork { invoke work procedure} ;
end   { vdt$keyboard} ;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 5 of 7)

```
process vdt$screen( infile: text; level: interrupt_level );

  procedure vdt$swork;
  var
    ch: char;
    oldcursor: integer;
  begin { vdt$swork}
  crubase( base );
  repeat reset( infile );{ copy files forever}
    while not eof( infile ) do begin
      wait( exclusive_access_to_physical_device );
      sbo( select_word_w0_w1 );
      stcr( 11, oldcursor ) { save original cursor address} ;
      vdt$rollup( output_line );
      while not eoln( infile ) do begin
        read( infile, ch );
        sbz( select_word_w0_w1 );
        ldcr( 7, ord( ch ) ) { load into character buffer} ;
        sbo( set_low_intensity_w0 ) { low intensity} ;
        sbo( write_data_strobe_w0 ) { strobe data to screen} ;
        sbz( move_cursor_w0 ) { move cursor right} ;
        end { while not eoln( infile )} ;
    { restore cursor address to oldcursor}
      sbo( select_word_w0_w1 );
      ldcr( 11, oldcursor );
      signal( exclusive_access_to_physical_device );
      readln( infile );
      end { while not eof( infile ) } ;
    until eternity;
  end   { vdt$swork } ;

  procedure vdt$sexception;
  begin { vdt$sexception }
  if err$class = f$$class_file_error
    and err$reason = f$$reason_channel_abort
    then begin
      err$rset { clear error } ;
      vdt$swork { invoke work procedure again} ;
      end { then } ;
  { no other exceptions are handled.
    other exceptions will cause process termination with
    error code and no error recovery } ;
  end   { vdt$sexception} ;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 6 of 7)

```
    begin { vdt$screen}
    {# priority = level;
       stacksize = vdt$screen_stack}
    f$master( infile ) { establish INFILE as channel master} ;
    if file_oriented then f$xaccess( infile );
    f$stlength( infile, screen_line_length ) { set max line length} ;
    f$ulength( infile ) { allow user's sequential file of shorter
      component length} ;
    f$stmode( infile, reading ) { set mod of screen channel} ;
    f$createchannel( infile )  {create screen channel} ;
    signal( partial_completion );
    onexception( location(vdt$sexception) ) { establish exception
      handler } ;
    vdt$swork { invoke screen work procedure} ;
    end   { vdt$screen} ;

  begin { vdt$supervisor}
  {# priority = level;
     stacksize = vdt$supervisor_stack;
     heapsize = vdt$supervisor_heap}
  crubase( base ) { set cru base to initialize device} ;
  sbz( select_word_w0_w1 ) { select word zero} ;
  sbo( enable_display_w0 ) { enable display} ;
  initsemaphore( partial_completion, 0 );
  initsemaphore( exclusive_access_to_physical_device, 1 );
  start vdt$keyboard( filenamed( name ), level );
    wait( partial_completion );
  start vdt$screen( filenamed( name ), level );
    wait( partial_completion );
  signal( initialization_complete );
  termsemaphore( partial_completion );
  end   { vdt$supervisor} ;

  procedure vdt$( base: cru_address;
    level: interrupt_level;
    name: alfa;
    file_oriented: boolean );
  { initialize a physical device interface system for a 911 VDT }
    var initialization_complete: semaphore;
  begin { vdt$}
  initsemaphore( initialization_complete, 0 );
  start vdt$supervisor(
    base, level, file_oriented, name, initialization_complete );
  wait( initialization_complete );
  termsemaphore( initialization_complete );
  end   { vdt$} ;

begin {$ nullbody}
end.
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM (Sheet 7 of 7)

SECTION XV

DEVELOPMENT SYSTEM OPERATION ON DX/10

15.1 DX/10 OVERVIEW

The Microprocessor Pascal Development System can be executed under a DX10 3.X operating system using the System Command Interpreter (SCI) procs described in this section.

A Microprocessor Pascal Development System session is started by using the Microprocessor Pascal System proc which then displays a menu of available SCI procs which may be invoked. The following is the menu which is displayed:

```
                MICROPROCESSOR PASCAL DEVELOPMENT SYSTEM

                           -- RELEASE 1.0 --

                   SELECT ONE OF THE FOLLOWING COMMANDS

EDIT     - EDIT MODULE            SHOW     - SHOW FILE
COMPILE  - COMPILE SYSTEM         PRINT    - PRINT FILE
DEBUG    - DEBUG SYSTEM           WAIT     - WAIT ON BACKGROUND
EXECUTE  - EXECUTE PROGRAM        PURGE    - PURGE SYNONYMS
SAVE     - SAVE SEGMENT           QUIT     - QUIT SESSION
COLLECT  - COLLECT STAND-ALONE RTS OBJECT MODULES
BATCH    - COMPILE AND SAVE SEGMENT (BACKGROUND)
SCI      - EXECUTE "SCI" COMMAND
```

Each SCI proc will be described in the following paragraphs by giving its function, showing the template for the proc, and explaining each prompt line including the default values for each.

15.2 EDIT

The EDIT proc allows the user to invoke the Source Editor to edit a file. The prompt is shown below:

```
     EXECUTE SOURCE EDITOR
```

When the Source Editor is invoked, it prompts the user for the input file to be edited:

```
     INPUT FILE ACCESS NAME:
```

After the edit session is complete the user will also be prompted for the file to which the edited file is sent.

15.3 COMPILE

The COMPILE proc allows the user to invoke the Compiler to compile an Microprocessor Pascal system. The prompt is shown below:

```
COMPILE A Microprocessor Pascal SYSTEM
          SOURCE:  source file
         LISTING:  listing file
          MEMORY:  stack,heap
      FOREGROUND: YES
            SAVE: NO
```

The SOURCE parameter specifies the file to be compiled and it defaults to the file which was last compiled. The LISTING file specifies the file to which the compilation listing is sent. The MEMORY parameter specifies the amount of stack and heap to be given to the compiler for the compilation. The default if none is given is "4,6" which should be enough to compile a reasonable size system. The FOREGROUND parameter specifies whether the compiler should execute in the foreground mode (the default) or the background mode. If foreground execution is selected, the compilation messages are displayed on the screen and the listing file is displayed after the compilation is complete. If backgound execution is selected, the compilation messages are sent to a file which may be accessed by the synonym MESSAGES. After the background compilation is completed either the listing file or message file may be shown or printed. Finally the SAVE parameter allows the user to save the results of the compilation as a segment. This is only valid in foreground mode, otherwise the BATCH (Section 15.7) should be used. If a value of YES is specified the SAVE command is automatically invoked after the compilation is finished.

15.4 DEBUG

The DEBUG proc allows the user to invoke the Host Debugger so that the user's system may be debugged. The prompt is shown below:

```
DEBUG A Microprocessor Pascal SYSTEM
                LOG FILE:  listing file
    SYSTEM HEAP IN KBYTES:  memory parameter
```

The LOG FILE parameter specifies the file to which a log of the debugging session is sent. Please note that this log only includes a history of the commands input to the debugger and output created by the debugger. It does not include user "messages" or a history of user input or output caused by connecting user files to the terminal. The parameter SYSTEM HEAP IN KBYTES specifies the total amount of memory to be given to the user's system for the debug session. The default value, if none is given, is "5" which should be sufficient for most debug sessions. The messages output by the user's system is displayed on the terminal as they are generated.

All prompts and displays generated by the Host Debugger are displayed on the terminal and Host Debugger commands are also accepted from the terminal.

15.5 EXECUTE

The EXECUTE proc allows the user to execute a conventional Pascal program in a non debug mode. The prompt is shown below:

```
EXECUTE A Microprocessor Pascal PROGRAM
     OUTPUT FILE:  listing file
  FLOATING POINT: NO
     USER MEMORY:  stack and heap
```

The OUTPUT FILE parameter specifies the file to which the default user "output" is sent. After the execution is completed, this listing file shown. If no listing file is given, a default file is used and displayed after the execution. The FLOATING POINT parameter specifies whether floating point (REAL) is needed by the program. The parameter USER MEMORY specifies the combined amount of stack and heap in K bytes to be given to the user's program for its execution. The default value, if none is given, is "1"K bytes. Please note that the amount of stack and heap specified by the concurrent characteristics are ignored and must be specified via the memory parameter described above. The "messages" output by the user's program will be displayed on the terminal as they are generated.

15.6 SAVE

The SAVE proc allows the user to save the interpretive code generated by the compiler as a 9900 object module so that it can be included in a target system. The segment may be included in a host debugging session, conventional program execution, or in a target system. The prompt is shown below:

```
SAVE A Microprocessor Pascal SEGMENT
             LISTING:  listing file
        SEGMENT FILE:  segment file
```

The LISTING parameter specifies the file to which the output generated by the save program is sent. This file is shown after the execution is complete. The SEGMENT FILE parameter specifies the file to which the save segment is to be sent. This file may then be included by a later execution. The user is prompted for input from the terminal.

15.7 BATCH

The BATCH proc combines the COMPILE and the SAVE commands. It allows the user to compile a Microprocessor Pascal system and save the interpretive code generated using only one command. The prompt is shown below:

```
COMPILE AND SAVE A Microprocessor Pascal SEGMENT
         SOURCE FILE:  source file
        LISTING FILE:  listing file
              MEMORY:  stack,heap
        SEGMENT FILE:  segment file
       SEGMENT/DEBUG:  commands
```

The SOURCE FILE parameter specifies the file to be compiled and it defaults to the last source file used. The parameter LISTING FILE is the combined listing generated by this command; both from the compilation and from the save procedure. The MEMORY parameter is the amount of stack and heap to be given to the compiler for this compilation. The default, if none is given, is "4,6" which should be enough to compile a reasonabe size system. The SEGMENT FILE parameter specifies the file to which the save segment is sent. The parameter SEGMENT/DEBUG specifies the commands to be used by the save program. This parameter is a list of two commands, such as "1,YES", the first of which is the segment number and the second is the debug flag.

15.8 COLLECT

The COLLECT proc allows the user to collect the Interpretive Run Time Support modules necessary to support the execution of the user's target system. The prompt is showm below:

```
  COLLECT RUN TIME SUPPORT SEGMENTS
          COMMANDS:  command file
   AMPL INPUT FILE:  ampl name file
RTS OBJECT MODULES:  module file
        KERNEL RTS:  yes/no
```

The COMMANDS parameter specifies the file which contains the commands for the collection program. The default file is "ME". The AMPL INPUT FILE parameter specifies the file to which the values of each user routine name is sent so that they can be used by the AMPL debugger. The RTS OBJECT MODULES parameter specifies the file to which the modules for the Interpretive Run Time Support functions are sent so that they can be included in a target system link edit. The KERNEL RTS parameter specifies whether the "kernel" RTS system is desired or whether the "full" RTS system is desired.

15.9 SHOW

The SHOW proc allows the user to display a file on the terminal for inspection. The prompt is shown below:

```
          SHOW FILE
             FILE PATHNAME:  file name
```

The FILE PATHNAME parameter specifies the file to be displayed which defaults to the last listing file used.

15.10 PRINT

The PRINT proc allows the user to print a compiler listing file on the line printer. The prompt is shown below:

```
          PRINT LISTING FILE
                FILE PATHNAME: <file name>
```

The FILE PATHNAME parameter specifies the file to be printed which defaults to the last listing file used. The compiler listing is automatically split into pages which are numbered by the compiler.

15.11 SCI

The SCI proc allows the user to access standard DX10 commands without exiting the software development session. The prompt is shown below:

```
          EXECUTE SCI COMMAND
                    COMMAND:  SCI command
```

The COMMAND parameter is the SCI command that is to be executed.

This proc should be used with caution since it may cause the Pascal development session to be terminated abnormally. This will happen if the CMD key is used when in a DX10 SCI prompt. When in doubt simply use the Microprocessor Pascal command to resume the Pascal development guide.

15.12 WAIT

The WAIT proc allows the user to suspend the foreground activity until the backgound compilation is completed. The prompt is shown below:

```
  - WAITING FOR BACKGROUND COMPILATION TO COMPLETE -
```

This capability is very similar to the normal WAIT procedure and is provided for completeness.

15.13 PURGE

The PURGE proc allows the user to delete the files and synonyms which were used during the software development session without terminating the current session. The following message is shown:

```
PURGE FILES AND SYNONYMS
```

15.14 QUIT

The QUIT proc allows the user to terminate the software development session and return to the default DX10 command procedures. The following message is shown:

```
QUIT SESSION
```

This allows the default DX10 command procedures to be invoked. To start another software development session, the user must again enter the MPP command.

SECTION XVI

DEVELOPMENT SYSTEM OPERATION ON TX990

16.1 TX990 OVERVIEW

The Microprocessor Pascal Development System may be executed under a TX990 2.3 operating system using the control program described in this section. To execute the Microprocessor Pascal Development System on a machine under TX990, the machine must have the following minimal hardware configuration:

- A 911 or 913 Video Display Terminal (VDT)
- Two floppy drives
- At least 28K words of memory
- TXDS terminal executive development system
- TXSLNK link edit utility
- TXMIRA assembler
- AMPL microprocessor prototyping lab

The Microprocessor Pascal Development System is delivered on three separate floppy diskettes. These discs are described below:

1) A TX990 Sysgen diskette with two different sysgens, one which supports floating point and one which doesn't. They both have support for a 810 line printer. The default system is the floating point version but this may be changed using the UTILITY command. A set file (SF) command of the form SF, :REALS/SYS selects the floating point system and SF, :NOREALS/SYS selects the non-floating point system the next time the system is initialized. There is also a library of parts needed to build a customized system n1eded to support different devices. The sysgen procedure is described in Appendix K.

2) A Development System diskette which includes most of the programs to support software development. These programs include the Source Editor, Compiler, Host Debugger, and other utility programs which are described in in this section.

3) An Interpretive Run Time Support Library diskette which has the collection program and all the parts necessary to support a target system.

The control program is incorporated into the TX990 system and is loaded using the standard loading procedure described below:

1) Insert the TX990 Sysgen diskette (disc 1) into drive one.

2) Sequentially depress the HALT/SIE, RESET, and LOAD switches on the front of the programmer panel.

3) After successful loading of the control program, the following will be displayed:

```
                    MICROPROCESSOR PASCAL SYSTEM -- RELEASE 1.0

                                   YEAR:
```

4) This display indicates that the date and time must be entered. All responses are in integer numbers such as 1979 for the year. After the year is entered, the MONTH, DAY of the month, HOUR of the day (24 hour day), and MINUTE of the hour are requested. Each prompt line should be terminated by the RETURN key.

5) Once the date and time have been entered, the following will be displayed followed by a message to install the system disc:

```
                    MICROPROCESSOR PASCAL SYSTEM -- RELEASE 1.0

                         COMMAND:
```

6) The TX990 Sysgen diskette should be removed from drive one and replaced with the Development System diskette (disc 2). The system is designed so that the system diskette should be in drive one and the user diskette should be in drive two. Therefore all user files should be kept on the user's diskette, because temporary file space is required on the system diskette.

16.2 CONTROL PROGRAM

The control program allows any one of the programs to be loaded and executed. The response to the COMMAND prompt is the name of the program to execute, such as "COMPILE" or "EDIT". After the progam has been loaded the output and input file names are requested. Some programs require only one of these files and others require neither one. The complete prompt is shown below:

```
    MICROPROCESSOR PASCAL SYSTEM -- RELEASE 1.0 --FLOATING POINT
                       SUPPORTED = NO or YES
```

```
                  COMMAND: <command name>
              OUTPUT FILE: <output pathname>
               INPUT FILE: <input pathname>
```

The response to the file prompts is a normal TX pathname. The terminal is named "ME" and is the default if no name is given. The printer is named "LP".

When any program is executing, except the Source Editor, the CMD key (HELP key on 913 VDT) may be used to terminate the program and return to the control program. This should only be used in case the program should be killed, because control cannot be given back to the program.

Each supported program will be described in the following paragraphs by giving its function and showing the command template for the program. Some of the programs generate temporary files on the primary disc but are normally deleted when the program terminates.

16.3 EDIT

The EDIT command allows the user to invoke the Source Editor to edit a file. The program prompts the user directly for the file to be edited, so neither the input or output file is requested in the prompt.

16.4 COMPILE

The COMPILE command allows the user to invoke the Compiler to compile an Microprocessor Pascal System. The prompt is shown below:

```
    COMMAND: COMPILE
OUTPUT FILE:  listing file
 INPUT FILE:  source file
```

The INPUT FILE parameter specifies the file to be compiled. The OUTPUT FILE parameter specifies the file or device to which the compilation listing is sent.

The compiler produces two temporary files on the primary disc, ".PCODE" and ".PROCFIL", which are used use by the "SAVE", "DEBUG", and "EXECUTE" commands and therefore should not be deleted by the user.

16.5 DEBUG

The DEBUG command allows the user to invoke the Host Debugger so that the user's system may be debugged. The prompt is shown below:

```
    COMMAND: DEBUG
OUTPUT FILE:  listing file
```

The OUTPUT FILE parameter specifies the file to which a log of the debugging session is sent and should not be "ME". Please note that this file only contains a log of the commands input to the debugger and the listing produced by the debugger. Any user messages output to the terminal or user files connected to the terminal are not copied to the log.

All prompts and displays generated by the Host Debugger are displayed on the terminal and Host Debugger commands are also accepted from the terminal. The following prompt is used by the Host Debugger so that the amount of memory which is to be available for the complete system may be specified:

```
          Enter system heap size in (K)bytes: 5
```

The default amount of system memory is 5 K bytes. If this amount is not enough, a larger value may be entered followed by a carriage return.

16.6 EXECUTE

The EXECUTE command allows the user to execute a conventional Pascal program in a non debug mode. The prompt is shown below:

```
          COMMAND: EXECUTE
      OUTPUT FILE:  listing file
```

The OUTPUT FILE parameter specifies the file to which the default user output is sent. Please note that the concurrent characteristic specification of the stacksize and heapsize in the user's program is ignored. The total remaining amount of memory is given to the user's program and is used for either stack or heap as needed.

16.7 SAVE

The SAVE command allows the user to save the interpretive code generated by the Compiler as a 9900 object module so that it can be included in a target system. The prompt is shown below:

```
          COMMAND: SAVE
      OUTPUT FILE:  listing file
```

The OUTPUT FILE parameter specifies the file to which the output generated by the save program is sent. The segment file pathname is prompted by the program as follows:

```
          ENTER FILE NAME WHERE SEGMENT IS TO BE STORED:
```

16.8 COLLECT

The COLLECT command allows the user to collect the Interpretive Run Time Support modules necessary to support the execution of the user's target system. The prompt is shown below:

```
          COMMAND: COLLECT
      OUTPUT FILE:  file name
```

The OUTPUT FILE parameter specifies the file where the values of each user routine name is sent so that they can be used by the AMPL debugger. The collection program prompts for commands from the terminal and first of all requests the file to which the modules which are collected are to be sent as follows:

```
          ENTER PATHNAME FOR COLLECTED OBJECT
```

The TX pathname of the file should then be entered followed by a carriage return. Then the program prompts for user segments. After all segments have been entered one at a time, the prompts should be terminated with a empty line and a carriage return. The collection program will then prompt whether the Kernel RTS Library or the full RTS Library is desired as follows:

```
          DO YOU WANT KERNEL RTS SYSTEM? (YES/NO)
```

The phrase YES or NO should then be entered followed by a carriage return. Once this is done, the collection program will start collecting the required modules. This process may require about 5 minutes.

16.9 SHOW

The SHOW command allows the user to display a file on the terminal for inspection. The prompt is shown below:

```
          COMMAND: SHOW

       INPUT FILE:  file name
```

The INPUT FILE parameter specifies the file to be displayed. The user may scroll through the file by using the F1 (forward) or F2 (backward) keys. On the 913 VDT the ROLL UP and ROLL DOWN keys may be used instead of the F1 and F2 keys to scroll through the file. Also the user may input a number which represents the absolute line number to show, and a relative number with a leading plus or minus which indicates to go forward or backward the specified number of lines. The CMD key (HELP key on 913 VDT) must be used to terminate the show file utility.

16.10 COPY

The COPY command allows the user to copy a "TEXT" file to another file or device. The prompt is shown below:

```
         COMMAND: COPY
     OUTPUT FILE:  destination
      INPUT FILE:  source file
```

The OUTPUT FILE parameter specifies the place to which the file is to be copied, and the INPUT FILE parameter specifies the file to be copied.

16.11 UTILITY

The UTILITY command allows the user to perform utility functions on disc files. The prompt is shown below:

```
         COMMAND: UTILITY
     OUTPUT FILE:  listing file
```

The OUTPUT FILE parameter specifies where the listing of the utility program is to be sent.

The commands provided by the utility program are described below:

CF, file name - create a file named file name

CM, file name - compress a file name file name , that is delete all unused space

CN, old file name , new file name - change the name of the file old file name to the name new file name

CP, file name , U or W or D - change the protection of the file file name from its existing protection to unprotected (U), delete protection (D), or write and delete protection (W)

DF, file name - delete the file named file name

DO, file name - change output file destination from its current file or device to the file or device specified by file name . This capability is useful for the map disc command described below.

MD, disc name - produce a map of all the files on a specified disc. Note that only disc device names are allowed and not disc volume names. If no disc name is given, the map of the system disc (drive one) is produced.

SF, file name - make the specified file file name the system file the next time the system is initialized

TI - display current date and time

TE - terminate utility program

The file name parameters used above are valid TX file pathnames. If an invalid command is given a menu of the valid commands if displayed.

An example of the map disc (MD) command is shown below:

```
     DISC I.D.:  disc volume identification
     VOLUME NAME:  name     FREE ADUS:  number of available units

      FILE NAME     TYPE  PROT   ADUS   BLKS

      date and time
```

The file TYPE is either REL for relative record or SEQ for sequential. The file PROTection is either U for unprotected, D for delete protected, or W for write and delete protected. The number of ADUS given specifies the number of allocation units currently used by the file. The number of BLKS given specifies the number of disjoint areas on the disc currently used by the file.

16.12 CONTROL PROGRAM MESSAGES

The control program displays messages at the bottom line of the screen when a program terminates which indicates if it terminated normally or with an error. These messages are described in this section.

NORMAL TERMINATION - This message indicates that the program terminated normally.

INTERPRETER ERROR xx - This message indicates that the program terminated abnormally with an interpreter detected error indicated by "xx". The values of "xx" are described in Appendix E. This message should not be generated by any programs.

NOT ENOUGH MEMORY - This message indicates that the program cannot be loaded into the amount of memory available on the machine or that an overlay cannot be loaded. The only remedy to this problem is to buy more memory for your machine.

I/O ERROR xx - This message indicates that an I/O error was found by the executive when it was trying to open the program file or the output or input file. The error code "xx" indicates the status returned by the I/O service call and may be found in Appendix E.

16.13 MANUAL SYSTEM RESTART

When an error has occurred which causes the control program to abort, the control program can again be given control via the manual system restart procedure described below:

1) Transfer control to the Programmer Panel by pressing the HALT/SIE switch.

2) Press the RESET switch.

3) Press the CLR switch and observe that the indicators of the display are not lit.

4) Press the 8 switch and the 10 switch, setting a number of 00A0 in the display.

5) Press the ENTER PC switch.

6) Press the RUN switch.

7) Press the unlabeled orange key (RESET key on 913 VDT) followed by the ! key.

8. The initialize date and time prompt should be displayed as described in Section 16.1 item 3. If the date and time prompt is not displayed, the TX system may have been destroyed in which case the control program must be reloaded as described in Section 16.1.

SECTION XVII

CONFIGURING AND DEBUGGING TARGET SYSTEMS

17.1 OVERVIEW

Once a user's system has been compiled and debugged on the host system, it is ready to be configured into an object load module for the target machine. During this configuration a simple description of the target machine must be given to identify such things as ROM/RAM addresses and the location of the target machine's restart vector. The user may also include his own assembly language interrupt handlers and system crash handler at this time. The result of this configuration process is a 9900 object load module which may be debugged using AMPL or programmed into ROM for execution on the target system. At the end of this section is an example showing the steps necessary to configure the example in section III into an object load module for a hypothetical target machine.

17.2 CONFIGURING THE MICROPROCESSOR PASCAL SYSTEM INTERPRETIVE RTS FOR TARGET MACHINE

Configuration of a target system requires the user to build a simple specification of the target machine into one of the Interpretive RTS modules, "CONFIG". CONFIG contains the specification of the system's RAM organization and the locations of the system restart and LREX vectors. Usually, the version of CONFIG supplied to the user (see figure 17-1) will not contain the correct specifications for the system being configured and hence will require some of the modifications outlined in Sections 17.2.1 and 17.2.2.

```
     IDT  'CONFIG'
     TITL 'CONFIG:          CONFIGURATION MODULE'
ABSTRACT:
     THIS MODULE DEFINES THE CONFIGURATION OF THE USER'S
     MICROPROCESSOR PASCAL SYSTEM

     DEF  $RAMTB,$RESTA,$LREX
     DEF  INT$WP,IWP$,BAD$WP
--------------------------------------------------------
THIS MODULE SPECIFIES THE FOLLOWING CONFIGURATION
PARAMETERS:
  1) INTERRUPT WORKSPACE (IWP$)
     THIS IS THE WORKSPACE FOR RUN TIME SUPPORT
     INTERRUPT HANDLING.
  2) BAD INTERRUPT AND XOP TRAP WORKSPACE (BAD$WP)
     THIS IS THE WORKSPACE FOR UNSERVICED INTERRUPTS
     AND XOP'S.
  3) INTERPRETER WORKSPACE (INT$WP)
     THIS IS THE WORKSPACE OF THE MICROPROCESSOR PASCAL
     CODE INTERPRETER.
     IT MUST BE THE LAST WORKSPACE ALLOCATED, BECAUSE
     SYSTEM STRUCTURES ARE ALLOCATED IN THE MEMORY
     FOLLOWING THIS WORKSPACE.
  4) RAM CONFIGURATION ($RAMTB)
     THIS IS THE ADDRESS OF A LIST OF PAIRS OF
     LENGTH-IN-BYTES, START-ADDRESS
     OF VALID RAM TO BE USED BY RTS.
     THIS LIST MUST BE TERMINATED BY AN ENTRY WITH
     LENGTH-IN-BYTES = 0.
  5) RESTART BLWP VECTOR ADDRESS ($RESTA)
     THIS IS THE ADDRESS OF THE RESTART BLWP VECTOR.
     THIS WILL BE ZERO (LEVEL ZERO INTERRUPT BLWP)
     OR >FFFC (LREX INSTRUCTION BLWP)
     OR THE ADDRESS OF ANY USER DEFINED BLWP VECTOR
     FOR DOING A SYSTEM 'RESTART'.
  6) LREX BLWP ADDRESS VECTOR ($LREX)
     THIS IS THE ADDRESS OF THE LREX BLWP VECTOR TO BE
     COPIED INTO HIGH MEMORY.  IF THERE IS TO BE NO
     LREX BLWP OR HIGH MEMORY IS ROM, THEN THIS
     SHOULD BE ZERO.
--------------------------------------------------------
```

FIGURE 17-1. CONFIG. (Sheet 1 of 2)

```
-----------------------------------------------------------
   THE ENTRIES FOR THIS MODULE SPECIFY:
     1) RAM FROM A000 TO AFFF,
        RAM FROM C000 TO CFFF,
        RAM FROM FF00 TO FFFF
     2) RESTART IS THE SAME AS A LEVEL ZERO INTERRUPT
     3) THERE IS NO LREX BLWP VECTOR
-----------------------------------------------------------
STATIC EQU  >A000
       DORG STATIC
IWP$   BSS  >20
BAD$WP BSS  >20
INT$WP EQU  $                    THIS WORKSPACE MUST BE LAST!
-----------------------------------------------------------
       PSEG
$RAMTB DATA >B000-INT$WP,INT$WP
       DATA >1000,>C000
       DATA >100,>FF00
       DATA 0
$RESTA EQU  0                    RESTART IS LEVEL 0 INTERRUPT
$LREX  EQU  0                    NO LREX BLWP VECTOR
       END
```

Figure 17-1. CONFIG. (Sheet 2 of 2)

17.2.1 Specification of RAM Locations

Configuration of the target system requires a description of the system's RAM (the RAM table) to be inserted into the module CONFIG. Often, no additional work is required to specify the target system; the user simply fills in the RAM table that is contained in CONFIG with the addresses and sizes of the target machine's RAM memory segments. The RAM table is a list of data value pairs. The first value of the pair is the length in bytes of the RAM segment and the second value is the starting address of that segment. The list is terminated with an entry that has a zero specified for the length. The RAM table for a target system with RAM from Hex addresses A000 to AFFF, C000 to C7FF, D000 to D0FF, and FF00 to FFFF looks like Figure 17-2.

Figure 17-3 is a copy of the source for CONFIG (without some of its documentation) as it would appear after being modified for a target machine with the same RAM segments as used in the previous example.

```
$RAMTB DATA >1000,>A000      A000 - AFFF
       DATA >800,>C000       C000 - C7FF
       DATA >100,>D000       D000 - D0FF
       DATA >100,>FF00       FF00 - FFFF
       DATA 0
```

FIGURE 17-2. SIMPLE RAM TABLE

```
         IDT  'CONFIG'
         TITL 'CONFIG:          CONFIGURATION MODULE'
*
*        THIS MODULE DEFINES THE CONFIGURATION OF THE USER'S
*        MICROTIP SYSTEM.
*
         DEF  $RAMTB,$RESTA,$LREX
         DEF  INT$WP,IWP$,BAD$WP
STATIC   EQU  >A000
         DORG STATIC
IWP$     BSS  >20
BAD$WP   BSS  >20
INT$WP   EQU  $                 THIS WORKSPACE MUST BE LAST!
         PSEG
-------------------------------------
$RAMTB   DATA >B000-INT$WP,INT$WP
         DATA >800,>C000                    RAM
         DATA >100,>D000
         DATA >100,>FF00                    TABLE
         DATA 0
-------------------------------------
$RESTA   EQU  0
$LREX    EQU  0
         END
```

FIGURE 17-3. USE OF RAM TABLE IN CONFIG

Note that the RAM table in figure 17-3 has been changed. The memory specified in the RAM table is linked together to form the system data structures, for example the system heap. Actual RAM not included in the RAM table will not be used for these system data structures. In the example of figure 17-3 the first >40 bytes of real RAM, locations >A000 through >A03F, are not included in the RAM table. Instead, these locations are used for the allocation of two workspaces, "IWP$" and "BAD$WP". Although not shown in this example, additional real RAM could have been excluded from the RAM table for use by the user's assembly language modules. The symbol "INT$WP" should always mark the beginning of the RAM specified in the RAM table, and hence the begining of the Interpretive RTS data structures.

17.2.2 Specification of Restart and LREX Vectors Locations

The symbol $RESTA within CONFIG, should be equated to the address of a BLWP vector to be used for a system restart. If $RESTA is zero, then a system restart is the same as a level 0 interrupt. If $RESTA is >FFFC, then a system restart is the same as an LREX instruction. The LREX vector is often used for the restart in systems that have real level 0 interrupts. Finally, $RESTA may be a value other than 0 or >FFFC, in which case a restart is distinct from level 0 interrupts and LREX instructions.

In some systems the LREX instruction is used for restarting or reloading the system. The LREX transfer vector is located in high memory at Hex locations >FFFC through >FFFF. If these locations are in RAM then the Executive Run Time Support must load these locations with the proper values. A user specifies which values should be copied into high memory by creating a copy of the transfer vector and equating this copy's address to the symbol $LREX in CONFIG. If high memory is ROM or if there is to be no LREX vector, then $LREX should be left zero.

17.2.3 Allocation of Workspaces in CONFIG

In addition to the RAM table and locations of the restart and LREX vectors, CONFIG contains several workspaces. The supplied version of CONFIG, shown in Figure 17-1, defines three workspaces. IWP$ is the interrupt workspace. This is the workspace used for interrupt handling in the RTS. BAD$WP is the bad interrupt and XOP workspace. This is the workspace which is usually specified in the transfer vectors of all unimplemented interrupts and XOPS (See Section 17.3). INT$WP is the Microprocessor Pascal System interpreter workspace. This workspace occupies the first 16 words of RAM specified in the RAM table. All other workspaces defined within CONFIG (including IWP$ and BAD$WP) should be allocated out of RAM space which has been excluded from the RAM table.

17.2.4 Example

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).

Figure 17-4 is a version of CONFIG (without most of its documentation) that might be used for this system.

```
        IDT  'CONFIG'
        TITL 'CONFIG:           CONFIGURATION MODULE'

        THIS MODULE DEFINES THE CONFIGURATION OF THE USER'S
        MICROTIP SYSTEM.

        DEF  $RAMTB,$RESTA,$LREX
        DEF  INT$WP,IWP$,BAD$WP
        REF  BGN$PC                     <--- ADDED
 STATIC EQU  >B000                      <--- CHANGED
        DORG STATIC
 IWP$   BSS  >20
 BAD$WP BSS  >20
 BGN$WP BSS  >20                        <--- ADDED
 INT$WP EQU  $
        PSEG
 $RAMTB DATA >C000-INT$WP,INT$WP        <--- CHANGED
        DATA >1000,>D000
        DATA 0
 $RESTA DATA BGN$WP,BGN$PC              <--- ADDED
 $LREX  EQU  0                     NO LREX BLWP VECTOR
        END
```

FIGURE 17-4. CONFIG WITH USER MODIFICATIONS

The RAM table of figure 17-4 reflects the two RAM memory segments, but notice that the ROM memory segment addresses have no effect on CONFIG. The workspace BGN$WP has been added to CONFIG along with the restart vector $RESTA.

17.3 USER CUSTOMIZATION OF THE INTERPRETIVE RUN TIME SUPPORT

The Interpretive RTS may be customized by the user in two ways: assembly language interrupt handlers, and user-coded crash routines. Both of these customizations involve additions and changes to the module "USERINIT" (the user initialization module). USERINIT as it it supplied to the user is shown in Figure 17-5.

```
          IDT   'USERINIT'
  ROUTINE LIST:
          RSET$, SYS$CR
  COPY MODULES:
          NONE
  MACRO DEFINITIONS:
          NONE
  EXTERNAL ROUTINES:
          SEG0-SEG63
  EXTERNAL DATA:
          $LREX, $RAMTB, $RESTA, BAD$WP, INT$WP, IWP$
  MODULE CONSTANTS:
R0        EQU   0
R1        EQU   1
R2        EQU   2
R12       EQU   12
          DEF   SYS$CR,IN$PC,$SEGTB,RSET$
          REF   INT$WP,IWP$,BAD$WP
          REF   $RAMTB,$EXEC,$RESTA,$LREX
          REF   SEG0,SEG1,SEG63   REQUIRED SEGMENTS
          SREF  SEG2,SEG3,SEG4,SEG5,SEG6,SEG7
          SREF  SEG8,SEG9,SEG10,SEG11,SEG12,SEG13,SEG14,SEG15
          SREF  SEG16,SEG17,SEG18,SEG19,SEG20,SEG21,SEG22,SEG23
          SREF  SEG24,SEG25,SEG26,SEG27,SEG28,SEG29,SEG30,SEG31
          SREF  SEG32,SEG33,SEG34,SEG35,SEG36,SEG37,SEG38,SEG39
          SREF  SEG40,SEG41,SEG42,SEG43,SEG44,SEG45,SEG46,SEG47
          SREF  SEG48,SEG49,SEG50,SEG51,SEG52,SEG53,SEG54,SEG55
          SREF  SEG56,SEG57,SEG58,SEG59,SEG60,SEG61,SEG62
  MODULE VARIABLES:
          PSEG
```

FIGURE 17-5. USERINIT. (Sheet 1 of 7)

```
----------------------------------------------------------
    IF THIS MODULE IS LOADED AT ADDRESS ZERO, THEN
    THE TRAP VECTORS (WHICH FOLLOW) OF THE MACHINE
    ARE ALREADY INITIALIZED.  OTHERWISE, MICROTIP
    INITIALIZATION MOVES THE FOLLOWING DATA TO
    ABSOLUTE ADDRESS ZERO.
----------------------------------------------------------
TRAPS  EQU  $                    CONFIGURATION OF TRAP VECTORS
       DATA INT$WP,RSET$         LEVEL 0
       DATA IWP$,I1PC            LEVEL 1
       DATA IWP$,I2PC            LEVEL 2
       DATA IWP$,I3PC            LEVEL 3
       DATA IWP$,I4PC            LEVEL 4
       DATA IWP$,I5PC            LEVEL 5
       DATA IWP$,I6PC            LEVEL 6
       DATA IWP$,I7PC            LEVEL 7
       DATA IWP$,I8PC            LEVEL 8
       DATA IWP$,I9PC            LEVEL 9
       DATA IWP$,I10PC           LEVEL 10
       DATA IWP$,I11PC           LEVEL 11
       DATA IWP$,I12PC           LEVEL 12
       DATA IWP$,I13PC           LEVEL 13
       DATA IWP$,I14PC           LEVEL 14
       DATA IWP$,I15PC           LEVEL 15
       DATA BAD$WP,BAD$PC        XOP 0
       DATA BAD$WP,BAD$PC        XOP 1
       DATA BAD$WP,BAD$PC        XOP 2
       DATA BAD$WP,BAD$PC        XOP 3
       DATA BAD$WP,BAD$PC        XOP 4
       DATA BAD$WP,BAD$PC        XOP 5
       DATA BAD$WP,BAD$PC        XOP 6
       DATA BAD$WP,BAD$PC        XOP 7
       DATA BAD$WP,BAD$PC        XOP 8
       DATA BAD$WP,BAD$PC        XOP 9
       DATA BAD$WP,BAD$PC        XOP 10
       DATA BAD$WP,BAD$PC        XOP 11
       DATA BAD$WP,BAD$PC        XOP 12
       DATA BAD$WP,BAD$PC        XOP 13
       DATA BAD$WP,BAD$PC        XOP 14
       DATA BAD$WP,BAD$PC        XOP 15
----------------------------------------------------------
 PANEL  BSS  >20                 FRONT PANEL WORKSPACE GOES
                                 AT ADDRESS >80 IF NECESSARY
----------------------------------------------------------
       TITL 'RSET$:              SYSTEM STARTUP CODE'
       PAGE
```

FIGURE 17-5. USERINIT. (Sheet 2 of 7)

```
  ABSTRACT:
       THIS MODULE IS PROVIDED BY THE USER.  IT IS
       POSITION-DEPENDENT BECAUSE TRAP VECTORS WHICH
       CAN BE IN ROM MUST REFERENCE STATICALLY LOCATED
       CODE AND DATA SPACE AT LOCATIONS DEFINED HEREIN.
       NOTICE HOWEVER THAT ALTHOUGH THIS MODULE DEF'S
       SYMBOLS, ALL CODE OF MICROPROCESSOR PASCAL
       INTERPRETIVE CODE RUN-TIME SUPPORT IS POSITION-INDEPENDENT
       AND DOES NOT DEPEND ON ITS LOCATION OR THE LOCATION
       OF USER-WRITTEN CODE.  THE ONLY REF'S IN THIS
       MODULE ARE TO SEGMENT LOCATIONS, TO THE ENTRY
       POINT OF THE INTERPRETER, AND TO USER-DEFINED
       CONFIGURATION INFORMATION.  THE USER MAY PLACE
       MICROPROCESSOR PASCAL SEGMENTS (WHICH ARE POSITION-
       INDEPENDENT ANYWHERE IN CODE SPACE, AND MUST BUILD A 64-ENTRY
       SEGMENT TABLE WHICH IS GIVEN BELOW.  ADDITIONALLY,
       THIS ROUTINE INITIALIZES ALL OF RAM.
  CALLING SEQUENCE:
       B    @RSET$
  EXCEPTIONS AND CONDITIONS:
       THE FIRST ENTRY IN THE RAM TABLE MUST BE LARGE
            ENOUGH FOR ALL OF SYSTEM INITIALIZATION
            (ABOUT 1300 BYTES (DECIMAL))
  EXTERNALS LIST:
       ROUTINES: NONE
       VARIABLES: NONE
       OTHER: EXECPC, SEG0-SEG63
  LOCAL DATA:
VAL    DATA >FF00                FORCE AN ADDRESSING ERROR
ZERO   DATA 0
ALTZER DATA >5555
ALTONE DATA >AAAA
  ENTRY POINT:
RSET$  EQU  $                    RSET VECTOR PC POINTS HERE
       RSET                      USER IS RESPONSIBLE TO RESET
                                 THE MACHINE
       LWPI INT$WP               ESTABLISH WORKSPACE IF BRANCH
                                 TO RSET$ INSTEAD OF BLWP
```

FIGURE 17-5. USERINIT. (Sheet 3 of 7)

```
-------------------------------------------------------------
  INITIALIZE MEMORY TO GARBAGE

    (PRIMARILY FOR DEBUGGING:  ALL THAT IS NEEDED IS FOR
    THE USER TO SET UP THE RAMTABLE BY EITHER CODE OR
    DATA STATEMENTS.)
-------------------------------------------------------------
       LI   R2,$RAMTB
       MOV  @2(R2),R0             R0 := START ADDRESS
       MOV  *R2,R1                R1 := LENGTH
       A    R0,R1                 R1 := ADDR FIRST NON-RAM WORD
       LI   R0,INT$WP
       AI   R0,>20                <============================
ZAP    EQU  $
       MOV  @ALTZER,*R0
       C    @ALTZER,*R0
       JNE  EOB
       MOV  @ALTONE,*R0
       C    @ALTONE,*R0
       JNE  EOB
       MOV  @VAL,*R0+             ZAP RAM WORD
       C    R0,R1
       JL   ZAP
       JMP  NEXTAB

EOB    EQU  $                     RAM REALLY ISN'T
       MOV  R0,R1                 R1 := ADDRESS OF BAD RAM
       LI   R0,6                  CRASH CODE 6 = ROM/RAM ERROR
       B    @SYS$CR               CRASH

NEXTAB EQU  $
       AI   R2,4
       C    *R2,@ZERO             END OF RAM TABLE?
       JEQ  OUT                   IF YES, GO ON
       MOV  @2(R2),R0             R0 := START ADDRESS
       MOV  *R2,R1                R1 := LENGTH
       A    R0,R1                 R1 := ADDR FIRST NON-RAM WORD
       JMP  ZAP

OUT    EQU  $
-------------------------------------------------------------
  END OF RAM TABLE AND MEMORY INITIALIZATION
-------------------------------------------------------------
       LI   R0,TRAPS              PASS ADDRESS OF TRAPS
*                                   CONFIGURATION
       LI   R12,SEG63
       MOV  *R12,R12
       AI   R12,SEG63
       BL   *R12                  BRANCH TO RTS
*                                   INITIALIZATION
*                                   (SEG63, ENTRY=0)
```

FIGURE 17-5. USERINIT. (Sheet 4 of 7)

```
-----------------------------------------------------------
   IMMEDIATELY FOLLOWING THE BL MUST BE:
     1) THE ADDRESS OF THE RAM/ROM TABLE
     2) THE ADDRESS OF THE INTERPRETER ENTRY POINT
     3) THE ADDRESS OF CRASH CODE
     4) THE ADDRESS OF THE RE$START BLWP VECTOR
     5) THE ADDRESS OF THE LREX BLWP VECTOR (IF RAM IN
        HIGH MEMORY, ELSE A ZERO ENTRY)
     6) THE SEGMENT TABLE (64 ENTRIES)
-----------------------------------------------------------

       DATA $RAMTB                 1) THE ADDRESS OF THE RAM/ROM
                                      TABLE

       DATA $EXEC                  2) THE ADDRESS OF INTERPRETER
                                      ENTRY POINT

       DATA SYS$CR                 3) ADDRESS OF CRASH CODE

       DATA $RESTA                 4) THE ADDRESS OF RE$START
                                      BLWP VECTOR

       DATA $LREX                  5) THE ADDRESS OF THE LREX
                                      BLWP VECTOR (IF RAM IN
                                      HIGH MEMORY, ELSE 0)

$SEGTB EQU  $                      6) THE 64 ENTRY SEGMENT TABLE
       DATA SEG0,SEG1,SEG2,SEG3,SEG4,SEG5,SEG6,SEG7
       DATA SEG8,SEG9,SEG10,SEG11,SEG12,SEG13,SEG14,SEG15
       DATA SEG16,SEG17,SEG18,SEG19,SEG20,SEG21,SEG22,SEG23
       DATA SEG24,SEG25,SEG26,SEG27,SEG28,SEG29,SEG30,SEG31
       DATA SEG32,SEG33,SEG34,SEG35,SEG36,SEG37,SEG38,SEG39
       DATA SEG40,SEG41,SEG42,SEG43,SEG44,SEG45,SEG46,SEG47
       DATA SEG48,SEG49,SEG50,SEG51,SEG52,SEG53,SEG54,SEG55
       DATA SEG56,SEG57,SEG58,SEG59,SEG60,SEG61,SEG62,SEG63
                                   THE USER'S INITIAL PROCESS
                                   MUST BE IN SEGMENT 1.
                                   THE USER CAN USE ADDITIONAL
                                   SEGMENTS AS NEEDED.

                                   UNUSED SEGMENT ENTRIES
                                   ARE ZERO.

-----------------------------------------------------------
I1PC   EQU  $                      LEVEL 1 INTERRUPT
       LIMI 0
       LI   R0,1
       JMP  IN$PC
```

FIGURE 17-5. USERINIT. (Sheet 5 of 7)

```
I2PC    EQU  $                  LEVEL 2 INTERRUPT
        LIMI 0
        LI   R0,2
        JMP  IN$PC
I3PC    EQU  $                  LEVEL 3 INTERRUPT
        LIMI 0
        LI   R0,3
        JMP  IN$PC
I4PC    EQU  $                  LEVEL 4 INTERRUPT
        LIMI 0
        LI   R0,4
        JMP  IN$PC
I5PC    EQU  $                  LEVEL 5 INTERRUPT
        LIMI 0
        LI   R0,5
        JMP  IN$PC
I6PC    EQU  $                  LEVEL 6 INTERRUPT
        LIMI 0
        LI   R0,6
        JMP  IN$PC
I7PC    EQU  $                  LEVEL 7 INTERRUPT
        LIMI 0
        LI   R0,7
        JMP  IN$PC
I8PC    EQU  $                  LEVEL 8 INTERRUPT
        LIMI 0
        LI   R0,8
        JMP  IN$PC
I9PC    EQU  $                  LEVEL 9 INTERRUPT
        LIMI 0
        LI   R0,9
        JMP  IN$PC
I10PC   EQU  $                  LEVEL 10 INTERRUPT
        LIMI 0
        LI   R0,10
        JMP  IN$PC
I11PC   EQU  $                  LEVEL 11 INTERRUPT
        LIMI 0
        LI   R0,11
        JMP  IN$PC
I12PC   EQU  $                  LEVEL 12 INTERRUPT
        LIMI 0
        LI   R0,12
        JMP  IN$PC
I13PC   EQU  $                  LEVEL 13 INTERRUPT
        LIMI 0
        LI   R0,13
        JMP  IN$PC
```

FIGURE 17-5. USERINIT. (Sheet 6 of 7)

```
I14PC  EQU  $                 LEVEL 14 INTERRUPT
       LIMI 0
       LI   R0,14
       JMP  IN$PC
I15PC  EQU  $                 LEVEL 15 INTERRUPT
       LIMI 0
       LI   R0,15
       JMP  IN$PC
IN$PC  EQU  $                 SERVICE INTERRUPT

  PASS INTERRUPT LEVEL IN R0
  PASS INTERPRETER'S WORKSPACE IN R1
  PASS ADDRESS OF SEG63, PROCEDURE 1 IN R2

       LI   R1,INT$WP
       LI   R2,SEG63
       MOV  @2(R2),R2
       AI   R2,SEG63
       B    *R2               BRANCH TO RTS
                                (SEG63, ENTRY 1)
----------------------------------------------------------
  CODE FOR STANDARD TM990 SYSTEM
----------------------------------------------------------
BAD$PC EQU  $
       JMP  SYS$CR
*
SYS$CR EQU  $
       LIMI 0
       IDLE
       JMP  SYS$CR
*
       END  RSET$            ENTRY POINT OF STANDARD SYSTEM
```

FIGURE 17-5. USERINIT. (Sheet 7 of 7)

17.3.1 Assembly Language Interrupt Handlers

USERINIT contains the interrupt transfer vectors. Each transfer vector refers to a label within USERINIT of the form IxPC where x is a numeral between 1 and 15. The code labeled with IxPC is the code that will be executed when a interrupt at level x occurs. For example, a level 3 interrupt will cause the program counter to advance to I3PC. Figure 17-6 contains the code that would normally be executed when a level 3 interrupt occurs in an unmodified system.

```
I3PC    EQU     $
        LIMI    0
        LI      R0,3
        JMP     IN$PC
```

FIGURE 17-6. STANDARD CODE FOR LEVEL 3 INTERRUPT

Since workspace IWP$ is used by more than one interrupt handler, each handler must begin with "LIMI 0" to assure exclusive access to IWP$.

Loading R0 with 3 and jumping to IN$PC causes the Interpretive RTS to generate a signal for the Microprocessor Pascal System semaphore associated with level 3 interrupts.(This association was established with a call to EXTERNALEVENT or ALTEXTERNALEVENT in the user's part of the system.)

It is possible to handle the level x interrupt completely at IxPC without attempting to signal an associated semaphore. This might be done to avoid the overhead of leaving assembly language to handle the interrupt. Figure 17-7 is a skeleton of an assembly language only handler for level 3 interrupts.

In 990/4 or 990/10 systems the 8.33 millisecond clock generates level 5 interrupts. An example of an assembly language handler for these interrupts is given in Figure 17-8.

This clock interrupt handler increments a CSEG counter, TIME, which other assembly language routines in the system could easily access.

```
I3PC    EQU     $
        LIMI    0
        .
        .       PROCESS LEVEL 3 INTERRUPT
        .
        RTWP        RETURN FROM INTERRUPT HANDLER
```

FIGURE 17-7. SKELETON OF ASSEMBLY LANGUAGE INTERRUPT HANDLER

```
        CSEG    'TIME'
TIME    DATA    0
        PSEG
I5PC    EQU     $
        CKOF
        CKON
        INC     @TIME
        RTWP        RETURN FROM INTERRUPT HANDLER
```

FIGURE 17-8. ASSEMBLY LANGUAGE CLOCK INTERRUPT HANDLER

```
               CSEG   'COUNT'
        COUNT  DATA   0
               PSEG
        TEN    DATA   10
        I5PC   EQU    $
               LIMI   0
               CKOF
               CKON
               DEC    @COUNT
               JEQ    SIGNAL
               JLT    SIGNAL
               RTWP                  RETURN FROM INTERRUPT HANDLER
        SIGNAL MOV    @TEN,@COUNT    RESET COUNTER
               LI     R0,5
               JMP    IN$PC          SIGNAL AND RETURN
```

FIGURE 17-9. DIVIDE BY 10 CLOCK INTERRUPT HANDLER

A level x interrupt may be handled by a combination of assembly language and Microprocessor Pascal System code. The assembly language may use any registers except R13, R14, and R15. R0, R1, and R2 must be set to the interrupt level, interrupt workspace, and entry point of the RTS interrupt handler, respectively, at the time of the call to RTS. (The common code at IN$PC should be used.) RTS uses registers R0 through R6 and R11, but leaves registers R0, R1, and R2 unchanged.

Figure 17-9 is a third example of a clock interrupt handler. This handler is activated on every hardware clock interrupt, but only generates a signal to the Microprocessor Pascal System semaphore associated with the clock interrupts every tenth interrupt. This provides the rest of the system with an effective clock period of 83.33 millisecond. USERINIT might be customized in this way to reduce the swapping overhead in a system which did a swap on every "clock" interrupt.

If interrupts are to be masked in any of the code being inserted in USERINIT, use only a LIMI 0. Interrupts must be masked in the clock interrupt handler since it sha es its workspace with other USERINIT interrupt handlers. No LIMI 0 would have to be required if the clock handler were given its own workspace.

17.3.2 Crash Routine

The label SYS$CR marks the point in USERINIT to which control will transfer if a system crash occurs. The code which is provided in USERINIT (shown in Figure 17-10) idles with interrupts masked. In a customized system a system crash might cause an automatic restart, the activation of an alarm, or the transmission of a message.

```
SYS$CR EQU    $
       LIMI   0
       IDLE
       JMP    SYS$CR
```

FIGURE 17-10. STANDARD CRASH CODE

More elaborate crash routines can be developed. Figure 17-11 is a routine which will flash the front panel lights of a 990/4 or 990/10 with the crash code and exception codes when the system crashes.

As when adding interrupt code to USERINIT, use only a LIMI 0 if interrupts are to be masked in the crash routine.

```
VECTOR  DATA NEWWP,LIGHTS
NEWWP   BSS  >20
SHOW    LI   R12,>1FE0
        LDCR R1,8
        SWPB R1
        LDCR R1,8
        LI   R0,50000          SPIN FOR ONE SECOND
WAIT    SRA  R1,14             40 CLOCKS
        DEC  R0                10 CLOCKS
        JNE  WAIT              10 CLOCKS
        B    *R11
        DEF  SYS$CR
SYS$CR  EQU  $                 CRASH$ POINT
        LIMI 0
        BLWP @VECTOR
LIGHTS  SETO R1                FLASH FRONT PANEL
        BL   @SHOW
        MOV  @2*R0(R13),R1     SHOW CRASH CODE
        BL   @SHOW
        SETO R1                FLASH FRONT PANEL
        BL   @SHOW
        LI   R1,INT$WP         WORKSPACE OF INTERPRETER
        MOV  @2*R8(R1),R1
        MOV  @>3E(R1),R1       SHOW EXCEPTION CODES
*                              OF CURRENT PROCESS
        BL   @SHOW
        JMP  LIGHTS
```

FIGURE 17-11. ELABORATE CRASH ROUTINE

17.4 ASSEMBLY LANGUAGE CODING CONVENTIONS

It is possible to write assembly language routines which are callable from Microprocessor Pascal by following the conventions outlined in this section. A segment may consist of assembly language routines which may be declared and called as any other external procedures or functions. Figure 17-12 on the following page is an example of an assembly language segment.

```
       IDT  'SEGn'      n = segment number

PRCS   EQU  R8          process record pointer
ENTRY  EQU  R12         my routine address
SP     EQU  R14         my first parameter
CALLER EQU  R15         my caller's stack frame

MKSIZ  EQU  -14         administration area size
SEGCOD EQU  ((segment_number)*2 + 1)

       DEF  rout1,rout2
rout1  EQU  (0*256 + SEGCOD)
rout2  EQU  (1*256 + SEGCOD)

       DEF  SEGn        segment dictionary
SEGn   DATA entry1-SEGn   displacement to first entry point
       DATA entry2-SEGn   displacement to second entry point

LIT    DATA value       local literal

entry1 DATA 0           assembly language flag
       DATA parms*2     parameter size (bytes)

   code for routine

   use local literal

       MOV  @LIT-entry1(ENTRY),R1

       B    *R11        return

entry2 DATA 0           assembly language flag
       DATA parms*2     parameter size (bytes)

   code for routine - FUNCTION

       AI   SP,MKSIZ    administration area for function
       MOV  result,*SP+  return function result
       B    *R11        return

       END
```

FIGURE 17-12. ASSEMBLY LANGUAGE SEGMENT

Each assembly language routine in the segment should have a DEF for the routine name. The symbol should be defined by an EQU statement which defines the routine number and segment number. The value should have the form:

```
        routine number * 256 + SEGCOD
```

where SEGCOD is segment number * 2 + 1.

The first routine in a segment is number 0 and so on.

The segment dictionary must be defined by a symbol of the form "SEGn", where "n" is the number of the segment. The dictionary should be labeled by the segment name, and each entry in the dictionary should be of the form:

```
        DATA  routine-SEGn
```

where "routine" is the label of the routine preamble. The routine preamble consists of two words of data, the first of which must be zero (0) which indicates that the routine is an assembly language routine, and the second word indicates the size of the parameter area for the routine in bytes.

The registers available for use by the assembly language routine are R0 through R6. Registers R8 through R15 may be used if they are saved and restored by the routine. Register R7 is used for interrupts and should not be changed by the user. Some of the registers used by the Microprocessor Pascal System interpreter may be useful by the routine. The dedicated registers are described below:

R7 - address of next instruction handler
R8 - address of the process record
R9 - address of the current work space (register set)
R10 - address of next available work space
R11 - return address
R12 - assembly language routine entry point address
R13 - caller's program counter
R14 - address of my parameters
R15 - address of my caller's stack frame

The parameters of the routine may be accessed via the SP (R14) register. The first parameter may be referenced via *SP, the second parameter may be referenced via @2(SP), and so on for as many parameters as were passed. If the assembly language routine is a function, the administration area for the function must be discarded by decrementing SP by the administration area size (14 bytes). The function result must then be pushed on the stack using *SP+ references.

An assembly language routine should be coded so that it is position independent. This may be done by using "jump" instructions rather than "branch" instructions, and by referencing local data or literals as follows:

```
@literal-entry(R12)
```

When writing assembly language, masking of interrupts should be done with a LIMI 0. Do not use any other interrupt masks and do not adjust the mask via an RTWP instruction. If a BLWP instruction is used, the workspace pointer, program counter, and status register of the caller must remain in R13 - R15 of the callee's workspace.

17.5 PRODUCING AN INTERPRETIVE RTS LOAD MODULE

To run a stand-alone system, an Interpretive RTS load module must be created. This process is called configuring a system, and involves the collection and connection of all code in the system with other user code and Run Time Support code. The major aspects of creating an Interpretive RTS load module are the compilation and saving of source modules, the configuration of Interpretive RTS Library segment(s), and link editing. The load module thus created may then be debugged using AMPL or programmed into ROM. Producing a load module for a stand-alone system consists of the following steps (TX and DX systems):

1) Compile and save all user segments. In the save phase of segment preparation, each segment must be assigned a unique number from 1 to 50 (The Interpretive RTS must use segments 0 and 51 through 63). Execution always begins in the SYSTEM (or PROGRAM in the case of a conventional PASCAL program) which occurs in segment 1.

2) Collect routines from the Interpretive RTS Library. The set of user-written segments to form the system must be scanned by the Run Time Support collector to determine which Run Time support routines are used and to produce segment(s) of needed Run Time Support routines.

3) Modify the 'CONFIG' module to be consistent with the user's system. This allows the user to specify the addresses at which RAM/ROM boundaries and restart and LREX vectors occur. This module must be edited and assembled.

4) Optionally modify the 'USERINIT' module. If special customization is required, the 'USERINIT' module should be edited and reassembled. For further information on customization, see Section 17.3. If a standard version of 'USERINIT' is to be used, then only an assembly need be done.

5) Create a link edit control file to 'INCLUDE' the user segments, the segments of Run Time Support routines, the user initialization module, and the user configuration module.

6) Link edit using the above control file to create a linked object module.

17.5.1 Configuration of Interpretive RTS Segment(s)

The user has the option of using the Kernel RTS system (with minimal run-time support) or the "full" RTS system. The Run Time Support collector asks for the pathname of a file containing a list of user segments, the pathname of an AMPL input file, the pathname of a file into which the Interpretive RTS system code is to be collected, and the system desired. The control file (the list of user segments) is simply a list of all Microprocessor Pascal System segments by pathname which are in the final system, one pathname per line, terminated by end of file. Assembly language segments (if any) need not be included for the collector. From this information, a file is created which contains segment(s) of required routines of Run Time Support and required modules of the interpreter, which is to be included later at link edit time.

When debugging, the user may not need to collect the Interpretive RTS code every time a segment has been recompiled and resaved. If no calls have been added to previously uncalled RTS routines, and if one of the data types REAL, LONGINT, SET, or packed array of characters has been added; then the user does not need to re-collect RTS. As an example, reordering statements within a segment will not require collecting RTS again; the output from the last collect will still be configured correctly. However, if a call to a procedure or function of RTS is added, unless the routine was already being called, a collect of RTS must be redone.

The choice of using the "full" RTS Library or the kernel system is primarily dependent upon the features needed and the size requirements of the user. "Full" RTS Library implements all features but in general requires more space than the Kernel RTS Library. In each case, the user must decide which system best fits his requirements.

17.5.1.1 "Full" RTS Library. The "Full" RTS Library is the one which has been described in this document, with all features implemented. When producing a load module using the "Full" RTS Library, the Run Time Support collector creates two support segments which must be included. The Run Time Support collector creates a file of support routines which must be included in the link edit step.

Support for "Full" RTS Library systems may be as small as 18K and as large as 30K. The size is dependent upon the features of RTS that are referenced in the user's system.

17.5.1.2 Kernel RTS Library. The Kernel RTS Library is like the full system in that it has the following features.

1) multiple processes and preemptive scheduling

2) semaphores and interrupts

3) memory management

4) dynamic process creation with full process semantics as specified in the Microprocessor Pascal System

5) process termination without reclaiming process resources

6) system crash upon process exception.

However, the kernel system lacks the following features.

1) Process resources are not reclaimed on process termination. No attempt is made to reclaim the structures, resources, etc. of processes which terminate. Instead, the terminated process waits on a semaphore which cannot be signaled.

2) Process exception handling and process abort are not allowed. Any exception which a process encounters is fatal and causes a system crash.

3) No files are implemented, although standard ENCODE and DECODE routines may be called.

The following routines are not available in the kernel version of the Run Time Support Library:

1) start$term

2) onexception

3) p$abort

4) all file routines.

The following routines are available but do nothing in the kernel version of the Run Time Support Library:

1) ct$enter

2) ct$exit.

By specifying to the Run Time Support collector that the Kernel RTS Library is desired, an appropriate file of support routines is created to be included in the link edit step.

Support for a Kernel RTS system may be as small as 9K bytes and as large as 19K bytes.

17.5.1.3 Standard Procedure MESSAGE. The standard procedure MESSAGE is not defined by Run-Time Support under a stand-alone environment; it must be defined by the user. This is done by declaring a procedure called MSG$ as follows.

```
TYPE STRING = PACKED ARRAY [1..80] OF CHAR;
PROCEDURE MSG$(VAR S: STRING; LENGTH: INTEGER);
{ BODY OF MSG$ GOES HERE}
```

STRING is the character string passed to MESSAGE. LENGTH is the number of characters in the string.

17.5.2 Link Editing

The DX link edit control file should be patterned after the following:

```
SYMT
TASK SYSTEM
INCLUDE USERIN.OBJECT              ! USER'S USERINIT MODULE
INCLUDE CONFIG.OBJECT              ! USER'S CONFIG MODULE
INCLUDE RTS.SEGMENT                ! RESULT OF RTS COLLECT
INCLUDE SEG1.SEGMENT               ! USER'S SEG1
INCLUDE SEG2.SEGMENT               ! USER'S SEG2
...
INCLUDE SEGn.SEGMENT               ! USER'S SEGn
END
```

The TASK card names the system (maximum of 8 characters). Each INCLUDE has the pathname of a file of either assembly language (as with USERIN and CONFIG) or Microprocessor Pascal System code (as with the SEGi.SEGMENT pathnames) and an INCLUDE exists for each module of the user's system.

Under DX, the link editor is invoked by typing XLE. The user is then presented with the following menu:

```
EXECUTE LINKAGE EDITOR
       CONTROL ACCESS NAME:
 LINKED OUTPUT ACCESS NAME:
       LISTING ACCESS NAME:
               PRINT WIDTH:
            EXECUTION MODE:
```

The pathname of the file (above) with the INCLUDEs should be given as the CONTROL ACCESS NAME. The output of the link editor (to be loaded or programmed into ROM) goes to the LINKED OUTPUT ACCESS NAME. Finally, a link editor listing is produced on the LISTING ACCESS NAME. EXECUTION MODE is either FOREGROUND or BACKGROUND.

The TXSLNK edit control file should be patterned after the following:

```
     SYMT
     TASK SYSTEM
     INCLUDE DSC2:USERIN/OBJ          ! USER'S USERINIT MODULE
     INCLUDE DSC2:CONFIG/OBJ          ! USER'S CONFIG MODULE
     INCLUDE DSC2:RTS/SEG             ! RESULT OF RTS COLLECT
     INCLUDE DSC2:SEG1/SEG            ! USER'S SEG1
     INCLUDE DSC2:SEG2/SEG            ! USER'S SEG2
     ...
     INCLUDE DSC2:SEGn/SEG            ! USER'S SEGn
     END
```

One INCLUDE exists for each file of the user system, with a pathname now being a TX file pathname.

Under TX, the link editor is invoked by typing an exclamation point (!) to create a 'PROGRAM:' prompt. Make sure that the floppy disk with the link editor is mounted. Type the pathname of the link editor program, for instance DSC:TXSLNK/SYS, or, since it is on DSC, just :TXSLNK/SYS. TX prompts for input; your reply is the pathname of the above link control file. Then, TX prompts for output. Here, enter the pathname of the file to direct the object to, followed by a comma (,), followed by the pathname of the link editor listing file. TX now requests options, if any. The only valid option is memory, specified as Mnnnnn, where nnnnn is the number of bytes to use. This option need not be specified if the default is sufficient.

17.5.2.1 ROM/RAM Specification. The user must ensure that his code is placed appropriately for ROM programming, and that his data areas (DSEGs and CSEGs) placed into RAM or ROM areas (as appropriate) by the link editor. The preferred method of partitioning user data areas is using the PROGRAM, DATA, and COMMON statements of the link editor. These statements allow alignment of PSEGs, DSEGs, and CSEGs at addresses specified by the user. Each command may be used more than once, but the first command needs to be placed before the first INCLUDE statement. The DATA and COMMON statements are ignored if used without a PROGRAM statement. For further information on these commands, consult the TI990 LINK EDITOR REFERENCE MANUAL. Another way to accomplish this is by adding BSS statements or modules to align modules on ROM/RAM boundaries, but this method requires two links to determine module lengths so that alignment is possible.

17.6 TARGET DEBUGGER

The Target Debugger consists of a library of AMPL command procedures which permit real-time debugging of Interpretive RTS systems at the target machine level. This section of the manual describes how to use the AMPL procedures effectively.

There are essentially two separate libraries of AMPL procedures available. One set of procedures is for use with AMPL Version 1 systems, the other set of procedures is for use with AMPL Version 2 systems.

17.6.1 Overview

The AMPL debugging procedures for Microprocessor Pascal-based Systems provide a capability which is similar to that of the Host Debugger, described in Section VI.

Like the host debugger, the AMPL target debugger is oriented towards source-level debugging. Breakpoints can be set at specific source language statements. Commands are also provided to display the status of all processes in the system and the status of a single process. Tracing can be performed at the statement level, the routine entry/exit level, and at the process scheduling level. It is also convenient to examine and modify data in stack frames, heap packets, or absolute memory. Simply stated, the procedures comprising the target debugger provide a flexible, high-level form of debugging Microprocessor Pascal systems in a real-time environment.

The command library for the target debugger assumes an AMPL configuration in which the emulator and trace module are connected.

The following documents contain additional information related to the AMPL Version 1 system: "Model 990 Computer AMPL Microprocessor Prototyping Laboratory System Operation Guide" (part #946244-9701), and "Model 990 Computer AMPL System Tutorial" (part #989621-9701).

The following documents contain additional information related to the AMPL Version 2 system: "Model 990 Computer AMPL Microprocessor Prototyping Laboratory Operation Guide" (part #946275-9701), "Model 990 Computer AMPL Microprocessor Prototyping Laboratory Tutorial" (part #946276-9701), and "Model 990 Computer AMPL Microprocessor Prototyping Laboratory Command Library User's Guide" (part #946277-9701).

17.6.2 An Example

A system to be monitored using the debugger must be compiled with the DEBUG option set. This is done by inserting a .$DEBUG option comment into the source code before it is compiled -- see Section 5.3 of this manual for the available compiler options. The compiler listing contains useful information for debugging as follows: 1) in the body section of each module (between the BEGIN-END pair), statement numbers are listed in the left margin, 2) in the declaration section of a module, stack frame displacements for variables are listed. Consider the example compiler listing provided in Figure 17-13.

```
      DX Microprocessor Pascal System Compiler     1.0    05/29/79 15:53:4

        0 {$ DEBUG, MAP}
        0 PROGRAM example;
        0 VAR
        0     n: INTEGER;
        2     m: INTEGER;
        0 FUNCTION factorial (i: INTEGER): INTEGER;
        1 BEGIN
        1   IF i <= 1 THEN
        2     factorial := 1
        3   ELSE
        3     factorial := i * factorial (i - 1)
        4 END;
        1 BEGIN  {# stacksize = 200}
        1   n := 5;
        2   m := factorial (n);
        3 END.

     PROGRAM EXAMPLE ;
         STACK SIZE = 0004

               VARIABLE  DISP      TYPE      SIZE
               N         0000      INTEGER   2
               M         0002      INTEGER   2

     FUNCTION FACTORIA ( I       :INTEGER):INTEGER;
```

FIGURE 17-13. EXAMPLE COMPILER LISTING

In the program "example", the variable "n" is at displacement 0 and "m" is at displacement 2. The value parameter "i" in the function "factorial" is stored at displacement 0 in its stack frame. The function result is stored at displacement 2 in the stack frame for "factorial". The statement numbers are listed in the body section.

In the following AMPL debugging session, note that user responses immediately follow any question mark (?) prompt.

```
  ?  COPY('DSC2:AMPL1/PRC')

   INITIALIZE EMULATOR AND TRACE MODULE
EINT(<EMULATOR NAME>)        'EMU' FOR TX AND 'EM01' FOR DX
TINT(<TRACE NAME>)           'TRA' FOR TX AND 'TM01' FOR DX
EUM = ON
LOAD(<PATHNAME OF LOAD MODULE>, 0, IDT)
COPY(<PATHNAME OF USER NAMES>)
RESET    ..  TO START TEST
HELP     ..  FOR MENU COMMANDS

?  EINT('EMU')

?  TINT('TRA')

?  EUM = ON

?  LOAD('DSC2:TEST/OBJ')      ..make sure user diskette is now in drive

?  COPY(DSC2:NAMES')

?  RESET
RESET INITIALIZATION COMPLETE

  ?  AB(FACTOR,1)

  ?  AB(EXAMPL,1)

  ?  LB
```

```
LIST OF ALL BREAKPOINTS
"FA   CT   OR   "   STATEMENT =        1
"EX   AM   PL   "   STATEMENT =        1

  ?  DB(EXAMPL,1)

  ?  LB
LIST OF ALL BREAKPOINTS
"FA   CT   OR   "   STATEMENT =        1

  ?  AB(EXAMPL,3)

  ?  AB(EXAMPL,2)

  ?  AB(EXAMPL,1)

  ?  LB
LIST OF ALL BREAKPOINTS
"FA   CT   OR   "   STATEMENT =        1
"EX   AM   PL   "   STATEMENT =        3
"EX   AM   PL   "   STATEMENT =        2
"EX   AM   PL   "   STATEMENT =        1

  ?  GO
*** BREAKPOINT *** "EX   AM   PL   "  STATEMENT        1     PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX   AM   PL   "  AT >3EC8
LEXICAL LEVEL      NAME            STATEMENT    FRAME
     1         "EX   AM   PL   "        1        >3DCE

  ?  SF(EXAMPL)
STACK FRAME FOR "EX   AM   PL   "
 3DCE (>0000) >FF00 >FF00                           (..   ..          )

  ?  SF(FACTOR)
STACK FRAME NOT FOUND

  ?  GO
*** BREAKPOINT *** "EX   AM   PL   "  STATEMENT        2     PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX   AM   PL   "  AT >3EC8
LEXICAL LEVEL      NAME            STATEMENT    FRAME
     1         "EX   AM   PL   "        2        >3DCE

  ?  SF
STACK FRAME FOR "EX   AM   PL   "
>3DCE (>0000) >0005 >FF00                           (..   ..          )
```

```
  ?  GO
*** BREAKPOINT *** "FA   CT   OR   "  STATEMENT      1     PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL   "  AT >3EC8
LEXICAL LEVEL      NAME           STATEMENT    FRAME
     2          "FA   CT   OR   "      1         >3DF0
     1          "EX   AM   PL   "      2         >3DCE

  ?  SF
STACK FRAME FOR "FA   CT   OR   "
 3DF0 ( 0000)   0005  >FF00                   (..   ..            )

  ?  SF(FACTOR)
STACK FRAME FOR "FA   CT   OR   "
 3DF0 ( 0000)   0005   FF00                   (..   ..            )

  ?  GO
*** BREAKPOINT *** "FA   CT   OR   "  STATEMENT      1     PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL   "  AT >3EC8
LEXICAL LEVEL      NAME           STATEMENT    FRAME
     2          "FA   CT   OR   "      1         >3E12
     2          "FA   CT   OR   "      3         >3DF0
     1          "EX   AM   PL   "      2         >3DCE

  ?  SF
STACK FRAME FOR "FA   CT   OR   "
>3E12 (>0000)  >0004  >0001                   (..   ..            )

  ?  GO
*** BREAKPOINT *** "FA   CT   OR   "  STATEMENT      1     PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL   "  AT >3EC8
LEXICAL LEVEL      NAME           STATEMENT    FRAME
     2          "FA   CT   OR   "      1         >3E34
     2          "FA   CT   OR   "      3         >3E12
     2          "FA   CT   OR   "      3         >3DF0
     1          "EX   AM   PL   "      2         >3DCE

  ?  SF
STACK FRAME FOR "FA   CT   OR   "
>3E34 (>0000)  >0003  >0001                   (..   ..            )

  ?  GO
*** BREAKPOINT *** "FA   CT   OR   "  STATEMENT      1     PROCESS =>3EC8
```

```
  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL  "  AT >3EC8
LEXICAL LEVEL      NAME              STATEMENT    FRAME
     2          "FA  CT  OR  "           1        >3E56
     2          "FA  CT  OR  "           3        >3E34
     2          "FA  CT  OR  "           3        >3E12
     2          "FA  CT  OR  "           3        >3DF0
     1          "EX  AM  PL  "           2        >3DCE

  ?  SF
STACK FRAME FOR "FA  CT  OR  "
>3E56 (>0000) >0002 >0001                         (..  ..         )

  ?  GO
*** BREAKPOINT *** "FA  CT  OR  "  STATEMENT      1    PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL  "  AT >3EC8
LEXICAL LEVEL      NAME              STATEMENT    FRAME
     2          "FA  CT  OR  "           1        >3E78
     2          "FA  CT  OR  "           3        >3E56
     2          "FA  CT  OR  "           3        >3E34
     2          "FA  CT  OR  "           3        >3E12
     2          "FA  CT  OR  "           3        >3DF0
     1          "EX  AM  PL  "           2        >3DCE

  ?  SF
STACK FRAME FOR "FA  CT  OR  "
>3E78 ( 0000) >0001 >0001                         (..  ..         )

  ?  GO
--- BREAKPOINT --- "EX  AM  PL  "  STATEMENT      3    PROCESS =>3EC8

  ?  DP
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "EX  AM  PL  "  AT >3EC8
LEXICAL LEVEL      NAME              STATEMENT    FRAME
     1          "EX  AM  PL  "           3        >3DCE

  ?  SF
STACK FRAME FOR "EX  AM  PL  "
 3DCE (>0000) >0005 >0078                         (..  .x         )

  ?  GO
EMULATOR IS IDLE
EMULATOR IS IDLE
EMULATOR IS IDLE

                       ..AT THIS POINT, "CMD" KEY WAS PRESSED
                       ..BY THE USER TO WAKE UP AMPL
  ?  HALT

  ?  STAT
EMULATOR IS HALTED

  ?  QUIT
```

17.6.3 How to Get Going

Once the AMPL system has been started, a question mark (?) is displayed to prompt the user to enter AMPL commands or statements. Here is a step-by-step sequence of commands that can be used to get started.

1. Copy the target debugger procedures using a COPY command. Here are the pathnames for the different systems which are available.

```
 COPY('$MPP.AMPL1')         .. DX AMPL Version 1 procs
 COPY('$MPP.AMPL2')         .. DX AMPL Version 2 procs
 COPY('DSC2:AMPL1/PRC')     .. TX AMPL Version 1 procs
 COPY('DSC2:AMPL2/PRC')     .. TX AMPL Version 2 procs
```

2. The following messages are displayed after the procs are copied.

```
 .. for DX and TX AMPL Version 1 procs

   INITIALIZE EMULATOP AND TRACE MODULE
 EINT('<EMULATOR NAME>)     'EMU' FOR TX AND 'EM01' FOR DX
 TINT(<TRACE NAME>)         'TRA' FOR TX AND 'TM01' FOR DX
 EUM = ON
 LOAD(<PATHNAME OF LOAD MODULE>), 0, IDT)
 COPY(<PATHNAME OF USER NAMES>)
 RESET   .. TO START TEST
 HELP    .. FOR MENU OF COMMANDS

 .. for DX and TX AMPL Version 2 procs

   INITIALIZE EMULATOR AND TRACE MODULE
 EINT('EM01',1,'TM01')
 LOAD(<PATHNAME OF LOAD MODULE>) 0, IDT)
 COPY(<PATHNAME OF USER NAMES>)
 RESET   .. TO START TEST
 HELP    .. FOR MENU OF COMMANDS
```

3. The AMPL emulator and trace modules should be initialized using the appropriate commands given above.

4. The user should then load the target load module and module

names with the load and copy commands given above. When using a TX system, the AMPL procs diskette should be replaced with the user diskette before the LOAD and COPY commands are performed.

5. Enter a RESET command to get the Target system initialized. The following message will then be displayed.

   RESET INITIALIZATION COMPLETE

6. The Target system is now ready to be debugged by using any of th commands available. The GO command is used to execute the syste until it reaches a breakpoint or some error condition occurs.

17.6.3.1 HELP Command. The HELP command displays a list of availablem commands with a short description of each one. The list is as follows:

```
RESET - "RESET" EXECUTION                 *SM(ADDR,<LEN>) - SHOW MEMORY
GO - RESUME EXECUTION                     *MM(ADDR,OLD,NEW) - MODIFY MEMORY
STAT - SHOW EMULATOR STATUS               *AB(NAME,STMT) - ASSIGN BREAKPOINT
HALT - HALT SYSTEM                        *DB(NAME,STMT) - DELETE BREAKPOINT
QUIT - QUIT DEBUGGING SESSION             *LB - LIST BREAKPOINTS
DAP - DISPLAY ALL PROCESSES               *DAB - DELETE ALL BREAKPOINTS
SDP(PRCS) - DISPLAY DEFAULT PROCESS       *SS(<FLAG>) - SINGLE-STEP MODE
DP(<PRCS>) - DISPLAY PROCESS              *TP(<FLAG>) - TRACE PROCESS
SF(NAME,DISP,LEN) - SHOW FRAME            *TR(<FLAG>) - TRACE ROUTINES
SH(<ADDR>) - SHOW HEAP PACKET(S)          *TS(<FLAG>) - TRACE STATEMENTS
SP(<PRCS>) - SHOW PROCESS                 *DT - DISPLAY TRACE DATA
```

17.6.3.2 RESET Command. The RESET command is used to start execution of the user's system at the "reset" trap vector. At the completion of this procedure, the message "RESET INITIALIZATION COMPLETE" is displayed. This command also allows the debugging session to be restarted. To restart the system, the emulator must be halted (this can be forced using a HALT command) before the RESET command is issued.

RESET does not cause execution of the user's system to begin. A GO command (see the next section) must be entered before execution starts. This allows breakpoints to be set before the system is placed into execution.

17.6.3.3 GO Command. This command is used to start execution or to resume execution of the user's system after it has become suspended for some reason, e.g. encountering a breakpoint. After entering a GO command, AMPL does not return control to the user until a breakpoint is encountered, an exception occurs, or the CMD key is pressed.

The GO command can only be entered when the emulator is halted. If a GO command is attempted and the emulator is running, the following message is displayed: "EMULATOR IS RUNNING".

If all processes terminate, the message "EMULATOR IS IDLE" is displayed periodically. The emulator will remain in the idle state until an interrupt occurs or until the system is restarted manually. To restart the system, perform the following steps:

1. Press the CMD key to wake up AMPL.

2. When the ? prompt appears, enter a HALT command to halt the emulator.

3. Enter a RESET command.

4. When the "RESET INITIALIZATION COMPLETE" message appears, the system is ready to go. Enter a GO command to start execution.

17.6.3.4 STAT Command. This command displays the current status of the emulator. One message of the follwing form is displayed:

EMULATOR IS RUNNING - this indicates that the emulator is running

EMULATOR IS RUNNING AT IDLE - this indicates that the emulator is running but "executing" an IDLE instruction, it will remain in the idle state until an interrupt occurs or until the system is restarted

EMULATOR IS HALTED - this indicates that the emulator has been halted

17.6.3.5 HALT Command. This command halts the emulator. A GO command can be issued to resume execution. Alternatively, the debugging session can be restarted using a RESET command.

17.6.3.6 QUIT Command. This command terminates the AMPL debugging session. It is equivalent to performing an AMPL "EXIT" command.

17.6.4 Status Displays/Selection of Default Process. Three commands are provided to display the status of processes being debugged. The DAP (Display All Processes) command is used to obtain the status of all processes in the system. The DP (Display Process) command shows the state of a single process, including its present dynamic calling sequence. The SP (Show Process) command gives detailed low-level information about a single process.

17.6.4.1 Display All Processes - DAP. This command lists the status of every process currently known to the system. Consider the following example:

```
STATUS SUMMARY OF ALL EXISTING PROCESSES

                              LEXICAL     CURRENT
  PROCESS NAME      ADDRESS    LEVEL      STATUS          PRIORITY
  "AS   R7   33   "  >8FE2      0         ACTIVE
  "CS   XI   N    "  >7440      1         WAITING
  "FO   MA   T    "  >8326      2         WAITING
  "FO   RM   AT   "  >7A30      1         READY              6
  "CS   XO   UT   "  >84F0      2         WAITING
  "CS   XO   UT   "  >8208      2         READY              6
  "KE   YI   N    "  >910A      1         WAITING
  "PR   IN   T    "  >89E4      1         READY              6
```

The first column contains the name of each process. If the name for a process (or routine) cannot be found, the name is displayed as a segment name followed by the displacement in that segment. For example, the string "SEGO .+>012A" represents the process (or routine) in the segment with IDT name SEGO, at displacement 012A.

The second column contains the address of the process record for each process. This address uniquely identifies the process in commands such as SDP (Select Default Process), DP (Display Process), and SP (Show Process).

The third column is the static lexical nesting level for the process.

The fourth column shows the current status of the process. There is a single ACTIVE process, and (possibly) several READY processes. All other processes are blocked for some reason as indicated by the status WAITING.

The final column shows the priority of all processes which are READY.

17.6.4.2 Select Default Process - SDP(process). This command is used to select a default process to be used in subsequent DP or SP commands. A single parameter to this command is the address of the process record as it appears in the DAP display.

If the process address does not point to a valid process record the message "INVALID PROCESS" is displayed. Otherwise, a message containing the name of the selected process is shown, for example,

```
                 SELECTED PROCESS = "EX AM PL "
```

17.6.4.3 Display Process - DP(<process>). This command displays a detailed status for a single process. An optional parameter to this routine is the address of the process record as it appears in the DAP display. If no parameter is given, the default process is assumed. If the process parameter does not point to a valid process, the message "INVALID PROCESS" is displayed. If the process has not been started yet, the message "PROCESS NOT STARTED" is displayed. The following display is shown for a valid process parameter.

```
STATIC/DYNAMIC CALLING ORDER FOR PROCESS "CS  XI  N    " AT >7440
LEXICAL LEVEL      NAME              STATEMENT     FRAME
      5        "GE  TC  HA  "           8          >2EDC
      4        "LI  NE  IN  "           7          >2EBA
      3        "CO  MM  AN  "          27          >2E98
      3        "SE  TU  P   "          12          >2E76
      2        "CS  XI  N   "           5          >2E54
      1        "AS  R7  33  "           -          >2E34
```

The top line of the display gives the name and address of the process being displayed. The names of all active routines (ones which have not returned) are listed in order from the most recently called to the oldest one called. The first column shows the static lexical nesting level for each routine. The second column is the name of the routine. The third column is the statement number for the last statement executed. The fourth column is the address of the stack frame for the routine.

17.6.4.4 Show Process - SP(<process>). This command is used to show important information about a particular process. The single parameter is the address of the desired process record. If no parameter is given, the default process is assumed. If the process parameter does not point to a valid process record, the message "INVALID PROCESS" is displayed. Otherwise, the following display is produced.

```
SHOW PROCESS "EX  AM  PL    " AT >3EC8
STACK BASE = >3D94    STACK LIMIT = >3EC8        STACK BOUNDARY = >3E7C
STACK SIZE = >0138    STACK USED (MAX) = >00EC   STACK USED (CUR) = >0058
HEAP SIZE = >0E9E     HEAP USED (MAX) = >0516    HEAP USED (CUR) = >0516
PRIORITY = 32766
NO OUTSTANDING EXCEPTIONS
NEXT PROCESS IN LIST = >3FA6    NEXT PROCESS IN QUEUE = >0000
QUEUE POINTER = >3F1E
CREATORS ID = >00    MY ID = >01
```

The first line displays the name and address of the process. The remaining items are discussed below.

STACK BASE is the base address for the stack belonging to the process.

STACK LIMIT is the maximum address the stack can ever grow to.

STACK BOUNDARY is the maximum address the stack has achieved so far.

STACK SIZE is the total amount of stack available for the process.

STACK USED (MAX) is the maximum amount of stack used thus far.

STACK USED (CUR) is the current amount of stack used by the process.

HEAP SIZE is the total heap size available for use by the process.

HEAP USED (MAX) is the maximum amount of heap used thus far.

HEAP USED (CUR) is the current amount of heap used by the process.

PRIORITY is the priority of the process.

NO OUTSTANDING EXCEPTIONS indicates that no outstanding exceptions exist. If an exception occurs, an appropriate error message is displayed here. See Section 17.6.8 for an explanation of all possible error messages.

NEXT PROCESS IN LIST is the address of the next process in the list of all processes.

NEXT PROCESS IN QUEUE is the address of the next process in the same queue as this process (if any). If the process is not in a queue or is the only process in the queue, this field is equal to zero.

QUEUE POINTER is a pointer to the queue on which this process resides. The active process is not on a queue and this field is equal to zero.

CREATORS ID is the identification field of the creator process.

MY ID is the identification field of this process.

17.6.5 Show/Modify Memory Commands. The commands to show data areas produce a formatted display. The following example is the display from a SF (show frame) command.

```
>A4DA  (>0000) >FFFF >0000 >0001 >5341    (..  ..  ..  SA)
>A4E2  (>0008) >4D50 >4C45 >2020 >0000    (MP  LE      ..)
>A4EA  (>0010) >0002 >0004 >0006 >0008    (..  ..  ..  ..)
>A4F2  (>0018) >000A >000C >000E >0010    (..  ..  ..  ..)
>A4FA  (>0020) >0012 >0014 >0016          (..  ..  ..    )
```

The first word of every display line is the absolute memory address of the first data word displayed on the line. In the example, the data displayed on the first line starts at memory address A4DA. Immediately following the address is the displacement into the stack frame, or heap packet. The displacement is enclosed in parentheses. Each line contains up to four words of data. At the end of each line, the data is displayed in ASCII format; those bytes which represent non-printable ASCII characters are displayed as a period.

17.6.5.1 Show Frame - SF(<name>,<displacement>,<length>). This command is used to display a stack frame (or portion thereof). The command has three parameters which are all optional. The first parameter specifies the name of the routine. If no parameters are given, the most recent frame for the default process is displayed. A succession of SF commands without parameters shows all stack frames starting with the youngest one to the oldest.

The second parameter is the byte displacement into the stack frame at which to start the display. The default displacement is zero.

The third parameter is the number of bytes to be displayed. By default, the entire stack frame is displayed.

The following error messages can result from a SF command.

INVALID NAME - the routine name does not represent a valid routine

STACK FRAME NOT FOUND - the given routine has not been invoked or is not invoked through the default process

NO MORE STACK FRAMES - all stack frames for the process have been shown

BAD DISPLACEMENT - the displacement value is out of bounds for the frame

BAD LENGTH - the length parameter is out of bounds for the frame

17.6.5.2 Show Heap - SH(address,<displacement>,<length>). This command is used to show the contents of a heap packet. A heap packet is identified by an absolute memory address which is specified by the first parameter.

The second parameter is the byte displacement into the heap packet at which to start the display. The default displacement is zero.

The third parameter is the number of bytes to display. If omitted, the entire heap packet is displayed.

The following error messages can result from a SH command.

INVALID HEAP PACKET - the specified address does not point to a heap packet

BAD DISPLACEMENT - the displacement parameter extends beyond the heap packet

BAD LENGTH - the length parameter extends beyond the heap packet

17.6.5.3 Show Memory - SM(address, <length>). This command is used to show the contents of a portion of absolute memory. The first parameter is the address of the memory area to be displayed. The second parameter is the length, in bytes, to be displayed. If no length is given, one word is displayed.

17.6.5.4 Modify Memory - MM(address, old value, new value).This command is used to modify the contents of any single (word) location in memory. The first parameter is the address of the word to be modified. The second parameter is the old value for the word. If the old value parameter given does not match the true old value, the word is not modified and the message "INCORRECT OLD VALUE" is given. The third parameter is the new value for the word. If the modification is performed, the message

```
@>XXXX = >YYYY
```

is displayed, where >XXXX is the address of the word which was modified and >YYYY is the new value of the word.

17.6.6 Breakpoints/Single-Step Commands

Breakpoints can be set in any routine which was compiled with the .$DEBUG compile option. When a breakpoint is encountered, execution is suspended to enable examination/modification of the state of the system. A message of the following form is displayed:

```
 *** BREAKPOINT ***  "name"    STATEMENT nn         PROCESS =>mmmm
```

where "name" is the name of the routine, "nn" is the statement number, and ">mmmm" is the address of the process containing the routine.

A maximum of four breakpoints may be set at one time.

17.6.6.1 Assign Breakpoint - AB(name,<statement>). This command is used to assign a breakpoint to a routine. The first parameter is the name of the routine in which to set the breakpoint. The second parameter is the statement number at which to breakpoint. The default statement number is one.

The following error messages can result from an AB command.

INVALID NAME - the specified routine name is not valid

TOO MANY BREAKPOINTS - only four breakpoints can be set at once

17.6.6.2 Delete Breakpoint - DB(name,<statement>). This command is used to delete a breakpoint from a routine. The first parameter is the name of the routine in which to delete the breakpoint. The second parameter is the statement number for the breakpoint to be deleted. The default statement number is one.

The following error messages can result from a DB command.

NO BREAKPOINTS ASSIGNED - there are currently no breakpoints set

INVALID NAME - the specified routine name is not valid

BREAKPOINT NOT FOUND - the specified breakpoint has not been set

17.6.6.3 List Breakpoints - LB. This command is used to list all breakpoints which are currently set. The following list is an example.

```
LIST OF ALL BREAKPOINTS
"EX  AM  PL   "  STATEMENT =      1
```

```
"EX   AM   PL    "  STATEMENT =      2
"FA   CT   OR    "  STATEMENT =      1
"FA   CT   OR    "  STATEMENT =      5
```

The list indicates that four breakpoints are set. Breakpoints in the routine EXAMPL are set at statements 1 and 2. Breakpoints in the routine FACTOR are set at statements 1 and 5.

If no breakpoints are assigned, the message "NO BREAKPOINTS ARE ASSIGNED" is displayed.

17.6.6.4 Delete All Breakpoints - DAB. This command is used to delete all breakpoints.

17.6.6.5 Single-Step - SS(<flag>). This command is used to perform single-step execution. The optional parameter is a flag which indicates whether to enable or disable single-step mode. The value TRUE enables single-step mode; the value FALSE disables single-step mode. The default value is TRUE.

While in single-step mode, statements are executed one at a time. A breakpoint is forced between every statement. A message of the following form is displayed:

```
   --- BREAKPOINT ---   "name"    STATMENT  nn     PROCESS = <mmmm
```

where "name" is the name of the routine, "nn" is the statement number, and ">nnnn" is the address of the process containing the routine.

17.6.7 Tracing Commands

There are three kinds of tracing available. Tracing is useful to examine the behavior of the scheduling algorithm, to observe the dynamic behavior of routine calls and exits, and to determine the actual control flow of statements.

Tracing is performed in real-time using the Target Debugger. AMPL maintains a trace buffer which contains at most 256 words. The Target Debugger requires two words per trace sample which limits the number of trace samples that can be examined at any point to 128. When the user's system becomes suspended, the contents of the trace buffer can be displayed using the DT command (note that the emulator must not be running, i.e. a HALT command should be issued first). The trace buffer always contains the most recent trace samples.

17.6.7.1 Trace Process Scheduling - TP(<flag>). This command is used to enable or disable the tracing of process scheduling. The flag parameter must be either TRUE or FALSE. The default value is TRUE.

17.6.7.2 Trace Routine Entry/Exit - TR(<flag>). This command is used to enable or disable routine entry and routine exit tracing. The flag parameter must be either TRUE or FALSE. The default value is TRUE.

17.6.7.3 Trace Statement Flow - TS(<flag>). This command is used to enable or disable statement tracing. The flag parameter must be either TRUE or FALSE. The default value is TRUE.

17.6.7.4 Display Trace - DT(<count>). This command is used to display the current contents of the trace buffer. The optional ¼count½ parameter specifies the number of trace samples to be displayed. If the count is not specified, the contents of the entire trace buffer is displayed. If the count buffer is specified and the count is larger than the number of trace samples in the trace buffer, the contents of the entire trace buffer is displayed.

Trace data has the following four forms.

```
--- TRACE ---     "name"     PROCESS ACTIVE >nnnn
--- TRACE ---     "name"     ROUTINE ENTRY
--- TRACE ---     "name"     ROUTINE EXIT
--- TRACE ---     "name"     STATEMENT mm
```

The first form is for process tracing. The name and address of the process is given. The second and third form are for routine entry/exit tracing. The fourth form is for statement level tracing where "mm" is the statement number. Warning: the names listed for process tracing may be inaccurate for processes which have terminated.

17.6.8 Error Messages

If a run-time execution error is detected by the interpreter, one of the following messages is displayed, followed by the statement number and routine in which the error was detected:

INVALID OPCODE - an invalid opcode was encountered by the interpreter

STACK OVERFLOW - the allocated stack memory region has been exhausted

UNRESOLVED PROCEDURE CALL - an attempt to call an unresolved procedure was made, the system should be reconfigured using the Microprocessor Pascal COLLECT command

DIVISION BY ZERO - a divide by zero was attempted

FLOATING POINT ERROR - a real value is too large or too small to be represented

SET RANGE ERROR - a set member has an ordinal value less than 0 or greater than 1023

ASSERT ERROR - the expression in an ASSERT statement evaluated to false

CASE ERROR - no OTHERWISE clause was found in a CASE statement and the case selector did not evaluate to any case label present value of a label

ARRAY INDEX ERROR - an array index is out of bounds

POINTER ERROR - an attempt to reference through a NIL pointer

SUBRANGE ERROR - a value to be assigned is out of bounds of the subrange

UNKNOWN ERROR = nn - unknown error, should be reported at once

If a run-time execution error is detected by run-time support code, one of the following messages is displayed:

USER ERROR: REASON = nn - see Section 13.2.1 of the manual

SCHEDULING ERROR: REASON = nn - see Section 13.2.2 of the manual

SEMAPHORE ERROR: REASON = nn - see Section 13.2.3 of the manual

INTERRUPT ERROR: REASON = nn - see Section 13.2.4 of the manual

PROCESS MGMT ERROR: REASON = nn - see Section 13.2.5 of the manual

EXCEPTION ERROR: REASON = nn - see Section 13.2.6 of the manual

MEMORY MGMT ERROR: REASON = nn - see Section 13.2.7 of the manual

FILE ERROR: REASON = nn - see Section 13.2.8 of the manual

17.7 EXAMPLE

Tne program in Section III shall serve as an example of how to make a load module for a stand-alone system. Each of the steps required in this process will be shown and explained. For our demonstration, we shall use BEGIN END; as the body of MSG$, primarily to eliminate an unresolved reference. MSG$ is discussed in Section 17.5.1.3.

This example will be run under DX, but the user interface is similar under TX, and any differences are discussed in the section on TX.

If the source is on file .SOURCE, then a typical sequence to create a load module would look like this. Pathnames have been chosen arbitrarily, and are intended only to represent possible pathnames. (Any comments about what is being done are in braces .)

17.7.1 Compile and Save

```
{After adding MSG$ to the source, a compile and save are done.}
[] COMPILE
{Giving the menu and replies}
 COMPILE A Microprocessor Pascal System
           SOURCE: .SOURCE
          LISTING: .LIST
           MEMORY: 4,6
       FOREGROUND: YES
{Messages from the compiler are displayed on the screen}
COMPILER  EXECUTION BEGINS
SCANNER IS FINISHED
PRODUCER
CONSUMER
MSG$
TUTORIAL
NO ERRORS IN COMPILATION
NORMAL TERMINATION
STACK USED =   2440  HEAP USED =  4008

[] SAVE
{Giving the menu and replies}
 SAVE A Microprosser Pascal System SEGMENT
           LISTING: .MAP
      SEGMENT FILE: .SEGMENT
```

{Messages from the save program are displayed on the screen. Since this is the segment in which execution is to begin, the segment number is one (1). If other segments were to be in the system, several other compiles and saves would be done here as well.}

```
SAVE EXECUTION BEGINS
ENTER THE SEGMENT NUMBER:
1
```

```
INSERT DEBUG INFO? (YES/NO):
YES { 'NO' would be used for the final product}
NORMAL TERMINATION
STACK USED =    3982  HEAP USED =  1388
```

17.7.2 Collect

```
[] COLLECT
{Giving the menu and replies}
 COLLECT RUN TIME SUPPORT SEGMENTS
          COMMANDS: ME
   AMPL INPUT FILE: .AMPL
RTS OBJECT MODULES: .RTS
        KERNEL RTS: no
```

{Messages from the collect program are displayed on the screen. Since the system has only one user segment, only the pathname .segment is entered into the collector, followed by end of file. Had other segments been in the system, the collector would need to know the pathnames of the other segments.}

```
COLLECT   EXECUTION BEGINS
ENTER PATHNAME OF USER SEGMENTS
.SEGMENT
ENTER NEXT USER SEGMENT PATHNAME
                                {User hits 'RETURN' key}
READING RTS XREF FILES
BUILDING RTS SEGMENTS
COPYING INTERPRETER MODULES
NORMAL TERMINATION
STACK USED =    3954  HEAP USED = 12288
```

17.7.3 Modify the 'CONFIG' Module

```
[] EDIT
```

{Editing a copy of CONFIG to include>ROM/RAM partitioning: The hardware has RAM for system use from >8000 to >9FFF and from C000 to CFFF. The tables in CONFIG would look similar to the following.}

```
STATIC EQU >8000
       DORG STATIC
IWP$   BSS >20
BAD$WP BSS >20
INT$WP EQU  $                    THIS WORKSPACE MUST BE LAST!
----------------------------------------------------------
       PSEG
$RAMTB DATA >A000-INT$WP,INT$WP (>A000-INT$WP) BYTES
                                          BEGINNING AT INT$WP
       DATA >1000,>C000         >1000 BYTES BEGINNING AT >C000
       DATA 0
$RESTA EQU  0                   RESTART IS LEVEL 0 INTERRUPT
$LREX  EQU  0                   NO LREX BLWP VECTOR
       END
```

```
[] SCI
{Giving the menu and replies
                   COMMAND: XMA
{Giving the menu and replies
 EXECUTE MACRO ASSEMBLER
        SOURCE ACCESS NAME: .CONFIG .pathname of CONFIG source
        OBJECT ACCESS NAME: .CONFOBJ
       LISTING ACCESS NAME: .CONFLST
         ERROR ACCESS NAME:
                   OPTIONS:
    MACRO LIBRARY PATHNAME:
            EXECUTION MODE: FOREGROUND
{The assembler finishes and the listing (.CONFLST) shows no errors}
```

17.7.4 Create a Link Edit File

```
[]EDIT
{Create a link edit control file (.LINK) including
the appropriate modules.}

     SYMT
     TASK EXAMPLE
     INCLUDE .USERIN {pathname of standard userinit object}
     INCLUDE .CONFOBJ
     INCLUDE .RTS
     INCLUDE .SEGMENT
     END
```

17.7.5 Link Edit

```
[] SCI
{Giving the menu and replies}
                      COMMAND: XLE
{Giving the menu and replies}
EXECUTE LINKAGE EDITOR
       CONTROL ACCESS NAME: .LINK
 LINKED OUTPUT ACCESS NAME: .LOAD
       LISTING ACCESS NAME: .LINKMAP
               PRINT WIDTH: 80
            EXECUTION MODE: FOREGROUND
{The link edit shows no errors, and .LOAD is the pathname
of a load module of a stand-alone system.}
```

17.7.6 Target Debugger

To use AMPL Version 1, the following steps are taken.

```
[] SCI
{Giving the menu and replies}
                      COMMAND: AMPL
{Giving the menu and replies}
AMPL MICROPROCESSOR PROTOTYPING LAB
      USER MEMORY (K): 16
{To get the session started, the following commands are entered.}

? COPY('$MPP.AMPL1')

 INITIALIZE EMULATOR AND TRACE MODULE
EINT(<EMULATOR NAME>)      'EMU' FOR TX AND 'EM01' FOR DX
TINT(<TRACE NAME>)         'TRA' FOR TX AND 'TM01' FOR DX
EUM = ON
LOAD(<PATHNAME OF LOAD MODULE>, 0, IDT)
COPY(<PATHNAME OF USER NAMES>)
RESET    .. TO START TEST
HELP     .. FOR MENU OF COMMANDS

?  EINT('EM01')        .. to initialize the emulator module

?  TINT('TM01')        .. to initialize the trace module

?  EUM = ON            .. to specify that AMPL memory is to be used

?  LOAD('.LOAD')       .. to load target load module

?  COPY('.AMPL')       .. to copy target module names

? RESET                .. to initialize target system
RESET INITIALIZATION COMPLETE
{Now AMPL Target Debugger commands can be entered}
```

To use the AMPL Version 2, the following steps are taken.

```
[]SCI
{Giving the menu replies}
                 COMMAND:  AMPLDX
{Giving the menu replies}
DX10 - AMPL MICROPROCESSOR PROTOTYPING LAB
           USER MEMORY (K): 16
                 COPY FILE:
{To get the session started, the following commands are entered.}

?  COPY('$MPP.AMPL2')

{The following messages will then appear after the procs have been
 copied}

  INITIALIZE EMULATOR AND TRACE MODULE
EINT('EM01',1,'TM01')
LOAD(<PATHNAME OF LOAD MODULE>, 0, IDT)
COPY(<PATHNAME OF USER NAMES>)
RESET   .. TO START TEST
HELP    .. FOR MENU OF COMMANDS

?  EINT('EM01',1,'TM01')    ..  to initialize emulator and trace

?  LOAD('.LOAD')            ..  to load target load module

?  COPY('.AMPL')            ..  to copy target module names

?  RESET                    ..  to initialize target system
RESET INITIALIZATION COMPLETE
{Now AMPL Target Debugger commands can be entered}
```

APPENDIX A

GLOSSARY

active process: The single process which is currently executing (assigned to the processor).

address space, logical: A hypothetical contiguous memory area which is addressable by software, generally limited in size by the instruction set of the computer; for example, the 990/10 has a logical address space of 65,536 bytes, thus allowing a memory reference to consume a maximum of 16 bits.

address space, physical: The actual physical memory (hardware) which is available to a computer system; on the 990/10, a 16-bit logical address is mapped to some physical memory location through a hardware function referred to as memory mapping.

blocked process: A process which is not currently eligible for execution because it is waiting for some resource or some signal before it can continue.

breakpoint: A point in a system at which execution can be suspended, especially for debugging purposes.

buffer, line: A data area associated with each text file which contains the component (line) being encoded or decoded.

buffer, look-ahead: A data area associated with each file opened for reading which contains the component which will be read next. For text files, the line buffer doubles as a look-ahead buffer.

call by reference: A kind of parameter passing in which the address of the actual parameter is passed to the called module and this address is used to access the actual parameter indirectly, sometimes called variable substitution.

call by value: A kind of parameter passing in which the actual parameter is evaluated and the resulting value is assigned to the corresponding formal parameter, sometimes called value substitution.

channel: A shared data structure through which file variables are linked to devices and to other file variables.

channel, device: A dedicated channel associated with a device which connects a file variable to that device.

concurrency: The property of several distinct processes whose execution proceeds at the same time.

concurrent characteristic: One of several characteristics associated with the definition of a process, including the priority of the process, the amount of stack data space required, and the amount of heap data space required for it to execute.

critical transaction: A sensitive sequence of code which must be allowe to execute from top to bottom, with no possibility of another process being scheduled during the sequence.

CRU (communication register unit): The general-purpose, command-driven hardware interface of the TI 990/9900 family, used to communicate with many supported devices.

deadlock: The situation when two (or more) processes become blocked waiting for conditions that can never hold because of a circular dependency; each process is waiting on a condition that cannot occur because some other process is not active to cause it.

device channel: A dedicated channel associated with a device, which connects a file variable to that device.

device, logical: A device with which programs can perform logical (device-independent) I/O.

device, physical: A device which communicates with programs through CRU or memory-mapped I/O and interrupts.

end of consumption: The state of a channel when all connected reading files become closed.

end of file: The state of a channel and all connected reading files when all connected writing files become closed.

event: Something noteworthy that takes place either externally (in the real world) or internally (inside the Executive RTS).

exception: An error detected during the execution of a system, e.g. divide by zero or subscript out of range.

exception handling: The ability of a process to deal with exceptions and to possibly recover from them.

extent: The time during system execution that a computational quantity may be considered to exist; the extent of a variable is the time during which space is allocated for the variable.

file variable: A process-local port which interfaces the process with its external environment.

first-in first-out (FIFO) queue: A queue in which the components which arrive first are the first ones to leave.

heap: A data area which holds dynamically allocated variables which are not declared, but are created and destroyed by the procedures NEW and DISPOSE.

heap, program: A heap region that is allocated from system memory.

heap, nested: A heap region that is allocated from another heap, called the parent, so that a hierarchy of heaps may be created.

heap packet: An arbitrary size data area allocated from a heap.

heapsize: A concurrent characteristic which specifies the size of the heap required by a process; a zero value means to use the parent's heap, a non-zero value means to allocate a private heap from the parent's heap (nested heap).

idle process: A process with the lowest possible priority which becomes active when no other processes are ready to execute.

interleaving: Repetitive switching of processes, used to create the illusion of many processes executing concurrently.

interpretive code: Code produced by the MicroTIP compiler which can be executed by an interpreter or can be translated into 9900 native code by the code generator.

interrupt: A stimulus from the external environment of a processor to pass an event to a process executing within the processor.

interrupt demultiplexer: A process which waits for an interrupt from a physical device, determines the logical device for which the interrupt is intended, and signals a semaphore corresponding to the logical device.

I/O, logical: Device-independent I/O.

I/O, memory-mapped: I/O which is performed by reading and writing to "memory locations" which are dedicated to a device.

I/O, physical: Device-dependent I/O.

mask, interrupt: The hardware mask (specifically bits 12 through 15 of the ST register) which determines the enabled interrupt levels.

memory-mapped I/O: I/O which is performed by reading and writing to "memory locations" which are dedicated to a device.

memory mapping: A hardware function whereby a logical address is mapped to a physical memory location.

message: Any data which can be copied from one process to another, examples are a string of characters, an integer, an array, a record, or a pointer.

message buffer: A shared data structure through which messages are transferred and buffered among processes.

module: Any unit of Pascal which may be invoked, that is, either a system, program, process, procedure, or function.

multiprocessing: The concurrent execution of several communicating processes, possibly on different processors.

multiprogramming: The practice of having several sites of execution within one program at the same time (concurrent processes).

multitasking: A term used interchangeably with multiprogramming.

native code: Code which can be executed by a specific computer, e.g. TI 9900 code.

preempted process: A process which, because of the scheduling policy, must relinquish the processor to another process.

priority: A property of a process which indicates the relative urgency of the process; a lower priority number means the process is more urgent than one of a higher number.

process: A separately executing entity which has its own run-time environment for its data.

process record: A data area maintained by run-time support code for every instance of a process, which contains all necessary information about the process and its state

processor: A single CPU hardware device.

program: A process that is self-contained with respect to accessing data via scope of variables or pointers; it corresponds to the program construct of the standard Pascal language.

ready process: A process which is ready to execute, i.e. it is not currently blocked for any reason.

recursion: A property whereby an algorithm is expressed in terms of itself; this occurs whenever a routine calls itself either directly or indirectly (through a series of calls).

reentrancy: A property of code which allows multiple copies of a code module to be executing at the same time; the code must not be self-modifying and data references must be relative to the stack region.

scheduling policy: A discipline enforced by the Executive RTS which determines the assignment of a processor to one of several processes.

scheduling queue: A queue containing all processes which are ready to execute, in an order based upon priority.

scope: The range over which the declaration of a construct is effective

semaphore: A low-level structure associated with an event on which processes wish to synchronize.

SIGNAL operation: An operation performed on a semaphore by a process which signals the occurrence of a particular event.

spurious interrupt: An unexpected interrupt.

stack: The data area allocated to a process from which individual stack frames are allocated.

stacksize: A concurrent characteristic which specifies the number of words of storage which the process intends to use for its local variables and the variables associated with all susequent dynamic routine calls; this space is allocated from the heap of the lexical parent.

stack frame: A contiguous data area allocated for every activation of a routine, used to hold parameter values, local variables, temporary variables, and return linkage information.

suspended process: A process which is blocked, waiting for some change in the state of the system.

system: A process which comprises the outermost level of declarations and executable statements in which execution begins.

urgency: The degree to which a process requires attention, expressed in terms of its priority; a lower priority number indicates a greater urgency.

WAIT operation: An operation performed on a semaphore by a process to wait for the occurrence of a particular event before proceeding.

APPENDIX B

Microprocessor Pascal REFERENCE CARD

DX10 COMMAND SUMMARY

MPP	- ENTER SESSION
EDIT	- EDIT MODULE
COMPILE	- COMPILE SYSTEM
DEBUG	- DEBUG SYSTEM
EXECUTE	- EXECUTE PROGRAM
SAVE	- SAVE SEGMENT
BATCH	- COMPILE AND SAVE SEGMENT(BACKGROUND)
COLLECT	- COLLECT SUPPORT MODULES
SHOW	- SHOW FILE
PRINT	- PRINT FILE
SCI	- EXECUTE "SCI" COMMAND
WAIT	- WAIT ON BACKGROUND
PURGE	- PURGE SYNONYMS
QUIT	- QUIT SESSION

TX990 COMMAND SUMMARY

EDIT	- EDIT FILE
COMPILE	- COMPILE SYSTEM
DEBUG	- DEBUG SYSTEM
EXECUTE	- EXECUTE PROGRAM
SAVE	- SAVE SEGMENT
COLLECT	- COLLECT SUPPORT MODULES
SHOW	- SHOW FILE
COPY	- COPY "TEXT" FILE
UTILITY	- PERFORM DISC "UTILITY" FUNCTION

UTILITY FUNCTIONS

CF	- CREATE FILE
CM	- COMPRESS FILE
CN	- CHANGE FILE NAME
CP	- CHANGE FILE PROTECTION
DF	- DELETE FILE
DO	- CHANGE LISTING FILE OR DEVICE
MD	- MAP DISC
TI	- DISPLAY DATE AND TIME
TE	- TERMINATE UTILITY PROGRAM

SOURCE EDITOR COMMAND SUMMARY

Command/Function	911 VDT Key
Setup and Termination	
Help	CMD/"HELP"
Edit/Compose Toggle	F7
Syntax Check	CMD/"CHECK"
Quit	CMD/"QUIT"
Abort	CMD/"ABORT"
Save	CMD/"SAVE"
Input	CMD/"INPUT"
Cursor Positioning	
Roll Up	F1
Roll Down	F2
New Line	RETURN
Tab	SHIFT TAB SKIP
Back Tab	FIELD
Set Tab Increment	CMD/"TAB(increment)"
Cursor Up	up-arrow
Cursor Down	down-arrow
Cursor Right	right-arrow
Cursor Left	left-arrow
Home	HOME
Find	CMD/"FIND(parameters)"
Relative Positioning	CMD/number
Top	CMD/"TOP"
Bottom	CMD/"BOTTOM"
Program Modification	
Insert Line	unlabeled gray key
Duplicate Line	F4
Delete Line	ERASE INPUT
Skip	TAB SKIP
Insert Character	INS CHAR
Delete Character	DEL CHAR
Clear Line	ERASE FIELD
Replace	CMD/"REPLACE(parameters)"
Split Line	F8
Insert	CMD/"INSERT"
Block Commands	
Start Block	F5
End Block	F6
Copy	CMD/"COPY"
Move	CMD/"MOVE"
Delete	CMD/"DELETE"
Put	CMD/"PUT"
Show Command	
Show	CMD/"SHOW"

HOST DEBUGGER COMMAND SUMMARY

Command Name	Meaning
Getting Started/Finished	
GO	Resume execution
QUIT	Quit debugging session
HELP(command name)	Help command
LOAD("pathname")	Load saved segment
SE	Show unresolved Externals
COPY("pathname")	Copy commands from file
Status Displays	
DP(process)	Display Process
DAP	Display All Processes
Breakpoints/Single Step	
AB(routine, statement number)	Assign Breakpoint
DB(routine, statement number)	Delete Breakpoint
DAB(process)	Delete All Breakpoints
LB(process)	List Breakpoints
SS(process , flag)	Single-Step execution mode
Showing/Modifying Data	
SF(routine , displacement , length)	Show Frame
SH(address , displacement , length)	Show Heap
SC(common name, displacement , length)	Show Common
SI(routine,displacement, length)	Show Indirect
SM(address, length)	Show Memory
MF(routine, displacement , verify value ,new value)	Modify Frame
MH(address, displacement , verify value ,new value)	Modify Heap
MC(common name, displacement , verify value ,new value)	Modify Common
MI(routine,displacement, verify value ,new value)	Modify Indirec
MM(address, verify value ,new value)	Modify Memory
Tracing Execution	
TP(process , flag)	Trace Process scheduling
TR(process , flag)	Trace Routine entry/exit
TS(process , flag)	Trace Statement flow
TOFF	Trace echo OFF
TON	Trace echo ON
Monitor Process Scheduling	
SDP(process)	Select Default Process
DEBUG(process name, flag)	Debug process
ABP(process)	Assign Breakpoint to Process
DBP(process)	Delete Breakpoint from Process
HP(process)	Hold Process
RP(process)	Release Process
Interprocess File Simulation	
CIF("internal file","external file")	Connect Input File
COF("internal file","external file")	Connect Output File
Interrupt Simulation	
SIMI(level)	SIMulate Interrupt
Selection of CRU Mode	
CRU(process ,cru mode)	select CRU mode

TARGET DEBUGGER COMMAND SUMMARY

Command Name	Meaning
Getting Started/Finished	
INIT	Initialization Command
HELP	Help Command
GO	Resume Execution
STAT	Current Status of the Emulator
HALT	Halt the Emulator
QUIT	Terminate AMPL Debugger Session
Status Displays/Selection of Default Process	
DAP	Display All Processes
SDP(process)	Select Default Process
DP(process)	Display Process
Show/Modify Memory	
SF(name , displacement , length)	Show Frame
SH(address , displacement , length)	Show Heap
SP(process)	Show Process
SM(address, length)	Show Memory
MM(address,old value,new value)	Modify Memory
Breakpoints/Single-Step	
AB(name, statement)	Assign Breakpoint
DB(name, statement)	Delete Breakpoint
LB	List Breakpoints
DAB	Delete All Breakpoints
SS(flag)	Single-Step
Tracing Execution	
TP(flag)	Trace Process Scheduling
TR(flag)	Trace Routine Entry/Exit
TS(flag)	Trace Statement Flow
DT(count)	Display Trace

APPENDIX C

Microprocessor Pascal STANDARD ROUTINES

The standard procedures and functions supported in Microprocessor Pascal are described in this section. In addition to the predeclared standard routines, there are several routines which the user may call by first declaring them to be EXTERNAL. The standard routines are categorized according to the function they serve.

C.1 DATA CONVERSION ROUTINES

FUNCTION FLOAT(X) - X may be of type INTEGER or LONGINT, the result is the converted REAL value

FUNCTION LINT(X) - X may be of type INTEGER, LONGINT, or REAL, the result is the converted LONGINT value

FUNCTION TRUNC(X) - X is of type LONGINT or REAL, the result is the truncated INTEGER value

FUNCTION LTRUNC(X) - same as TRUNC, except result is of type LONGINT

FUNCTION ROUND(X) - X is of type REAL, the result is the rounded INTEGER value defined as:

```
= TRUNC(X + 0.5),   X>=0
= TRUNC(X - 0.5),   X<0
```

FUNCTION LROUND(X) - same as ROUND, except result is of type LONGINT

PROCEDURE DECODE(S, N, STAT, Q) - S is a variable of type string. N is an INTEGER starting index into S. STAT is a returned status code, and Q is a "read parameter". This procedure converts the ASCII string starting at the Nth component of S to its internal form and places the value in the variable Q. Q may be a list of read parameters.

PROCEDURE ENCODE(S, N, STAT, P) - This procedure converts the value of the "write parameter" P into an ASCII string which is placed into the string starting at the Nth component of S. The STAT parameter is the returned status code from the operation. P may be a list of write parameters.

C.2 FILE MANIPULATION ROUTINES

FUNCTION EOF(F) : BOOLEAN - F is a file variable, the result, of type BOOLEAN is true if the file F is not open for input or is in the end-of-file state.

FUNCTION EOLN(F) : BOOLEAN - F is a text file variable, the result, of type BOOLEAN is true if the last character of the current line in the file F has been read.

FUNCTION FILENAMED(S) : ANYFILE - S is a string constant which specifies the file name, the result, of type ANYFILE is the file variable which is connected to the file with name S.

PROCEDURE RESET(F) - This procedure opens the file F for input and positions it to read its first component. If the file is empty, EOF(F) becomes TRUE, otherwise it becomes FALSE.

PROCEDURE REWRITE(F) - This procedure makes the file F empty and opens it for output. EOF(F) becomes TRUE. A REWRITE operation is automatically performed on the file OUTPUT.

PROCEDURE READ - read logical record (or data item from TEXT file). See section 8.7.2.1 for the form of a text file "read parameter".

PROCEDURE READLN - read next logical record from TEXT file

PROCEDURE WRITE - write logical record (or data item to TEXT file). See section 8.7.2.2 for the form of a text file "write parameter".

PROCEDURE WRITELN - write logical record to TEXT file

PROCEDURE SETNAME(F, PATHNAME) - This procedure is used to bind a logical file name F to a physical file path name. PATHNAME is a string of any length.

PROCEDURE MESSAGE(S) - This procedure is used to write the string S to the system log file.

C.3 HEAP MANAGEMENT ROUTINES

Dynamic memory areas referred to as heap packets may be allocated and deallocated using the procedures NEW and DISPOSE.

PROCEDURE NEW(P) or

PROCEDURE NEW(P, T1, ..., Tn) - This procedure is used to allocate a new dynamic memory area and return a pointer to it in the variable P. The size of the heap packet to be allocated is implicitly equal to the size of the variable which P points to. If P points to a record variable with variants, the tag values T1 through Tn may be given so the allocated packet is exactly as large as needed for the specified variants.

PROCEDURE DISPOSE(P) - This procedure deallocates the heap packet pointed to by P. The value of P is then set to NIL.

C.4 MISCELLANEOUS ROUTINES

FUNCTION PRED(X) - This function returns a value that is the predecessor of X which must be an enumeration type value.

FUNCTION SUCC(X) - This function returns a value that is the successor of X where must be an enumeration type value.

FUNCTION ORD(X) - This function returns the integer ordinal value of X which is of type BOOLEAN, CHAR, or scalar type.

FUNCTION ODD(X) - This function returns the BOOLEAN value TRUE if the INTEGER or LONGINT value X is odd; FALSE otherwise.

FUNCTION ABS(X) - This function returns the absolute value of the INTEGER, LONGINT, or REAL value X.

FUNCTION SQR(X) - This function returns the squared value of the INTEGER, LONGINT, or REAL value X.

FUNCTION CHR(X) - This function returns the character with ordinal value X which must be of type BOOLEAN, INTEGER, or scalar type.

PROCEDURE PACK(A, I, Z) - This procedure packs components of the array A into the packed array Z, starting at the Ith component of A. The component types of the two arrays must be compatible.

PROCEDURE UNPACK(Z, A, I) - This procedure unpacks components of the packed array Z into the array A, starting at the Ith component of A.

FUNCTION SIZE(X) - This function returns the integer size (in bytes) of X which may be a type identifier or a variable.

FUNCTION LOCATION(X) - This function returns the integer location of the unpacked variable or module X.

C.5 CRU ROUTINES

The following standard procedures and functions allow access to the hardware CRU instructions.

PROCEDURE CRUBASE(BASE) - This procedure allows the user to set the CRU base register (R12) to the value specified by the expression BASE which must be of type INTEGER.

PROCEDURE LDCR(WIDTH, VALUE) - This procedure implements the load CRU instruction. WIDTH must be an integer constant which specifies the number of bits of the integer VALUE to be transferred to the specified CRU address implied by the last CRUBASE.

PROCEDURE SBO(DISP) - This procedure implements the CRU instruction SBO which sets the bit to logic one specified by the integer constant displacement DISP from the last CRUBASE.

PROCEDURE SBZ(DISP) - This procedure implements the CRU instruction SBZ which sets the bit to logic zero specified by the integer constant displacement DISP from the last CRUBASE.

PROCEDURE STCR(WIDTH, VALUE) - This procedure implements the store CRU instruction. WIDTH must be an integer constant which specifies the number of bits of the integer the value at the specified CRU address implied by the last CRUBASE to be transferred to VALUE.

FUNCTION TB(DISP) : BOOLEAN - This procedure implements the TB instruction which tests the bit specified by the integer constant displacement DISP from the last CRUBASE. This function returns a BOOLEAN value.

C.6 USER DECLARED UTILITY ROUTINES

The following procedures and functions are not pre-declared in Microprocessor Pascal System but may be declared by the user to be EXTERNAL routines using the declarations shown and invoked to perform the functions indicated.

FUNCTION ARCTAN(X:REAL):REAL - This function returns the arc tangent for the value X specified.

FUNCTION COS(X:REAL):REAL - This function returns the cosine for the value X specified.

FUNCTION EXP(X:REAL):REAL - This function returns the exponential value for the value X specified.

FUNCTION LN(X:REAL):REAL - This function returns the natural logarithm for the value X specified.

FUNCTION SIN(X:REAL):REAL - This function returns the sin for the value X specified.

FUNCTION SQRT(X:REAL):REAL - This function returns the square root for the value X specified.

APPENDIX D

EXECUTIVE RUN TIME SUPPORT REFERENCE CARD

In this appendix we enumerate every user-callable RTS routine. This is intended to be a quick reference for the programmer. The enumeration is categorized; within each category routines are alphabetized.

D.1 PROCESSOR MANAGEMENT (SCHEDULING) ROUTINES

```
type priority = 0..32766;

procedure setpriority( var oldvalue: priority;
  newvalue: priority ); external;
procedure swap; external;
```

D.2 SEMAPHORE ROUTINES

D.2.1 Semaphore Operations

```
type nonneg = 0..32767;
  semaphorestate = ( awaited, zero, signaled );

function cksemaphore( sema: semaphore): boolean; external;
procedure csignal( sema: semaphore;
  var waiter: boolean ); external;
procedure cwait( sema: semaphore;
  var received: boolean ); external;
procedure initsemaphore( var sema: semaphore;
  count: nonneg ); external;
function semastate( sema: semaphore ): semaphorestate; external;
function semavalue( sema: semaphore ): integer; external;
procedure signal( sema: semaphore ); external;
procedure termsemaphore( var sema: semaphore ); external;
procedure wait( sema: semaphore ); external;
procedure waitsignal( waitfor, signalthe: semaphore ); external;
```

D.2.2 Semaphore Attributes

```
type interrupt_level = 0..15;

procedure altexternalevent( sema: semaphore;
  level: interrupt_level ); external;
procedure externalevent( sema: semaphore;
  level: interrupt_level ); external;
procedure noaltexternalevent( level: interrupt_level );
  external;
```

```
procedure noexternalevent( level: interrupt_level );
  external;
```

D.3 INTERRUPT ROUTINES

```
type interrupt_result = -1..15;

function intlevel: interrupt_result; external;
procedure mask; external;
procedure unmask; external;
```

D.4 PROCESS MANAGEMENT

```
type processid = @ processid;

function my$process: processid; external;
procedure p$abort( p: processid ); external;
function p$lastprocess( p: processid ): processid;
  external;
procedure start$term( var oldvalue: boolean;
  newvalue: boolean); external;
function p$successful( p: processid ): boolean;
  external;
```

D.5 MEMORY MANAGEMENT

```
type
  pointer = @ integer . or @any_other_structure  ;
  byte_length = 0..32767;

procedure free$( var ptr: pointer ); external;
procedure heap$term( var oldvalue: boolean;
  newvalue: boolean ); external;
procedure new$( var ptr: pointer; length: byte_length );
  external;
```

D.6 FILE ROUTINES

```
type channel_mode = ( reading, writing, usermode );

procedure close( var f: anyfile ); external;
function  column( var f: text ): integer; external;
procedure f$bspace( var f: text ); external;
procedure f$chabort( var f: anyfile ); external;
procedure f$chbuffers( var f: anyfile;
  minbufs : integer ); external;
function  f$clength( var f: anyfile ): integer; external;
procedure f$conditional( var f: anyfile;
  is_cond: boolean ); external;
procedure f$createchannel( var f: anyfile ); external;
```

```
function  f$eoc( var f: anyfile ): boolean; external;
function  f$lastsuccessful( var f: anyfile ): boolean; external;
procedure f$master( var f: anyfile ); external;
function  f$nextch( var f: text ): char; external;
procedure f$steoc( var f: anyfile ); external;
procedure f$stlength( var f: anyfile; length: integer ); external;
procedure f$stmode( var f: anyfile;
  m: f$$channel_mode ); external;
procedure f$ulength( var f: anyfile ); external;
procedure f$wait( var f: anyfile;
  var users_mode: f$$channel_mode ); external;
procedure f$xaccess( var f: anyfile ); external;
procedure ioterm( var f: anyfile;
  var oldv: boolean; newv: boolean ); external;
procedure page( var f: text ); external;
function  status( var f: anyfile ): integer; external;
```

D.7 ERROR RECOVERY AND EXCEPTION HANDLING

```
procedure ct$enter; external;
procedure ct$exit; external;
function err$class: integer; external;
procedure err$rset; external;
function err$reason: integer; external;
procedure exception( classcode, reasoncode: integer );
  external;
procedure onexception( handler_location: integer );
  external;
procedure re$start; external;
```

D.8 CRU ROUTINES

The following standard routines should not be declared by the user.

```
type base_range = 0..#1FFE;
  width_range = 1..16;
  displacement_range = -128..127;

procedure crubase( base: base_range );
procedure ldcr( width: width_range; value: integer );
procedure sbo( displacement: displacement_range );
procedure sbz( displacement: displacement_range );
procedure stcr( width: width_range; var value: integer );
function tb( displacement: displacement_range ): boolean;
```

D.9 ASSEMBLY LANGUAGE INTERFACE

```
procedure ckof; external;
procedure ckon; external;
procedure idle; external;
procedure lrex; external;
procedure rset; external;
```

APPENDIX E

MICROPROCESSOR PASCAL SYSTEM ERROR AND EXCEPTION CODES

E.1 TX RUN-TIME ERROR MESSAGES

The following list consists of all of the errors that can be generated by the TX executive while using the Microprocessor Pascal System on the TX990; these errors are not generated by the DX10 system. If one of these errors is generated, a message of the following form will be displayed:

```
INTERPRETER ERROR : ee
```

where the "ee" represents the specific error that occurred; the meanings associated with these hexadecimal digits are given in the list below.

01	Invalid Opcode
02	Stack Overflow
03	Invalid Procedure Call
04	Division by Zero
05	Floating Point Error
06	Set Range Error - element <0 or > 1023
07	Assert Error
08	Case Alternative Error
09	Array Index Error
0A	Pointer Error
0B	Subrange Assignment Error
0C	Array Longint Index Error
0D	Longint Subrange Assignment Error
14	Halt Called

E.2 I/O ERROR MESSAGES

The following errors are generated by both the DX10 system and the TX990 system. They represent the errors that occur as a result of invalid file manipulation. The general form of this type of error is:

```
      I/O ERROR : ee  ss  filename
```

where "ee" is the actual error that was encountered; the meaning associated with each value is given in the list below. The "ss" represents the standard SVC status code associated with the particular error that was generated; their meanings can be found in either the TX 990 OPERATING SYSTEM PROGRAMMER'S MANUAL, or the DX 10 OPERATING SYSTEM REFERENCE MANUAL - VOL. 6 ERROR REPORTING AND RECOVERY. The name of the file in which the error was detected is given by "filename".

0	Open Error Status
1	Open State Error
2	Close Error Status
3	Close State Error
4	Read Error Status
5	Read State Error
6	Write Error Status
7	Write State Error

E.2.1 Text I/O Error Messages

Text file errors which occur on either the DX10 system or the TX990 system are generated whenever text files are incorrectly manipulated. When one of theses errors occurs, a message of the following form is displayed:

```
      TEXT FILE I/O ERROR : ee  filename
```

where "ee" represents the specific error that was generated; the meanings associated with these values are given below. The text file connected with the error encountered is given by "filename".

0	Normal completion
1	Parameter out of range
2	Field width too large
3	Incomplete data
4	Invalid character in field
5	Value too large
6	Read past end of file
7	Field exceeds record size

E.3 SYNTAX ERROR MESSAGES

The following is a list of errors that are generated by the compiler. If one of these errors should occur it will appear in the source listing generated by the compiler, positioned directly below where the error was detected. The error is given as an integer value preceded by an "!"; an abbreviated description of each error value is given below.

1	error in simple type
2	identifier expected
3	'SYSTEM' expected
4	')' expected
5	':' expected
7	error in parameter list
8	'OF' expected
9	'(' expected
10	error in type
11	'[' expected
12	']' expected
13	'END' expected
14	';' expected
15	integer constant expected
16	'=' expected
17	'BEGIN' expected
18	error in declaration part
19	error in field list
20	',' expected
22	'..' expected
40	error in copy statement
41	statement expected
43	'FORWARD' or 'EXTERNAL' expected
50	error in constant
51	':=' expected
52	'THEN' expected
53	'UNTIL' expected
54	'DO' expected
55	'TO' or 'DOWNTO' expected
57	'FILE' expected
58	error in factor
59	error in variable
60	HEX expected
80	option identifier expected
81	unknown option identifier
82	system sensitive option not allowed here
83	module sensitive option not allowed here
84	null body expected
85	error in concurrent characteristic specification
101	identifier declared twice
102	lower bound exceeds upper bound
103	wrong kind of identifier

104	identifier not declared
105	sign not allowed
106	number expected
107	incompatible subrange types
108	file not allowed here
110	tagfield type must be scalar or subrange
111	incompatible variant label
113	index type must be scalar or subrange
115	set base type must be scalar or subrange
116	error in type of standard procedure parameter
119	repetition of parameter list not allowed
120	function result type must be scalar, subrange or pointer
121	file value parameter not allowed
122	repetition of result type not allowed
123	missing result type in function declaration
125	error in type of standard function parameter
126	number of parameters does not agree with declaration
127	actual parameter must not be packed
129	type conflict in assignment
130	expression is not a set
131	tests for pointer equality only
132	illegal operator
134	illegal type of operand(s)
135	type of expression must be Boolean
136	set element type must be scalar or subrange
137	set element types not compatible
138	type of variable is not array
139	index type is not compatible with declaration
140	type of variable is not record
141	type of variable must be pointer
142	illegal parameter substitution
143	illegal type of FOR expression
144	illegal type of CASE selector
146	assignment of files not allowed
147	incompatible CASE label
148	subrange bounds must be scalar
149	index type must not be integer
152	no such field in this record
154	actual parameter must be a variable
156	multidefined case label
157	case label range too large
158	missing corresponding variant declaration
160	previous declaration was not forward
161	module declared forward again
162	parameter size must be constant
163	missing variant in declaration
165	multidefined label
166	multideclared label
167	undeclared label
177	assignment to non-local function not allowed
178	multidefined record variant label
179	illegal escape
180	unaccessed common variable
181	assignment to FOR variable not allowed

182	actual reference parameter must not be FOR variable
183	illegal type transfer
184	type of COMMON must not be file
185	file element type must not be file or pointer
186	set bounds out of range
188	division by zero
189	statement must be structured statement
190	label in FOR or WITH statement not allowed
191	variable declarations not allowed at SYSTEM level
192	invalid nesting of SYSTEM, PROGRAM, or PROCESS
193	reference parameters not allowed for PROGRAM or PROCESS
194	pointer parameters not allowed for PROGRAM
195	INPUT or OUTPUT must be declared TEXT
196	INPUT or OUTPUT not declared
201	fraction expected
202	string constant must not exceed source line
203	integer constant exceeds range
206	exponent expected
207	hex digit expected
208	illegal long integer constant
209	nested comments
251	too many nested modules
252	too many modules declared
255	too many errors in this source line
258	too many identifiers declared in list
304	set element out of range
399	internal compiler error

E.4 Executive RTS Error and Exception Codes

The Executive RTS error and exception codes are divided into an integer-valued class and an integer-valued reason code. For a particular class, reason codes are unique to the class. The following constant declarations document all exception codes generated by the Executive RTS.

```
    const

  { system crash codes}

  unable_to_boot_system      = 1;
  no_exception_handler       = 2;
  no_interrupt_handler       = 3;
  illegal_interrupt_or_xop   = 4;
  scheduling_queue_in_error  = 5;
  ROM_RAM_partition_error    = 6;

  { class codes}

  user_error           = 1;
  scheduling_error     = 2;
  semaphore_error      = 3;
  interrupt_error      = 4;
  process_mgmt_error   = 5;
  exception_error      = 6;
  memory_mgmt_error    = 7;
  file_error           = 8;

  interpreter_error    = 99;

  { reason codes}

  { scheduling error}
  scheduling_queue_invalid          = 1;
  scheduling_queue_priority_error = 2;

  { semaphore error}
  semaphore_invalid                     = 1;
  semaphore_count_error                 = 2;
  semaphore_operation_error             = 3;
  semaphore_count_overflow              = 4;
  semaphore_in_handler_priority_error = 5;
```

```
{ interrupt error}
interrupt_invalid                  = 1;
interrupt_level_invalid            = 2;
interrupt_semaphore_invalid        = 3;
interrupt_not_handled              = 4;
interrupt_incorrect_trap_vector    = 5;
interrupt_handler_priority_error = 6;

{ exception error}
exception_handler_not_established_from_process              = 1;
exception_handler_cannot_have_parameters                    = 2;
exception_handler_cannot_be_in_assembly_language            = 3;
exception_handler_local_variables_too_large_for_stack = 4;

{ process mgmt. error}
not_a_process                                        = 1;
aborted                                              = 2;
not_started_invalid_priority                         = 3;
not_started_negative_stacksize                       = 4;
not_started_negative_heapsize                        = 5;
not_started_process_is_in_assembly_language          = 6;
not_started_no_memory_for_semaphore                  = 7;
not_started_no_memory_for_process_heap               = 8;
not_started_no_memory_for_process_stack              = 9;
not_started_no_memory_for_process_frame              = 10;

{ memory mgmt. error}
heap_invalid          = 1;
heap_overflow_error = 2;
heap_packet_error     = 3;

{ file error}
normal_completion                                                      = 0;
text_conversion_parameter_out_of_range                                 = 1;
text_conversion_field_width_too_large                                  = 2;
text_conversion_incomplete_data                                        = 3;
text_conversion_invalid_character_in_text_field                        = 4;
text_conversion_value_too_large                                        = 5;
text_read_past_end_of_file                                             = 6;
text_field_exceeds_record_size                                         = 7;
file_is_not_open_for_reading                                           = 8;
file_is_not_open_for_writing                                           = 9;
sequential_read_past_end_of_file                                       = 10;
no_system_memory_for_file_descriptor                                   = 50;
random_files_not_implemented                                           = 51;
file_component_length_is_incompatible_with_channel                     = 52;
no_system_memory_for_descriptor_of_file_parameter_by_value             = 53;
parameter_to_f$chbuffers_exceeds_255                                   = 54;
file_parameter_to_f$conditional_is_not_sequential                      = 55;
file_parameter_to_f$stlength_is_not_closed                             = 56;
f$stlength_component_length_is_not_in   1..8191                        = 57;
f$stlength_component_length_greater_than_declared_for_file             = 58;
f$reset_called_for_channel_master_before_f$createchannel               = 60;
f$reset_called_for_channel_master_and_master's_mode_is_writing        = 61;
```

```
f$reset_called_for_channel_master_and_master's_mode_is_usermode = 62;
f$reset_called_for_channel_master_after_f$wait_and_user's_mode_ = 63;
          is_reading
f$reset_called_for_channel_master_before_close_and_f$wait        = 64;
f$rewrite_called_for_channel_master_before_f$createchannel       = 65;
f$rewrite_called_for_channel_master_and_master's_mode_is_reading= 66;
f$rewrite_called_for_channel_master_before_f$wait                = 67;
f$rewrite_called_for_channel_master_and_user's_mode_is_writing   = 68;
f$rewrite_called_for_channel_master_before_close_and_f$wait      = 69;
f$master_called_and_file_not_closed                              = 70;
f$master_called_twice_for_same_file                              = 71;
no_system_memory_for_f$master_structures                         = 72;
f$eoc_called_and_f$steoc_not_called_for_file                     = 73;
file_parameter_to_f$steoc_is_not_channel_master                  = 74;
f$steoc_called_after_f$createchannel                             = 75;
parameter_to_f$stmode_is_not_in_  reading, writing, usermode     = 76;
file_parameter_to_f$stmode_is_not_channel_master                 = 77;
f$stmode_called_after_f$createchannel                            = 78;
file_parameter_to_f$ulength_is_not_channel_master                = 79;
f$ulength_called_after_f$createchannel                           = 80;
f$createchannel_called_before_f$master                           = 81;
f$createchannel_called_before_f$stmode                           = 82;
f$createchannel_called_twice                                     = 83;
file_parameter_to_f$wait_is_not_channel_master                   = 84;
f$wait_called_and_f$createchannel_not_called                     = 85;
file_parameter_to_f$wait_is_not_closed                           = 86;
file_parameter_to_f$xaccess_is_not_channel_master                = 87;
f$xaccess_called_after_f$createchannel                           = 88;
conditional_read_or_write_failed (nonfatal error)                = 103;
channel_aborted                                                  = 104;
no_system_memory_for_channel_buffers                             = 106;
no_system_memory_for_channel                                     = 200;
no_system_memory_for_pathname                                    = 201;
invalid_pathname                                                 = 202;
attempt_to_open_device_in_an_unsupported_mode                    = 203;
device_channel_not_initialized_before_user_connected             = 204;
attempt_to_initialize_device_channel_with_same_name_as_          = 205;
          existing_user_channel
attempt_to_open_multiple_device_channels_of_same_name_with_      = 206;
          conflicting_modes

{ interpreter error}
{ run time errors}
invalid_opcode                = 1;
stack_overflow                = 2;
unresolved_procedure_call     = 3;
division_by_zero              = 4;
floating_point_error          = 5;
set_element_out_of_bounds     = 6;
assert_error                  = 7;
missing_otherwise_in_case     = 8;
array_index_error             = 9;
pointer_equals_nil            = 10;
subrange_assignment_error     = 11;
```

```
longint_array_index            = 12;
longint_subrange_error         = 13;
escape_to_exception_handler    = 19;
run_time_support_error         = 20;
```

E.5 TX/DX SVC I/O ERROR CODES

The following is a partial list of SVC I/O errors which could be generated by the Microprocessor Pascal Sytem.

Code (Hexadecimal)	Description	
00	No Error	
01	Illegal Luno	
02	Illegal Operation Code	
03	Luno Is Not Yet Opened	
04	Record Lost Due To Power Failure	
05	Illegal Memory Address	
06	Time Out, or Abort	
07	Read Check Error	
11	Device Error	
12	No Address Mark Found	
15	Data Check Error	
19	Diskette Not Ready	
1A	Write Protect	
1B	Equipment Check Error	
1C	Invalid Track or Sector	
1D	Seek Error or ID Not Found	
1E	Deleted Sector Detected	
20	Luno Is In Use	
21	Bad Disc Name	
22	Pathname Has a Syntax Error Luno Previously Assigned	(TX) (DX)
23	Illegal Operation Code	
24	Bad Parameter in PRB	
25	Diskette Is Full	
26	Duplicate File Name	
27	File Name Is Undefined	
28	Illegal Luno	
29	System Buffer Area Full	
2A	System Can't Get Memory	
2B	File Management Error	
2C	Can't Release System Luno	
2D	File Is Protected	
2E	Abnormal File Management Termination	
2F	File Utility Doesn't Exist In System	
30	Non-Existent Record - File Not Initialized	
31	Event Key SVC Task Not In System	
3B	Invalid Access Privilege	
3E	File Control Block Error	
3F	File Directory Full	

APPENDIX F

MICROPROCESSOR PASCAL SYSTEM VS WIRTH'S PASCAL

F.1 SPECIAL SYMBOLS

Tne following Microprocessor Pascal System special symbols are not supported in the Wirth and Jensen version of Pascal.

" # ::

F.2 KEYWORDS

The following Microprocessor Pascal keywords are not supported in Pascal:

ACCESS ANYFILE ASSERT COMMON
ESCAPE LONGINT OTHERWISE PROCESS
RANDOM SEMAPHORE START SYSTEM

The following Microprocessor Pascal keywords are predefined identifiers in Pascal: BOOLEAN CHAR FALSE INPUT INTEGER OUTPUT REAL TEXT TRUE

F.3 IDENTIFIERS

In Pascal, identifiers may not contain the symbols $ or _. Most versions of Pascal impose a restriction on the maximum number of significant characters in an identifier; Microprocessor Pascal does not make that restriction.

The following standard Pascal identifiers are not known in Microprocessor Pascal System:

ARCTAN COS EXP GET
LN MAXINT PUT SIN
SQRT

F.4 CONSTANTS

Hexadecimal and LONGINT constants are not sıpported in Pascal.

Pascal does not allow hexadecimal characters embedded within string or character constants.

F.5 REMARKS

Remarks are not supported in Pascal, only conventional comments are available.

F.6 SYSTEM AND PROCESS DECLARATIONS

Pascal does not support a SYSTEM or PROCESS declaration or a START statement to invoke them. Pascal only supports conventional Pascal programs.

F.7 CONSTANT DECLARATIONS

Integer constant expressions are not allowed in the constant declaration section of a Pascal program.

F.8 COMMON AND ACCESS DECLARATIONS

Pascal does not support COMMON or ACCESS declarations.

F.9 PROCEDURE OR FUNCTION PARAMETERS

Microprocessor Pascal System does not support procedures or functions to be passed as parameters to other procedures or functions as Pascal does.

F.10 STANDARD DATA TYPES

The standard data types LONGINT, SEMAPHORE, and ANYFILE are not supported in Pascal.

F.11 SUBRANGE LOWER BOUNDS

In Pascal, a subrange must have a lower bound that is strictly less than the upper bound rather than possibly equal to the upper bound.

F.12 TYPE TRANSFER

Type structures of Pascal variables may not be overridden by performing a type-transfer as in Microprocessor Pascal System.

F.13 OPERATOR PRECEDENCE

Microprocessor Pascal System uses an operator precedence similar to that of ALGOL and FORTRAN as opposed to that of Pascal.

F.14 START STATEMENT

The START statement is not supported in Pascal. Only conventional Pascal programs are recognized.

F.15 ESCAPE STATEMENT

Pascal does not support the ESCAPE statement.

F.16 ASSERT STATEMENT

The ASSERT statement is not supported in Pascal.

F.17 GOTO STATEMENT

Microprocessor Pascal limits GOTOs to the local routine only; global GOTO statements are not allowed.

F.18 CASE STATEMENT

Pascal does not support the OTHERWISE clause in CASE statements.

F.19 FOR STATEMENT

The control variable of a FOR statement must be explicitly declared in Pascal.

F.20 WITH STATEMENT

Pascal supports only one form of the WITH statement; the <identifier> = <record variable> form is known only in Microprocessor Pascal System.

F.21 SEQUENTIAL I/O

Pascal's sequential input and output is handled by the GET and PUT statements, and the file variable pointer; Microprocessor Pascal uses READ and WRITE to perform I/O operations on sequential files.

F.22 HEX OUTPUT

Pascal does not support the HEX output option which is available in Microprocessor Pascal System

F.23 RANDOM FILES

Pascal does not support RANDOM files.

F.24 ENCODE AND DECODE

The ENCODE and DECODE procedures are not supported in Pascal.

F.25 STANDARD PROCEDURES AND FUNCTIONS

The following standard procedures and funtions are not supported in Pascal:

LINT LOCATION LROUND LTRUNC
MESSAGE SETNAME SIZE

The CRU routines which are supported by Microprocessor Pascal System are not allowed in Pascal.

F.26 OPTIONS

Microprocessor Pascal System and Pascal support entirely different sets of compiler options. The form of option specification is also significantly different between Pascal and Microprocessor Pascal System.

APPENDIX G

Microprqcessor Pascal System VS TIP

G.1 KEYWORDS

The following TIP keywords are not supported in Microprocessor Pascal System:

DECIMAL FIXED

The following Microprocessor Pascal System keywords are not supported in TIP: ANYFILE PROCESS SEMAPHORE START SYSTEM

G.2 CONSTANTS

Decimal, fixed, and extended precision real constants are not supported in Microprocessor Pascal System.

G.3 SYSTEM AND PROCESS DECLARATIONS

TIP does not support a SYSTEM or PROCESS declaration or a START statement to invoke them.

G.4 CONSTANT EXPRESSIONS

Microprocessor Pascal System supports integer constant expressions only in the CONST section and does not support any other type of constant expression in the CONST section.

G.5 PROCEDURE OR FUNCTION PARAMETERS

Microprocessor Pascal System does not support procedures or functions to be passed as parameters to other procedures or functions.

G.6 QUESTION MARK PARAMETERS

Microprocessor pascal System does not support question mark upper bound array or set parameters.

G.7 FUNCTION SIDE EFFECTS

Microprocessor Pascal System does not enforce function side effects rules described in TIP.

G.8 EXTERNAL ROUTINES

Microprocessor Pascal System only supports external Pascal routines but not external Fortran or external Cobol routines. External Fortran may be supported in future Microprocessor Pascal versions.

G.9 STANDARD DATA TYPES

Microprocessor Pascal System does not support the standard data types FIXED or DECIMAL. It also only supports the default precision of REAL. TIP does not support the standard data types ANYFILE and SEMAPHORE.

G.10 DYNAMIC ARRAYS AND SETS

Microprocessor Pascal System does not support dynamic arrays or sets.

G.11 PACKING ALGORITHM

In Microprocessor Pascal structures always occupy a full number of words and may not be packed with other elements in a word.

G.12 TYPE COMPATIBILITY

In Microprocessor Pascal System, records and arrays must be non-distinct types to be compatible. In Microprocessor Pascal System, sets are compatible if they have compatible base types, but they may have different lengths.

G.13 START STATEMENT

The START statement is not supported in TIP.

14 Goto Labels

Microprocessor Pascal does not detect when a GOTO statement jumps into a FOR or WITH statement, but it does flag every label within a FOR or WITH

G.15 CASE STATEMENT

In Microprocessor Pascal the range of the case labels may be no greater than 256.

G.16 FOR STATEMENT

The FOR I IN ¼set½ DO form of the FOR statement is not supported in Microprocessor Pascal System.

G.17 FORMATTED TEXT INPUT

Formatted text input is not supported in Microprocessor Pascal System.

G.18 READ VARIABLES

Microprocessor Pascal does not allow the variables in a READ statement to be elements of a packed structure.

G.19 RANDOM FILE I/O

Microprocessor Pascal only allows RANDOM file READ and WRITE statements to have a single read or write variable.

G.20 WRITE PARAMETERS

In Microprocessor Pascal System, both RANDOM and sequential files require write parameters which are variables; expressions are not allowed as write parameters.

G.21 STANDARD PROCEDURES AND FUNCTIONS

The following TIP standard functions are not supported in Microprocessor Pascal System:

DEC FIX UB

The following TIP standard functions may be user declared in Microprocessor Pascal System:

COLUMN STATUS ARCTAN COS
EXP LN SIN SQRT

The following TIP standard procedures are not supported in Microprocessor Pascal System:

CLOSE DATE EXTEND HALT
IOTERM PAGE SETMEMBER SKIPFILES
TIME WRITEEOF

In Microprocessor Pascal System, the standard function FLOAT does not allow the second parameter which specifies the precision of the real value.

In Microprocessor Pascal System, the standard procedure SETNAME allows an arbitrary length string for the pathname rather than only a 8 character string.

In Microprocessor Pascal System, the following standard procedures are provided to access the CRU hardware:

- CRUBASE
- LDCR
- SBO
- SBZ
- STCR
- TB

G.22 OPTIONS

Microprocessor Pascal System does not support the following options:

- 370
- 980
- 990
- CKOVER
- CKPREC
- CKTAG
- FORINDEX
- GLOBALOPT
- GLOBALS
- LISTOBJ
- OPTIMIZE
- PROBER
- PROBES
- ROUND
- STANDARD
- UNSAFEOPT
- WARNINGS
- WIDELIST

APPENDIX H

EXECUTIVE RUN-TIME SUPPORT VS TIPMX

H.1 INTRODUCTION

The TI Pascal Microprocessor Executive (TIPMX) is the predecessor of the Executive RTS. TIPMX must be used with object modules generated by the TI Pascal (TIP) compiler, not the Microprocessor Pascal (MPP) System compiler. The capabilities of the Executive RTS are generally a superset of those of TIPMX. This section describes the changes that may have to be made to convert a TIPMX application to execute under the Executive RTS. Appendix G of the Microprocessor Pascal System User's Manual enumerates the differences between the Microprocessor Pascal System and TIP languages.

H.2 CRU ACCESS

All references to the CRU must be modified to use the new procedures and function provided by the Microprocessor Pascal System language. The CRU interface was changed so the Microprocessor Pascal System compiler can generate more efficient CRU accesses than the TIP compiler can.

H.3 PROCESS DECLARATION AND INVOCATION

The TIP language has no concept of "process" so the procedure STARTPROCESS is used to create and invoke an instance of procedure as a process. Every TIP procedure that is used as a process must be converted into a Microprocessor Pascal System, program, or process, with appropriate concurrent characteristics.

```
  TIPMX:
    { process}  procedure clock( interrupts: integer );

  Executive RTS:
    program clock( interrupts: integer );
    ...
    begin
      {# stacksize = clock_stack_size; priority = clock_priority}
      ...
    end . clock  ;
```

A process is started under TIPMX by a call to procedure STARTPROCESS. The statement START is used with MPP System, and successful creation is indicated by the function P$SUCCESSFUL.

```
TIPMX:
  startprocess( location( clock ), clock_stack_size, clock_priority,
    interrupts_per_tick, process_number, successful );

Executive RTS:
  start clock( interrupts_per_tick );
  successsful := p$successful( my$process );
```

Stack and heap requirements have to be recalculated for Executive RTS and specified as concurrent characteristics.

H.4 TERMINATION OF SEMAPHORES

Each semaphore in TIPMX is local to the process in which it is declared and is deallocated automatically when that process terminates. A variable of type SEMAPHORE in Microprcessor Pascal System contains a reference to an RTS-managed structure that implements the semaphore. If the resources associated with a semaphore are to be reclaimed by the Executive RTS, the last process that uses a semaphore must call procedure TERMSEMAPHORE with that semaphore as parameter.

H.5 SYNCHRONIZATION WITH INTERRUPTS

The Executive RTS treats interrupts as implicit signals to associated semaphores. This means that the Executive RTS procedure WAIT is used for synchronization with interrupts and SIGNAL can be used to simulate an interrupt. To convert TIPMX interrupt processes, a semaphore must be declared within scope and initialized to zero. This semaphore must then be associated with the appropriate interrupt level using the Executive RTS procedure EXTERNALEVENT. All occurrences of WAITINTERRUPT must then be changed to WAIT(s) where "s" is the interrupt semaphore. For example:

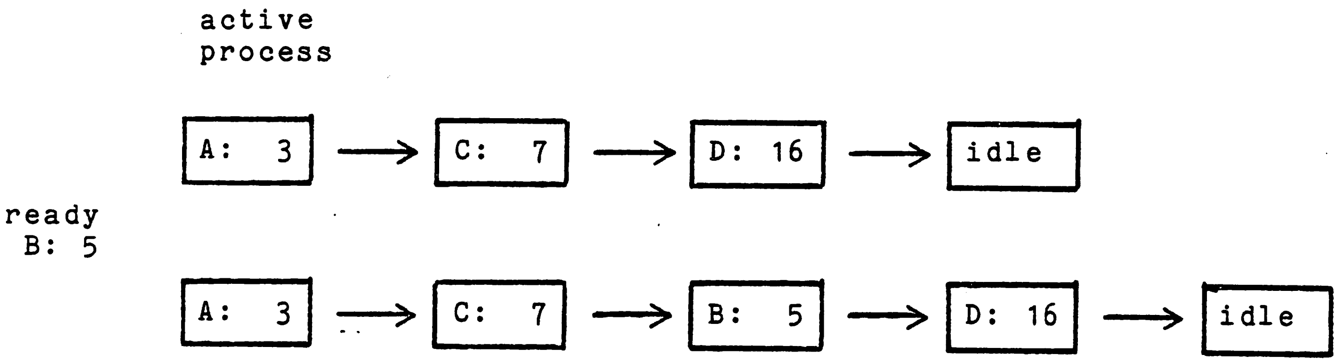
```
TIPMX:
  { process} clock( interrupts: integer );
    begin
    while true do begin
      for i := 1 to interrupts do begin
        ckon$;
        waitinterrupt;
        ckof$;
        end { for} ;
      signal( tick )
      end { while true}
    end { clock} ;
```

```
Executive RTS:
  program clock( interrupts: integer );
  var interrupt: semaphore;
  begin
    {# stacksize = clock_stack_size; priority = clock_priority}
    initsemaphore( interrupt, 0 );
    externalevent( interrupt, clock_priority );
    while true do begin
      for i := 1 to interrupts do begin
        ckon;
        wait( interrupt );
        ckof
        end { for } ;
      signal( tick )
      end { while true}
  end { clock} ;
```

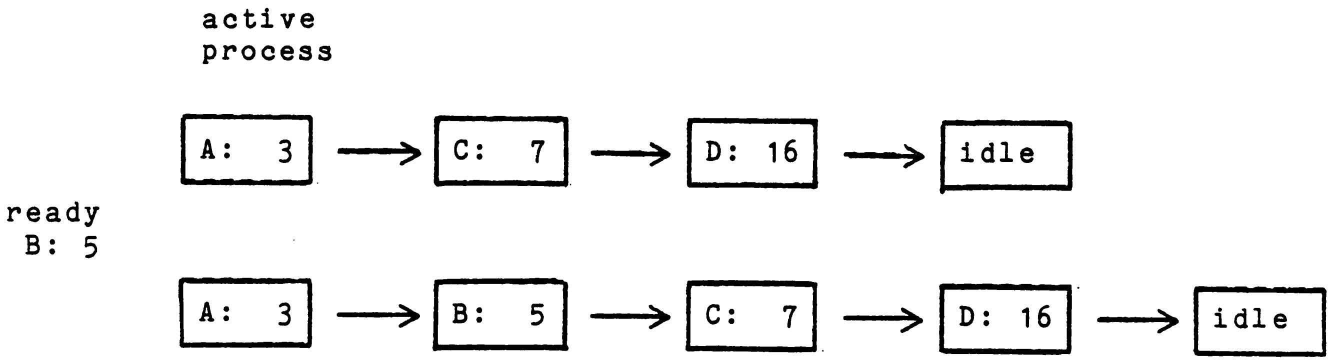
H.6 SCHEDULING POLICY

The scheduling policy of the Executive RTS is simpler to understand and more consistent than that of TIPMX. The differences pertain to the treatment of device processes.

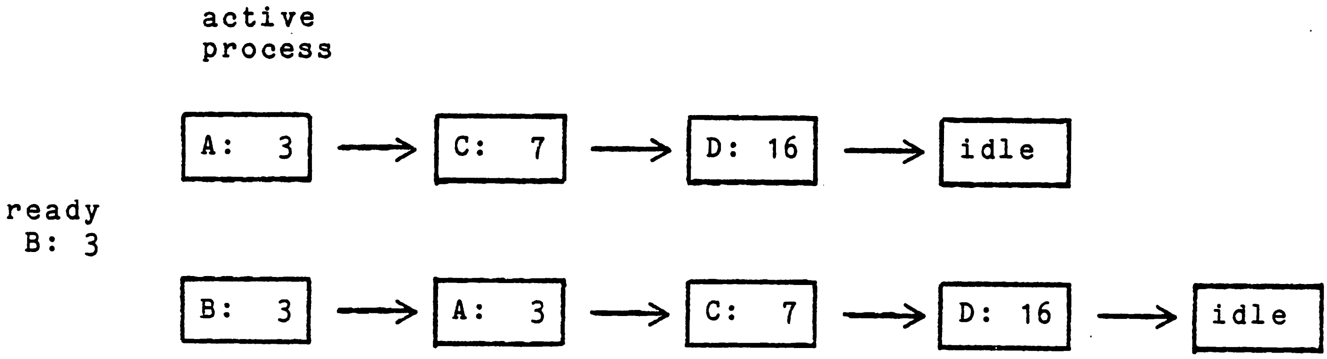
For example, suppose device process A activates a device process B that has a lesser urgency than A. Under TIPMX, process B becomes a non-interrupt process:



(See Section 9.1 for an explanation of these diagrams.) Note that the arrangement of the scheduling queue is not consistent with the urgencies of the processes. Under the Executive RTS, priorities are preserved:



This difference should present no problems since the activation of one device process from another usually occurs when interrupts are being demultiplexed, in which case both processes will have the same priority and hence the same treatment under both executives:



H.7 INTERPROCESS FILES

The concept of a "channel" has been introduced into the MPX interprocess file system. An MPX channel is basically equivalent to a TIPMX file connection. TIPMX allows multiple writing files and a single reading file in a single file connection while MPX allows both multiple reading and multiple writing files on a single channel. There are several system file routines available in MPX to facilitate the implementation of device handlers that are not available in TIPMX. The standard function FILENAMED is avaliable in MPX to specify the names of files passed as parameters to processes. This capability does not exist in TIPMX.

APPENDIX I

BNF of MICROPROCESSOR PASCAL SYSTEM

I.1 GENERAL The syntax of a programming language describes the form which a program in the language may take. In a language such as Microprocessor Pascal, the syntax may be expressed very concisely by extended Backus-Naur Form or BNF.

In BNF, each element of the language is defined by means of equation- like rules called productions, in which the entity being defined is written to the left of the symbol "::=" and the definition is written to the right of that symbol. The definition may be expressed in terms of language elements which are defined by previous or subsequent productions. The following symbols are used in writing definitions:

::=   for productions

< >   for enclosing non-terminal symbols, i.e. entities which are defined by a production

[ ]   for enclosing entities which are optional

{ }   for enclosing entities which may be repeated zero or more times

¦   for representing alternation, e.g. A ¦ B ¦ C means A or B or C

I.2 DECLARATION SYNTAX

```
<system               ::= SYSTEM <identifier> ; <system block> .

<system block         ::= <label declaration part>
                          <constant declaration part >
                          <type declaration part>
                          <common declaration part>
                          <access declaration part>
                          <system routines>
                          <body>

<label declaration part>::=
                          LABEL <statement label>{ , <statement label>} ;
                          ¦<empty>

<empty>               ::=

<statement label>     ::= <digit> { <digit>}

<constant declaration part> ::=
                          CONST <constant declaration>
                          { <constant declaration>}
                          ¦ <empty>

<constant declaration> ::=
                          <identifier> = <constant> ;
                          ¦ <identifier> = <integer constant expression> ;

<type declaration part> ::=
                          TYPE <type declaration> { <type declaration>}
                          ¦ <empty>

<type declaration> ::= <identifier> = <type> ;

<variable declaration part> ::=
                          VAR <variable declaration>
                          {< variable declaration>}
                          ¦ <empty>

<variable declaration> ::=   identifier list> : <type> ;

<identifier list>     ::= <identifier> { , <identifier>}

<common declaration part> ::=
                          COMMON <variable declaration>
                          { <variable declaration>}
                          ¦ <empty>

<access declaration part> ::=
                          ACCESS <identifier list> ;
                          ¦<empty>

<system routines>     ::= { <system routine>}
```

```
<system routine>    ::= <program declaration> ¦ <procedure declaration>
                     ¦<function declaration>

<program declaration> ::=
                    <program header> <program block> ;
                     ¦ <program header> FORWARD ;
                     ¦ <program header> EXTERNAL   PASCAL   ;
<program header>    ::= PROGRAM <identifier>
                     ¦<program parameter list>   ;

<program parameter list> ::=
                    ( <program parameter> { ; <program parameter>} )

<program parameter>::= <identifier list> : <type identifier>

<program block>     ::= <label declaration part>
                    <constant declaration part>
                    <type declaration part>
                    <variable declaration part>
                    <common declaration part>
                    <access declaration part>
                    <program routines>
                    <body>

<program routines> ::= { <program routine>}

<program routine>  ::= <process declaration   <procedure declaration>
                     <function declaration>

<procedure declaration> ::=
                    <procedure header> <block> ;
                     ¦ <procedure header> FORWARD ;
                     ¦ <procedure header> EXTERNAL   PASCAL   ;

<procedure header> ::= PROCEDURE <identifier>  < parameter list>   ;

<parameter list>    ::= ( <any parameter> { ; <any parameter>} )

<any parameter>     ::=  VAR  <identifier list> : <type identifier>

<block>             ::= <label declaration part>
                    <constant declaration part>
                    <type declaration part>
                    <variable declaration part>
                    <common declaration part>
                    <access declaration part>
                    <routines>
                    <body>

<routines >         ::= { <routine>}

<routine >          ::= <procedure declaration> ¦<function declaration>
```

```
<function declaration> ::=
                   <function header> <block> ;
                   ! <function header> FORWARD ;
                   ! <function header> EXTERNAL   PASCAL   ;

<function header>  ::= FUNCTION <identifier>  <parameter list>   :
                   <result type> ;

<process declaration> ::=
                   <process header> <program block> ;
                     <process header> FORWARD ;
                     <process header> EXTERNAL   PASCAL   ;

<process header>   ::= PROCESS <identifier>
                   [<program parameter list> ]  ;

<body>             ::= <compound statement>
```

I.3 TYPE SYNTAX

```
<type>                ::= <simple type> | <structured type>

<simple type>         ::= <scalar type> | <subrange type>
                          | <type identifier>

<type identifier>     ::= <identifier> | ANYFILE | BOOLEAN | CHAR|
                          | INTEGER | LONGINT | REAL | SEMAPHORE | TEXT

<scalar type>         ::= ( <scalar identifier> { , <scalar identifier> } )

<subrange type>       ::= <enumeration constant> .. <enumeration constant>

<enumeration constant> ::=
                          <character constant> | <boolean constant>
                          | <scalar identifier> | <integer constant>

<scalar identifier> ::= <identifier>

<structured type>     ::= [ PACKED] <unpacked structured type>
                          |<pointer type> | <file type> | <set type>

<unpacked structured type> ::=
                          <array type> | <record type>

<array type>          ::= ARRAY "[" <index type> { , <index type> }  "]"
                          OF <type>

<index type>          ::= BOOLEAN | CHAR | <scalar type> | <subrange type>
                          | <identifier>

<record type>         ::= RECORD <field list> END

<field list>          ::= <fixed part> | <fixed part> ; <variant part>
                          | <variant part>

<fixed part>          ::= <record section> { ; <record section>}

<record section>      ::= <field identifier> { , <field identifier> } :
                         <type  |  <empty

<field identifier> ::= <identifier>

<variant part>        ::= CASE  [  tagfield> ] <tagfield type> OF
                          <variant>{ ; <variant>}

<tagfield type>       ::= BOOLEAN | CHAR | INTEGER | LONGINT | <identifier>

<tagfield>            ::= <identifier> :

<variant>             ::= <variant label list> : ( <field list> )
                          | <empty>
```

```
<variant label list> ::=
                   <variant label> { ,  variant label>}

<variant label>    ::= <enumeration constant>
                   | <enumeration constant> {. <enumeration constant>}

<set type>         ::= SET OF <simple type>

<pointer type>     ::= ^ <type identifier>

<file type>        ::= [ RANDOM ] FILE OF <type>

<result type>      ::= BOOLEAN | CHAR | INTEGER | LONGINT | REAL
                   | SEMAPHORE | <identifier>
```

I.4 STATEMENT SYNTAX

```
<statement>            ::= [<statement label> : ] <simple statement>
                        ¦ [<statement label> : ] [ <escape label> :]
                       <structured statement>

<simple statement> ::= <empty statement> ¦ <assignment statement>
                       ¦<procedure statement> ¦ <start statement>
                       ¦<escape statement> ¦ <goto statement>
                       ¦<assert statement>

<empty statement>  ::= <empty>

<assignment statement> ::=
                       <variable> := <expression>

<procedure statement> ::=
                       <procedure identifier> [ <actual parameter list>]

<procedure identifier> ::= <identifier>

<actual parameter list> ::=
                       ( <actual parameter> { , <actual parameter>}  )

<actual parameter> ::= <expression> ¦ <variable>

<start statement>  ::= START <process identifier>
                       [<actual parameter list >]

<escape statement> ::= ESCAPE <escape label>
                       ¦ESCAPE <routine identifier>

<escape label>     ::= <identifier>

<routine identifier> ::=
                       <program identifier> ¦<process identifier>
                       ¦ <procedure identifier> ¦ <function identifier>

<goto statement>   ::= GOTO <statement label>

<assert statement> ::= ASSERT <expression>

<structured statement> ::=
                       <compound statement> ¦ <conditional statement>
                       ¦ <repetitive statement> ¦ <with statement>

<compound statement> ::=
                       BEGIN <statement  { ; <statement>} END

<conditional statement> ::=
                        if statement  ¦ <case statement>

<if statement>     ::= IF <expression> THEN <statement>
                       [ ELSE <statement>]
```

```
<case statement>    ::= CASE <expression> OF <case element>
                    { ; <case element>}
                    [ OTHERWISE <statement> { ; <statement>} ]
                    END

<case element>      ::= <case label list> : <statement> | <empty>

<case label list    ::= <case label> { , <case label>}

<case label     ::= <enumeration constant>
                    | <enumeration constant> .. <enumeration constant>

<repetitive statement> ::=
                    <for statement> | <while statment>
                    | <repeat statement

<for statement>     ::= FOR <control variable> <generator> DO
                    <statement>

<control variable> ::= <identifier>

<generator>         ::= := <initial value> TO <final value>
                    | := <initial value> DOWNTO <final value>

<initial value>     ::= <expression>

<final value>       ::= <expression>

<while statement>   ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> { ; <statement>}
                    UNTIL <expression>

<with statement>    ::= WITH <with variable list> DO <statement>

<with variable list> ::=
                    <with variable> { , <with variable>}

<with variable>     ::= <record variable>
                    | <identifier> = <record variable>
```

I.5 EXPRESSION SYNTAX

```
<expression>         ::= <boolean term>
                     ¦ <expression> OR <boolean term>

<boolean term>       ::= <boolean factor>
                     ¦ <boolean term> AND <boolean factor>

<boolean factor>     ::=  [NOT] <boolean primary>

<boolean primary>  ::= <simple expression> ¦ <boolean primary>
                    <relational operator> <simple expression>

<relational operator> ::= = ¦ <> ¦ <  ¦ <= ¦  > ¦ >= ¦ IN

<simple expression>::= <term> ¦ <adding operator>  term
                     ¦ <simple expression> <adding operator> <term>

<adding operator>  ::= + ¦ -

<term>               ::= <factor> ¦ <term> <multiplying operator> <factor>

<multiplying operator> ::= * ¦ / ¦ DIV ¦ MOD

<factor>     •        ::= ( <expression> )
                     ¦ <function identifier> [<actual parameter list>]
                     ¦ <set> ¦ <unsigned constant> ¦ <variable>

<function identifier> ::=
                    <identifier>

<set>                ::= "[" [<element list>] "]"

<element list>       ::= <element> { , <element> }

<element>            ::= <expression> ¦ <expression> .. <expresssion>

<unsigned constant> ::= <constant identifier>   <boolean constant>
                     ¦<scalar identifier> ¦ NIL
                     ¦ <character constant> ¦ <string constant>
                     ¦ <integer constant> ¦ <real constant>

<constant identifier> ::= <identifier>
```

I.6 VARIABLE SYNTAX

```
<variable>              ::= <variable identifier> ¦ <component variable>
                        ¦ <type-transferred variable>

<variable identifier> ::= <identifier>

<component variable> ::=
                        <indexed variable> ¦ <field designator>
                        ¦ <referenced variable>

<indexed variable> ::= <array variable> "[" <expression>
                        { , <expression>}  "]"

<array variable>    ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable>   ::= <variable>

<referenced variable> ::= <pointer variable>

<pointer variable> ::= <variable>

<type-transferred variable> ::=
                        <variable> :: <type identifier>
```

I.7 INTEGER CONSTANT EXPRESSION SYNTAX

```
<integer constant expression> ::=
                   <integer constant term>
                   ¦ <adding operator> <integer constant term>
                   ¦ <integer constant expression> <adding operator>
                   <integer constant term>

<integer constant term> ::=
                   <integer constant factor>
                   ¦ <integer constant term>  intmult operator>
                   <integer constant factor>

<intmult operator> ::= * ¦ DIV ¦ MOD

<integer constant factor> ::=
                   ( <integer constant expression> )
                   ¦ <integer constant identifier>
                   ¦ <integer constant >

<integer constant identifier> ::= <identifier>
```

I.8 LEXICAL SYMBOL SYNTAX

```
<symbol>              ::= <special symbol> ¦ <keyword symbol>
                      ¦<identifier> ¦ <constant>

<constant>            ::= <enumeration constant> ¦ <real constant>
                      ¦<string constant> ¦ <constant identifier>

<separator>           ::= <space> ¦ <end of the logical source record>
                      ¦ <comment> ¦ <remark>

<comment>             ::= <open comment> <any sequence of graphic
                      characters not containing <close comment> >
                      <close comment>

<open comment>        ::= "{" ¦ (*

<close comment>       ::= "}" ¦ *)

<remark>              ::= " <any sequence of graphic characters
                      extending to the end of the logical
                      source record>

<special symbol>      ::= +¦-¦*¦/¦=¦<¦>¦ (¦)¦.¦,¦;¦:¦@¦"[" "]"
                      ¦(.¦.)¦<=¦>=¦<>¦..¦:=¦::

<keyword symbol>      ::= ACCESS ¦ AND ¦ ANYFILE ¦ ARRAY ¦ ASSERT ¦ BEGIN
                      ¦ BOOLEAN ¦ CASE ¦ CHAR ¦ COMMON ¦ CONST ¦ DIV ¦ DO
                      ¦ DOWNTO ¦ ELSE ¦ END ¦ ESCAPE ¦ FALSE ¦ FILE
                      ¦ FOR ¦ FUNCTION ¦ GOTO ¦ IF ¦ IN ¦ INPUT ¦ INTEGER
                      ¦ LABEL ¦ LONGINT ¦ MOD ¦ NIL ¦ NOT ¦ OF ¦ OR
                      ¦ OTHERWISE ¦ OUTPUT ¦ PACKED ¦ PROCEDURE
                      ¦ PROCESS ¦ PROGRAM ¦ RANDOM ¦ REAL ¦ RECORD
                      ¦ REPEAT ¦ SEMAPHORE ¦ SET ¦ START ¦ TEXT ¦ THEN
                      ¦ TO ¦ TRUE ¦ TYPE ¦ UNTIL ¦ VAR ¦ WHILE ¦ WITH

<identifier>          ::= <letter> { <letter>¦ _ ¦ <digit> }

<letter>              ::= A¦B¦C¦D¦E¦F¦G¦H¦I¦J¦K¦L¦M¦N¦O¦P¦Q¦R
                      ¦S¦T¦U¦V¦W¦X¦Y¦Z¦$

<digit>               ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9

<boolean constant> ::= FALSE ¦ TRUE

<character constant> ::= '<character>'

<string constant>  ::= '<character> <character> { <character> } '

<character>           ::= <graphic character> ¦ #<hexdigit> <hexdigit>

<graphic character> ::= <special character> ¦ <letter> ¦ <digit>
                      ¦<space> ¦ <nonstandard;character>
```

```
<special character> ::=¦+¦-¦*¦/¦=¦<¦>¦(¦)¦.¦,¦;¦:¦@¦''¦"¦##¦_¦"["¦"]"
                    ¦"{"¦"}"

<space>             ::= " "

<nonstandard character> ::=
                    <any other character available on a particular
                    system or device>

<hexdigit>          ::= <digit> ¦ A ¦ B ¦ C ¦ D ¦ E ¦ F

<integer constant>  ::= <digits> [ L ]
                    ¦ # <hexdigit> { <hexdigit> } [L]

<digits>            ::= <digit> { <digit>}

<real constant>     ::= <digits> { <digits> }
                     <digits>   <digits> E <scale factor>
                     <digits> E <scale factor>

<scale factor>      ::= [ <sign> ] <digits>

<sign>              ::= + ¦ -
```

APPENDIX J

INTERPRETIVE RTS DATA STRUCTURES

This appendix describes the data structures used by the the Interpretive Run Time Support. The user initialized data structures are discussed first, such as the RAM configuration table, the segment table, and the trap table. The Run Time Support data structures are discussed next, such as the process record, global data structures, and process local data structures. The data structures used by the interpreter are discussed, such as the interpreter's registers and local variables, and stack areas, and finally the interpreted code itself is discussed.

J.1 USER INITIALIZED DATA STRUCTURES

The data structures described in this section must be initialized by the user and exist in either the "USERINIT" module or the "CONFIG" module.

J.1.1 RAM Configuration Table

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

TABLE J-1. RAM MEMORY

#00	length	Length of contiguous RAM (16-bit logical value in number of bytes)
#02	start address	Address at which contiguous RAM starts
	* * *	Length and start are repeated for each contiguous RAM area.
	length=0	End of table is indicated by a length of zero.

J.1.2 Segment Table

The following configuration information is included anywhere in the user's code space (ROM). An Interpretive RTS-based system starts in a user-written code module ("USERINIT") from application of power or from toggling an external reset switch. This module must perform low-level initialization and branch to Interpretive RTS initialization at entry zero in segment 63. Before branching to Interpretive RTS, register 11 (R11) must point to the structure shown in Table J-2 below.

TABLE J-2. SEGMENT TABLE

	configuration	Address of RAM configuration table
-#08	address of interp. entry	Entry point address of interpreter (REF and DATA of the symbol $EXEC)
-#06	crash address	Address in user-written code to service a system crash. Crash code is in R0.
-#04	address of re$start BLWP	Address of BLWP vector used by RE$START routine.
-#02	address of LREX BLWP	Address of BLWP vector which is copied to >FFFC.
#00	segment table * * *	<-- Address of segment table

The crash address points to user-written code at which Interpretive RTS branches if a system crash occurs. At the crash, the workspace is the current machine workspace, and register zero (R0) contains the crash code.

The address of the LREX BLWP vector, if not zero, is used to copy this vector to absolute address >FFFC in the case that RAM is located at>FFFC. If the address of the LREX BLWP vector is zero, then no copy is done.

The segment table consists of addresses of each valid interpretive segment in the entire system being executed. The segment table may contain up to 64 total segments. Each segment consists of a dictionary table which has entries for all routines and common modules which may be accessed by the routines in the segment.

J.1.3 Traps Configuration Table

The following configuration information is included anywhere in the user's code space (ROM). An Interpretive RTS-based system starts in a user-written code module ("USERINIT") from application of power or from toggling an external reset switch. This module must perform low-level initialization and branch to Interpretive RTS initialization at entry zero in segment 63. Before branching to Interpretive RTS, register zero (R0) may be zero indicating that trap vectors are already initialized in place at absolute address zero. Otherwise register zero must point to the following structure.

```
#00   -------------------- <-- R0 points here at entry to RTS
      |int level 0 WP    |      Interrupt level 0 workspace
#02   --------------------
      |int level 0 PC    |      Interrupt level 0 program counter
      --------------------
      |        *         |
      |        *         |
      |        *         |
#3C   --------------------
      |int level 15 WP   |      Interrupt level 15 workspace
#3E   --------------------
      |int level 15 PC   |      Interrupt level 15 program counter
#40   --------------------
      |XOP level 0 WP    |      XOP level 0 workspace
#42   --------------------
      |XOP level 0 PC    |      XOP level 0 program counter
      --------------------
      |        *         |
      |        *         |
      |        *         |
#7C   --------------------
      |XOP level 15 WP   |      XOP level 15 workspace
#7E   --------------------
      |XOP level 15 PC   |      XOP level 15 program counter
      --------------------
```

J.2 INTERPRETIVE RUN TIME SUPPORT DATA STRUCTURES

The data structures described in this section are used by Interpretive Run Time Support to manage processes, and the memory area associated with a process. The process record is the fundamentiel data structure used by Interpretive Run Time Support. From it one can get to all other data structures used by the Interpretive Run Time Support. All data structures except the process record are given in alphabetical order.

J.2.1 Process Record

The process record is the fundamental structure which is used by Interpretive RTS to access all other data structures. A unique process record exists for each instantiation of a process. The pointer returned by the RTS functions MY$PROCESS and P$LASTPROCESS point to a process record. Note that fields indicated by a "*" are not used by Kernel RTS. The layout of the process record is shown below:

#00	level 0	Display level 0 frame pointer
#02	level 1	Display level 1 frame pointer
#04	* * *	
#14	level 10	Display level 10 frame pointer
#16	unused	
#20	stack base	Base of stack address for process
#22	stack limit	End of stack address for process
#24	stack boundary	Maximum amount of stack used address
#26	global frame	Global stack frame for process
#28	local frame	Currently active stack frame for process
#2A	top of stack	Top of evaluation stack for process
#2C	program counter	Address of next instruction to execute
#2E	segment table	Address of segment table
#30	current segment	Address of current segment table dictionary
#32		

Process Record (continued)

#32	"output" file	Address of "output" file descriptor
#34	"input" file	Address of "input" file descriptor
#36	unused	
#38	priority	Priority of the process
#3A	packed data	Successflag (1 bit) (returned by the p$successful function). Interrupt level in progress when this process was activated (5 bits). Negative one means no interrupt in progress. Unused (2 bits)
#3B	packed data	Unused (4 bits) Interrupt mask of the process (4 bits)
#3C	RTS record	Address of executive record
#3E	packed data	Exception outstanding boolean (1 bit) (If true, then an exception has occurred but the process has not yet failed because it is nested with RTS code or a user-defined critical transaction.) Exception class code (7 bits)
#3F	reason code	Exception reason code
#40	except. handler dictionary entry	* Index in dictionary of segment where exception handler procedure is located.
#41	packed data	* Segment number where exception handler procedure is located (7 bits) * External boolean is always TRUE (1 bit)
#42	first rts frame	* Address of stack frame at which RTS code was first entered.
#44	rts nesting count	* Number of times currently nested within RTS code.
#45		

2.1 Process Record (continued)

#45	packed data	Unused (4 bits) * Type of object upon which RTS code is operating (4 bits) (none=0, exception=1, file=2, heap=3, interrupt=4, process=5, queue=6, semaphore=7, scheduling queue=8, critical transaction=9).
#46	object pointer	* Address of object upon which RTS code is operating
#48	next process of all processes	Address of next process record in a circular, one-way list of all process records.
#4A	last started	Address of process record last started by this process.
#4C	next process in queue	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.
#4E	verification	Address of this process record (used to verfiy a processid referencing this process record.
#50	queue pointer	Address of the queue record in which this process is enqueued (semaphore or scheduling queue or nil)
#52	creator's id	See explanation below.
#53	my id	See explanation below.

The field of the process record called "my id" (displacement #53) is set to a value as follows. A global count MOD 32767 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 #52) in a new process record is set to the value of "my id" of the process which created the new process.

J.2.2 Channel Record

The following record is referenced by Channel Control Records and by File Descriptors. It is used for buffer management. This record is not used in Kernel RTS.

#00	monitor	Address of monitor record to control access to channel record
#02	first empty	Address of first empty buffer record
#04	last empty	Address of last empty buffer record
#06	empty present	Semaphore upon which a producer waits for an empty buffer.
#08	first full	Address of first full buffer record
#0A	last full	Address of last full buffer record
#0C	full present	Semaphore upon which a consumer waits for a full buffer.
#0E	component length	Channel component length
	total buffers	Total buffers (8 bits)
#0F	channel status	Channel status (8 bits)

J.2.3 Channel Buffer Record

Several of the following records form a linked list starting at a Channel Record or one record may be referenced by a File Descriptor. This record is not used in Kernel RTS.

#00	next	Address of next channel buffer record
#02	reply channel	Unimplemented
#04	buffer * * *	

J.2.4 Channel Control Record

The following record is referenced from a pathname record and exists for each channel. It is used for synchronization during channel connections. This record is not used in Kernel RTS.

#00	channel	Address of channel record
#02	packed data	End-of-consumption significant boolean (1 bit) Number of consuming files (7 bits)
#03	producers	Number of producing files
#04	minimum buffers	Minimum channel buffers
#05	packed data	Device channel boolean (1 bit) Unused (7 bits)
#06	master	Address of master record if device channel boolean is TRUE.

J.2.5 Channel Directory Record

The following record is referenced by the Executive Record and exists once in RTS. This record is not used in Kernel RTS.

#00	monitor	Address of monitor record to control access to channel directory.
#02	temp sema	Temporary semaphore used during channel termination synchronization
#04	pathnames	Address of first pathname record
#06	tmp	80 byte card buffer

J.2.6 Executive Record

The following record exists once in the Run Time Support system and points to all other fundamental data structures. Every process record has a pointer to this record.

#00	interrupt record	Address of interrupt record
#02	active process	Address of the active process record.
#04	ready queue	Address of queue record holding all ready processes not more urgent than active.
#06	verification	Address of location(#02) above.
#08	system memory	Address of heap record for all data space used by Interpretive RTS.
#0A	process mgmt.	Address of process management record.
#0C	channel direct.	* Address of channel directory record.

J.2.7 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 Kernel RTS.

#00	column index	Current column index in line buffer (one relative index)
#02	last column	Index of last column in line buffer (one relative index)
#04	line buffer	Address of data in channel buffer record
#06	channel	Address of channel record (NIL if file is closed)
#08	error status	
#0A	reply channel	Unimplemented
#0C	pathname	Address of pathname record
#0E	minimum buffers	Minimum channel buffers
#0F	packed data	End-of-consumption significant boolean (1 bit) Terminate on error boolean (1 bit) Conditional boolean (1 bit) Unused (3 bits) File type - sequential=0, text=1, random=2 (2 bits)
#10	packed data	Component length (13 bits) File state - closed=0, open-writing=1, eoc-writing=2, open-reading=3, eof-reading=4 (3 bits)
#12	verification	Address of this file descriptor
#14	master	Address of master record
#16		

2.7 File Descriptor (continued)

#16	process files	Address of next file descriptor in a linked list of files declared by by this process. This linked list is created when this process terminates.
#18	length	For a sequential file, this is the declared component length. For a text file, this is the maximum default line length.

J.2.8 Heap Record

The system process has a heap, each program has a heap, and each process with a non-zero heapsize concurrent characteristic has a heap. Each heap is administered through the following heap record. A heap record is referenced from each process mark.

#00	free packet	Address of free packet
#02	minimum ptr.	Value of smallest valid heap pointer
#04	maximum ptr.	Value of largest valid heap pointer
#06	parent heap	Address of heap record in which this heap is nested
#08	maximum used	Maximum amount of allocated space ever used in this heap (16-bit logical value of number of bytes)
#0A	current used	Current amount of allocated space used in this heap (16-bit logical value of number of bytes)
#0C	mutex semaphore	Semaphore ensuring mutually exclusive access to heap data structures
#0E	verification	Address of this heap record

J.2.8.1 Free Heap Packet. A heap packet which is not allocated must be at least six bytes in length and has the following format.

#00	size	Size of this packet (16-bit logical value of number of bytes)
#02	previous ptr.	Address of previous free heap packet
#04	next ptr.	Address of next free heap packet
#06	* * *	Remainder of packet

J.2.8.2 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.

-#02	size+1		Size of this packet plus one (16-bit logical value of number of bytes)
#00	* * *	<-- pointer	Remainder of packet

J.2.9 Interrupt Record

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

#00	external event	Level 0 external event semaphore
#02	alternate event	Level 0 alternate event semaphore
#04		Level 1 external, alternate semaphores
	* * *	
#3C		Level 15 external, alternate semaphores
#40	no event	"No event" semaphore to which the above 32 semaphores are connected in default state.

J.2.10 Master Record

The following is referenced by the File Descriptor of a master file. This record is not used in Kernel RTS.

#00	user connects	Semaphore upon which master process waits for first user to connect
#02	packed data	Channel mode - reading=0, writing=1, user-mode=3, no-mode=4 (2 bits) Exclusive access boolean (1 bit) Boolean indicating that component length is to be specified by user (f$ulength routine) (1 bit) Channel created boolean (1 bit) Users connected boolean (1 bit) User mode - reading=0, writing=1, user-mode=3, no-mode=4 (2 bits) Unused (5 bits) Mode of master file - closed=0, open-writing=1, eoc-writing=2, open-reading=3, eof-reading=4 (3 bits)

J.2.11 Monitor Record

The following data structure is used by low-level synchronization in RTS while managing a channel directory or a channel. This record not used in Kernel RTS.

#00	urgent	Urgent semaphore
#02	mutex	Mutual exclusion semaphore
#04	verification	Address of this monitor record

J.2.12 Pathname Record

Several of the following records form a linked list starting at the Channel Directory Record. This record is not used in Kernel RTS.

#00	next	Address of next pathname record
#02	reading cc	Address of reading channel control record
#04	writing cc	Address of writing channel control record
#06	cc terminates	Semaphore upon which file openers wait when channel is closing (or not ready) or channel has a master specifying exclusive access. All waiters on this semaphore are signaled when channel is terminated or reset.
#08	packed data	Device pathname boolean (1 bit) Number of references to pathname (15 bits)
#0A	pathname * * *	Character string of pathname (first byte is length)

J.2.13 Process Management Record

The following record is referenced by the Executive Record and exists once in RTS. In the "full" RTS, when a process terminates, it switches from its normal stack and executes termination RTS code in the following stack area.

#00	number of starts	Number of started processes MOD 32767
#02	unused	
#04	mutex semaphore	Semaphore ensuring mutual exclusion in process management
#06	stack * * *	Stack in which a process executes that is terminating itself

J.2.14 Queue Record

A queue record is referenced by a Semaphore Record or by the ready queue field of the Executive Record.

#00	first member	Address of process record of first member of queue
#02	verification	Address of this queue record

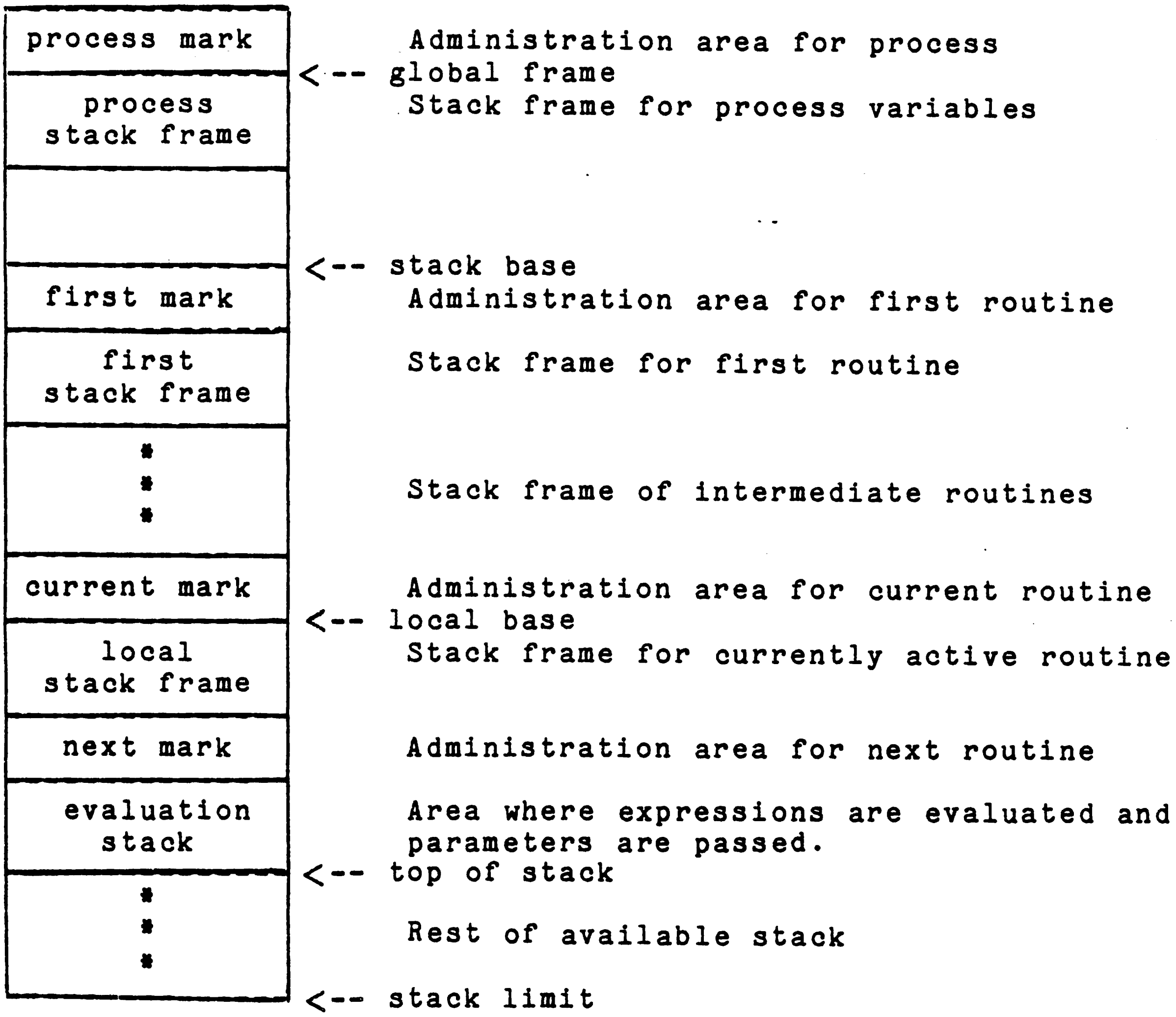
J.2.15 Semaphore Record

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

#00	count	Count of semaphore (returned by SEMAVALUE function).
#02	waiters	Address of queue record in which waiting processes are enqueued.
#04	verification	Address of this semaphore record

J.3 PROCESS STACK

The stack for a process is allocated as two separate regions, the first is the stack frame for the process frame, and the second is the stack to be used by routines which are called from the process. The second stack region is disposed when a process terminates. The first stack region is disposed when process variables contained in it are no longer addressable by the process or any lexically nested processes. The two regions have the following format:



```
process mark            Administration area for process
                  <--  global frame
process stack frame     Stack frame for process variables

                  <--  stack base
first mark              Administration area for first routine
first stack frame       Stack frame for first routine
*
*                       Stack frame of intermediate routines
*
current mark            Administration area for current routine
                  <--  local base
local stack frame       Stack frame for currently active routine
next mark               Administration area for next routine
evaluation stack        Area where expressions are evaluated and
                        parameters are passed.
                  <--  top of stack
*
*                       Rest of available stack
*
                  <--  stack limit
```

J.3.1 Stack Frame

Each stack frame contains the values of the variables and parameters for a given routine. This area may also contain temporaries used for "for" statements and "with" statements. The stack frame consists of four regions, any of which may be zero length. A stack frame is shown below:

parameters	Parameter variables
structured parameters	Structured value parameters are copied into this area.
local variables	Local variables
temporaries	Compiler generated temporaries

J.3.2 Administration Area

The administration area is used to describe the currently active routine and to describe how to return to the caller of the current routine. The administration area has the following form:

-#0E	CRU base	Current CRU base address
-#0C	statement #	Current statement number
-#0A	old PC	Program counter of caller
-#08	old display	Old value of display level
-#06	old dictionary	Segment dictionary of caller
-#04	old local base	Local base of caller
-#02	descriptor	Current routine descriptor
		<-- Frame pointer

J.3.3 Process Mark

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

-#1C	mutex semaphore	Semaphore ensuring mutually exclusive access to this process mark.
-#1A	process files	Address of first file descriptor in a linked list of files declared by by this process. This linked list is created when this process terminates.
-#18	process heap	Address of heap record for this process
-#16	packed data	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). Lexical level of this process (5 bits).
-#15	references	Number of references to this process frame by processes which can address this frame.
-#14	priority	Priority of this process
-#12	"output" file	Address of "output" file descriptor
-#10	"input" file	Address of "input" file descriptor
-#0E	CRU base	Current CRU base address
-#0C	statement #	Current statement number
-#0A	old PC	Contains zero
-#08	old display	Contains zero
-#06	old dictionary	Contains zero
-#04	old local base	Global frame pointer of lexical parent if the lexical level of this process is >= 2, otherwise contains zero.
-#02	descriptor	Routine descriptor of process

<-- Global frame pointer

J.4 DICTIONARY TABLE

Each interpretive segment begins with a dictionary table which has entries for all routines and common modules which may be accessed by the routines in that segment. The dictionary may contain up to 256 total entries, and each entry will be one of the following:

internal	Displacement from beginning of segment dictionary to the routine descriptor for the "internal" module.
external	"External" module entry for a routine which is not local to this segment. The upper byte contains the index in the external segment of the external module, and the lower byte contains the (segment table number * 2) + 1 which indicates that it is an external module entry.
common	"Common" module entry contains the address of the common data area.

J.5 ROUTINE DESCRIPTOR

The routine descriptor contains information about a routine which is used when the routine is called, such as the lexical level, data frame size, parameter size, and start of code for routine. The routine descriptor also contains a label table which is used by jump instructions and by the escape instruction. Each entry in the label table and the start of code entry are relative displacements from the beginning of the routine descriptor to the location in the interpretive code for that label. The first label in the label table is used for routine "escapes" and points to the epilogue for the routine.

When the routine is a "system", "program", or "process", there is one additional word in front of the routine descriptor which contains the data frame size for the process and the frame size in the descriptor includes only the size of the parameter area after any structured parameters have been copied.

When a segment has been "saved" with "debug" information, each routine descriptor has additional information preceding the routine descriptor which provides information for debugging. This information includes the routine name, number of statements in the routine, and number of temporaries in stack frame used for FOR and WITH statement variables.

The complete routine descriptor is shown below:

-#0C	temporary size	Size of temporary variable area (bytes)
-#0A	# statements	Number of statements in routine
-#08	routine name	Name of routine (6 characters)
-#02	frame size	Process frame size (bytes)
#00	start of code	Start of code (displacement)
#02	parameter size	Parameter area size (bytes)
#04	frame size	Data frame size for this routine (bytes)
#06	lexical level	Lexical level of routine (*2)
#08	label table * * *	Label Table
	routine code	Interpretive code for routine

J.6 INTERPRETER REGISTERS AND LOCAL VARIABLES

The registers and data area used by the interpreter is described in this section. The interpreter's data area is shown below which includes its registers.

#00	registers	Interpreter's work space
#20	unused	
#26	cswitch	Address of context switch handler
#28	process	Address of current process
#2A	unused	
#2C	debug handler	Address of routine entry/exit debug handler
#2E	error handler	Address of error handler
#30	trace word	AMPL debugging trace word
#32	debug flags	AMPL debugging flags
#34	break point table	AMPL breakpoint table (table is terminated with a zero)
#50		

Registers R0 through R6 are temporary registers used by the interpreter. Some of the registers used of the interpreter have special purposes. These registers are described below:

R7 - This register normally contains the address of the decode instruction handler. When an interrupt has occurred, this address is changed to the context switch handler so that after the current instruction has be executed, the interrupt process can be given control.

R8 - This register contains a pointer to the current process record.

R9 - This register points to the work space for the interpreter.

R10 - This register points to an area of available memory which may be used when the interpreter does a "BLWP R10".

R11 - This register contains the address of the instruction handler or the return address of a "BL" instruction.

R12 - This register contains the instruction opcode. The upper byte is > 10 and the lower byte is the instruction opcode. This word is used by the instruction decoder to select the appropiate instruction handler. When an assembly language module is called, this register points to the first word of the module.

R13 - This register contains the address of the next interpreted instruction to be executed.

R14 - This register contains the address of the top of the evaluation stack in the process stack. During an expression evaluation, values are pushed and popped from this stack. This register points to the next available word.

R15 - This register contains the address of the currently active routine's stack frame. It points to the routine's first variable and the administration area is immediately in front of the stack frame.

The AMPL area is used by the Target Debugger so that breakpoints and traces can be handled. The AMPL flag word contains flags which are used to indicate whether single step, process trace, routine entry/exit trace, and/or statement trace are to be performed.

J.7 INTERPRETIVE CODE DESCRIPTION

The interpretive code is designed to be as compact as possible. Since the major design goal is to minimize the length of the code gemerated, the most frequently used operations have a short form in which the operand is folded into the opcode.

Instructions in the interpretive code are specified by a one-byte opcode followed by zero or more operand bytes. The operands are one of four basic types:

UB - unsigned byte. The value of the operand is in one byte and in the range 0..255.

SB - signed byte. The value of the operand is in one byte and in the range -128..127.

V - variable length. This operand is one or two bytes in length. The operand is one byte long if the first byte is in the range 0..127 (most significant bit = 0). Otherwise, the most significant bit of the first byte is 1 and operand requires two bytes in the range 128..32767.

W - word. This operand is two bytes long and aligned on a word boundary

When an operation is to be performed on a LONGINT or REAL or a comparison of a SET or STRING, an escape opcode is followed by the actual opcode, such as a LOP operator followed by an ADD operator.

All operators manipulate the evaluation stack by loading (pushing) values onto the stack and storing (pulling) values from the stack. For most operand types, the value alone is stored on the stack, but for string operands only the address of the string is stored on the stack, and for set operands the length of the set is pushed on top of the set value.

Each interpretive code operator is described in the following tables by giving the hexadecimal value of the opcode, the symbolic name with possible operands, and finally the state of the evaluation stack before and after the operation has been performed. In some cases the semantics of the operation are given instead of the state of the stack.

op	name	before - stack - after	
00	EQ	OPERAND2 OPERAND1	OPERAND1 = OPERAND2
01	GE	OPERAND2 OPERAND1	OPERAND1 >= OPERAND2
02	GT	OPERAND2 OPERAND1	OPERAND1 > OPERAND2
03	LE	OPERAND2 OPERAND1	OPERAND1 <= OPERAND2
04	LT	OPERAND2 OPERAND1	OPERAND1 < OPERAND2
05	NE	OPERAND2 OPERAND1	OPERAND1 <> OPERAND2
06	ADD	OPERAND2 OPERAND1	OPERAND1 + OPERAND2
07	DIV	OPERAND2 OPERAND1	OPERAND1 / OPERAND2
08	MOD	OPERAND2 OPERAND1	OPERAND1 MOD OPERAND2
09	MUL	OPERAND2 OPERAND1	OPERAND1 * OPERAND2
0A	SUB	OPERAND2 OPERAND1	OPERAND1 - OPERAND2
0B	NEG	OPERAND	- OPERAND
0C	ABS	OPERAND	ABS(OPERAND)
0D	SQR	OPERAND	SQR(OPERAND)
0E	CVI	operand is converted from INTEGER to specified type	
0F	CVL	operand is converted from LONGINT to specified type	
10	CVR	operand is converted from REAL to specified type	
11	CVBI	operand below top of stack is converted from INTEGER to specified type	
12	CVBL	operand below top of stack is converted from LONGINT to specified type	

op	name	before - stack - after	
13	LDX,V	ADDRESS	load value at ADDRESS + V words
14	STX,V	ADDRESS VALUE	store VALUE at ADDRESS + V words
15	LDG,V		load value at global frame + V words
16	STG,V	VALUE	store VALUE at global frame + V words
17	LDL,V		load value at local frame + V words
18	STL,V	VALUE	store VALUE at local frame + V words
19	RTNF,V	return from a function and leave value at local frame + V words on top of stack	
1A	ODD	VALUE	ODD(VALUE)
1B	RNDR	VALUE	ROUND(VALUE)
1C	IXB	LOWER INDEX ADDRESS	load byte at (INDEX - LOWER) bytes from ADDRESS
1D	IXB0	INDEX ADDRESS	load byte at INDEX bytes from ADDRESS
1E	IXB1	INDEX ADDRESS	load byte at (INDEX - 1) bytes from ADDRESS

1F	IXP,UB	LOWER
		INDEX
		ADDRESS
			BITDISP := (LOWER - INDEX) MOD UB;
			ADDRESS := ADDRESS + (LOWER - INDEX) DIV UB;

20	IXP0,UB	INDEX
		ADDRESS
			BITDISP := LOWER MOD UB;
			ADDRESS := ADDRESS + LOWER DIV UB;

21	IXP1,UB	INDEX
		ADDRESS
			BITDISP := (LOWER - 1) MOD UB;
			ADDRESS := ADDRESS + (LOWER - 1) DIV UB;

22	DECT,V	decrement temporary at local frame + V words by one	
23	INCT,V	increment temporary at local frame + V words by one	
24	LDT,V		load temporary at local frame + V words
25	STT,V	VALUE	store VALUE in temporary at local frame + V words

```
op   name        before - stack - after
-----------------------------------------------------------------
26    IDX,V      LOWER           ADDRESS := ADDRESS + (INDEX - LOWER)
                 INDEX                  * V * 2;
                 ADDRESS

27    IDXO,V     INDEX           ADDRESS := ADDRESS + INDEX * V * 2;
                 ADDRESS

28    IDX1,V     INDEX           ADDRESS := ADDRESS + (INDEX - 1) * V * 2;
                 ADDRESS

29    CIDX,W     LOWER           LOWER      check if INDEX is between
                 INDEX           INDEX      LOWER and W (upper bound)

2A    CIDXO,W    INDEX           INDEX      check if INDEX is between
                                            0 and W (upper bound)

2B    CIDX1,W    INDEX           INDEX      check if INDEX is between
                                            1 and W (upper bound

2C    CSUB,W1,W2    VALUE        VALUE      check if VALUE is between
                                            W1 (lower) and W2 (upper)

2D    JMPG,UB,V     jump to label UB if value at local frame + V words
                    is greater than value at local frame + V + 1 words

2E    JPG,SB,V      add SB to PC if value at local frame + V words is
                    greater than value at local frame + V + 1 words

2F    JMPL,UB,V     jump to label UB if value at local frame + V words
                    is less than the value at local frame + V + 1 words

30    JPL,SB,V      add SB to PC if value at local frame + V words
                    is less than the value at local frame + V + 1 words

31    JMP,UB        jump to label UB

32    JP,SB         add SB to PC

33    JMPF,UB   VALUE            if VALUE = FALSE then jump to label UB

34    JPF,SB    VALUE            if VALUE = FALSE then add SB to PC

35    JMPT,UB   VALUE            if VALUE = TRUE then jump to label UB

36    JPT,SB    VALUE            if VALUE = TRUE then add SB to PC

37    JMPX,UB,W1,W2,(JUMP TABLE)    case jump

38    JPX,SB,W1,W2,(JUMP TABLE)     case jump
```

op	name	before - stack - after	
39	DIF	SET2 SET1	SET1 - SET2
3A	INN	SET VALUE	VALUE IN SET
3B	INT	SET2 SET1	SET1 * SET2
3C	SEL	VALUE	VALUE
3D	SRG	UPPER LOWER	LOWER .. UPPER
3E	UNI	SET2 SET1	SET1 + SET2
3F	LABL		label comment
40	LDNIL		NIL
41	LIN		INPUT file
42	LOUT		OUTPUT file
43	LDB	ADDRESS	load byte at ADDRESS
44	STB	VALUE ADDRESS	store byte VALUE at ADDRESS
45	ADD1	VALUE	VALUE + 1
46	SUB1	VALUE	VALUE - 1
47	MARK		function call mark
48	RTNP		return from procedure
49	ASSRT		assert error
4A	CASE		case alternative error
4B	CPTR	VALUE	error if VALUE = nil
4C	INCA,V	ADDRESS	ADDRESS + V * 2
4D	LAG,V		load address of global frame + V words
4E	LAL,V		load address of local frame + V words
4F	MOV,V	SOURCE DEST	move V words from SOURCE address to DEST address

op	name	before - stack - after	
50	COM,UB		load address of common UB
51	CSP,UB		call standard procedure UB
52	CUP,UB		call local procedure UB
53	CXP,UB		call external procedure UB
54	ECP,UB		escape to level UB
55	LDC,UB		load constant UB
56	LDP,UB	BITDISP ADDRESS	load field in word at ADDRESS starting at bit BITDISP of length UB bits
57	LDPS,UB	BITDISP ADDRESS	load field in word at ADDRESS starting at bit BITDISP of length UB bits with sign extension
58	LEX,UB		load base of lexical level UB
59	LOCP,UB		load address of procedure UB
5A	STP,UB	VALUE BITDISP ADDRESS	pack VALUE into word at ADDRESS starting at bit BITDISP of length UB bits
5B	LAC,UB,string		load address of string constant of length UB bytes
5C	ADJ,UB		adjust set so that it is UB words long
5D	LDCM,UB,set		load set constant of length UB words
5E	LDM,UB	ADDRESS	load set at ADDRESS of length UB words
5F	STM,UB	SET ADDRESS	store SET at ADDRESS of length UB words
60	STAT,UB		statement number UB
61	CSET,W	SET	error if set is larger than W words
62	LDC,W		load constant W
63	LDCL,W1,W2		load LONGINT constant W1 and W2
64	LDCR,W1,W2		load REAL constant W1 and W2

op	name	before - stack - after	
65	LOP,OP		next operator is LONGINT opcode
66	ROP,OP		next operator is REAL opcode
67	SETOP,OP		next operator is SET opcode
68	STGOP,OP,UB		next operator is STRING opcode where the strings are UB bytes long
69	CRU,OP		next operator is CRU operator
6A	SCOM,OP		statement comment
6B			
*	unused opcodes		
*			
6F			
70	LDC,0		load constant 0
*			
*			
7F	LDC,15		load constant 15
80	LDL,0		load value at local frame + 0 words
*			
*			
8F	LDL,15		load value at local frame + 15 words
90	STL,0	VALUE	store VALUE at local frame + 0 words
*			
*			
9F	STL,15	VALUE	store VALUE at local frame + 15 words
A0	LDG,0		load value at global frame + 0 words
*			
*			
AF	LDG,15		load value at global frame + 15 words
B0	JPF,1	VALUE	jump forward 1 byte if VALUE = FALSE
*			
*			
BF	JPF,16	VALUE	jump forward 16 bytes if VALUE = FALSE
C0	JP,1		jump forward 1 byte
*			
*			
CF	JP,16		jump forward 16 bytes

op	name	before - stack - after	
D0	LDX,0	ADDRESS	load value at ADDRESS + 0 words
*			
*			
D7	LDX,7	ADDRESS	load value at ADDRESS + 7 words
D8	STX,0	VALUE ADDRESS	store VALUE at ADDRESS + 0 words
*			
*			
DF	STX,7	VALUE ADDRESS	store VALUE at ADDRESS + 7 words
E0	LAL,0		load address of local frame + 0 words
*			
*			
E7	LAL,7		load address of local frame + 7 words
E8	LEX,1		load address of lexical level 1
*			
*			
EF	LEX,8		load address of lexical level 8
F0	INCA,1	ADDRESS	ADDRESS + 2
*			
*			
F7	INCA,8	ADDRESS	ADDRESS + 16
F8			
*	unused opcodes		
*			
FF			

APPENDIX K

TX SYSGEN EXAMPLE

This appendix describes system generation procedures for generating a TX990 operating system capable of supporting the Microprocessor Pascal Development System. It is assumed that the user is generally familiar with TX990 sysgen; if not, refer to the TX990 Operating System Programmer's Guide (Release 2).

To execute the Microprocessor Pascal Development System on a machine under TX990, the machine must have the following minimal hardware configuration:

- A 911 Video Display Terminal (VDT)
- Two floppy drives
- 28K words of memory

Texas Instruments supplies a software development system which includes TX990 and supports the 911 VDT, 810 line printer. The user may wish to generate a TX990 operating system to support functions for some of the following reasons:

- Delete floating point package to provide more user memory (about 1K words)
- To include support for a listing device other than the 810 line printer
- To eliminate the CMD key abortion capability
- To increase the nesting level of COPY files that may be used.

Generating a system requires the following:

- The TX990 Parts Diskette (part number 937803-0001)
- A 911 VDT TXDS system diskette (part number 937803-0005)
- The Microprocessor Pascal TX990 Sysgen Diskette

The procedures include those for executing the following programs:

- GENTX - To generate source code for TASKDF and TXDATA modules required for TX990
- OBJMGR - To select and concatenate object modules onto file :TXPART/OBJ for input to TXSLNK

- TXMIRA - To assemble modules TASKDF and TXDATA
- TXSLNK - To link modules TASKDF, TXDATA, and TXPART to form a new system load module on file :MPPSYS/SYS
- SYSUTL - To establish :MPPSYS/SYS as the system file on the newly created system diskette.

To generate a new TX990 system perform the following steps:

1. Place the TX990 parts diskette in the right-hand floppy disk unit and the TXDS system diskette in the left-hand unit. Place both units in ready.

2. Press the HALT/SIE switch on the programmer panel of the computer.

3. Press the RESET switch on the same panel.

4. Press the LOAD switch on the same panel.

5. The computer loads the system and displays the message similiar to the following:

```
TX990           2.3.0  78.244
MEMORY SIZE(WORDS): 28672    AVAILABLE: 20012
```

6. Enter an exclamation point (!) at the keyboard of the operator station. The computer displays a message similar to the following:

```
TXDS           2.3.0  78.244

PROGRAM:
```

7. Enter the following information to execute the GENTX utility to generate source code for system modules TASKDF and TXDATA:

```
PROGRAM:  :GENTX/*
```

8. GENTX requests parameters as shown below. Enter the values shown. Where no value is shown, the default applies. Depressing the RETURN key terminates each line. A value other than the default may be entered if required.

```
GENTX           2.3.0  78.244 TX990 SYSGEN

MEMORY AVAILABLE - 2000

LINE FREQ. -
TIME SLICING(Y OR N) - Y
TIME SLICE VALUE -
```

```
TASK SENTRY(Y OR N) - N
COMMON SIZE - 0                 * no common area needed
# OF EXPASION CHASSIS -
CHASSIS - 0
```

9. Define the operator station. The operator station may be either 911 VDT or a 913 VDT.

```
DEV NAME - ME                   * the assumed name
DEV TYPE - A911                 * AMPL 911
STATION # - 1
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

10. Define the floppy drive device.

```
DEV NAME - DSC
DEV TYPE - FD
CRU BASE ADDR -
INT LEVEL -
# OF DRIVES - 2                 * enter number of drives
```

11. Define any additional devices such as the line printer.

```
DEV NAME - LP
DEV TYPE - LP
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

12. Enter carriage returns for the following (leave blank) to terminate the device definition portion of GENTX:

```
  DEV NAME -                    * no more devices
CHASSIS -                       * no more chassis
```

13. Enter the following information to define the standard TXDS tasks:

```
SVC # -                         * no user defined SVCs
XOP # -                         * no user defined XOPs

TASK ID # ->F0                  * floppy disk drive 1
  PRIORITY LEVEL - 0
  INITIAL STATE -
  INITIAL DATA LABEL - FMP1
TASK ID # ->F1                  * floppy disk drive 2
  PRIORITY LEVEL - 0
  INITIAL STATE -
  INITIAL DATA LABEL - FMP2
```

14. When more than two floppy disk drives are required, repeat the floppy disk drive sequence for each additional drive. The task ID for the third drive is >F2 and the initial data label is FMP3. The task ID for the fourth disk drive is >F3 and the initial data label is FMP4.

15. Enter the following information to define additional standard TXDS tasks (OCP and CONTROL should not be included):

```
TASK ID # - >B                    * file management
  PRIORITY LEVEL - 1
  INITIAL STATE -
  INITIAL DATA LABEL - FUR
TASK ID# -> C                     * disk volume names
  PRIORITY LEVEL - 0              * may be deleted if volume
  INITIAL STATE -                 * name support is not neede
  INITIAL DATA LABEL - VOLUME
```

16. Enter the following information to define the control task:

```
TASK ID # - >11
  PRIORITY LEVEL - 1
  INITIAL STATE -
  INITIAL DATA LABEL - MPMAIN
```

17. Enter the following information to complete GENTX:

```
TASK ID # -                       * no more tasks
MULTIPLE DYNAMIC TASKS (Y OR N) - N
CONSOLE DEV NAME - ME             * required name
DEFAULT DISC DEV - DSC            * required name
DEFAULT PRINT DEV - LP
ASSIGN LUNO -                     * no preassigned lunos
# OF SPARE DEV LUNO BLKS -
# OF SPARE FILE LUNO BLKS - 8     * see note below
# OF FILE CONTROL BLOCKS - 8      * see note below
# OF DEFAULT BUFFERS -
# OF GENERAL BUFFERS -
```

NOTE

The number of file luno blocks and file control blocks relates to the number of files that will be open at any given time during the execution of a program. The Compiler requires a minimum of six files to be opened at any one time. If a listing file is required instead of a device, than one additional file is required. Then for each level of copy, an additional file must be open. Therefore the number 8 shown above allows a listing file and one level of copy files. If additional levels of copy files are required, this number should be increased appropiately, but remember that each additional file requires memory which is then unavailable for a program.

18. Insert a user diskette into the left-hand unit to create the TASKDF and TXDATA source files.

```
  TASKDF OUTPUT FILE NAME - :TASKDF/SRC
  TXDATA OUTPUT FILE NAME - :TXDATA/SRC

  END TX990 SYSGEN
```

19. Enter the following information to execute OBJMGR to select and copy modules to an input file for TXSLNK.

```
  PROGRAM:  :OBJMGR/*

   OBJMGR          2.3.0  78.244 OBJECT MANAGER

      OUTPUT FILE: :TXPART/OBJ
```

20. Remove the TX990 Parts Diskette from the right-hand unit and insert the Microprocessor Pascal TX990 Sysgen Diskette. Then select the object modules from file DSC2:MPLIB/OBJ. Enter C for modules to be included and S for modules to be skipped. The system that is shown is for a 911 VDT. If a 913 VDT is to be used, DSR911 should be skipped and DSR913 should be included. The system which is shown does include floating point support. If floating point is not desired, then NOREALS should be included and REALS should be skipped.

```
         INPUT FILE:  DSC2:MPLIB/OBJ
        REWIND INPUT FILE? Y

  DSR911   ? C           * 911 VDT DSR module
  IOCOM    ? C           * VDT io module
  NOREALS  ? S           * no floating point support
  REALS    ? C           * floating point support
  MAIN     ? A           * include rest of file
```

21. Remove the Microprocessor Pascal TX990 Sysgen Diskette from the right-hand unit and place the TX990 Parts Diskette in again. Select object modules from file DSC2:DSRLIB/OBJ. Enter C for supported devices and S for devices that are not supported. Selections must be consistent with devices defined in steps 10 and 11. Selections shown are for a system that has a line printer and two floppy disk units.

```
         INPUT FILE:  DSC2:DSRLIB/OBJ
        REWIND INPUT FILE? Y

  FPYDSR   ? C           * floppy disc drive
```

```
   DSR911    ? S          * use AMPL 911 VDT dsr
   DSR913    ? S
   DSR733    ? S          * 733 asr
   LPDSR     ? C          * line printer dsr
   FLPDSR    ? S          * fast line printer dsr
   DSRTTY    ? D          * no more required
```

22. Select modules from file DSC2:FMPLIB/OBJ for floppy disk drives.

```
          INPUT FILE:  DSC2:FMPLIB/OBJ
         REWIND INPUT FILE? Y

   VOLUME    ? C          * delete if volume names not wanted
   TXFMP1    ? C
   TXFMP2    ? C
   TXFMP3    ? S          * include for third disk drive
   TXFMP4    ? S          * include for fourth disk drive
   TXFMP     ? A          * include remainder of file
   END-OF-FILE
```

23. Select modules from file DSC2:TXLIB/OBJ. The IMGLDR and TSKLDR modules should not be included because Microprocessor Pascal System uses its own loader routines. The STA911 and SVC911 modules must not be included. The EVENTK module may be excluded if the feature allowing the abortion of a program via the CMD key is not desired. The STASK must not be included because a different version is supplied.

```
          INPUT FILE:  DSC2:TXLIB/OBJ
         REWIND INPUT FILE? Y

   TXROOT    ? C
   TITTCM    ? S
   EVENTK    ? C          * may be excluded
   IMGLDR    ? S          * do not include
   CRTPRO    ? S          * do not include
   STA913    ? S          * do not include
   SVC913    ? S          * do not include
   STA911    ? S          * do not include
   SVC911    ? S
   IOSUPR    ? C
   TSKFUN    ? C
   TSKLDR    ? S          * do not include
   CNVRSN    ? C
   MEMSVC    ? S
   TBUFMG    ? C
   DTASK     ? S
   TXSTRT    ? C
   TXEND     ? C
   STASK     ? S          * do not include
   END-OF-FILE
```

24. Respond to next request for an input file with an asterisk (*), terminating OBJMGR, as follows:

```
      INPUT FILE: *
END OBJECT MANAGER
```

25. Remove the TX990 parts diskette from the right-hand disk unit, and place the TXDS system diskette in the right-hand unit.

26. Enter the following information to execute TXMIRA to assemble modules TXDATA and TASKDF.

```
PROGRAM: :TXMIRA/SYS
  INPUT: :TXDATA/SRC
 OUTPUT: :TXDATA/OBJ,:TXDATA/LST
OPTIONS: CLS

PROGRAM: :TXMIRA/SYS
  INPUT: :TASKDF/SRC
 OUTPUT: :TASKDF/OBJ,:TASKDF/LST
OPTIONS: CLS
```

27. Enter the following information to execute TXSLNK to link the new system. The link edit control file must be entered from the LOG and is shown below:

```
PROGRAM: :TXSLNK/SYS
  INPUT: LOG
 OUTPUT: :NEWSYS/SYS,:NEWSYS/LST
OPTIONS: M10000

TXSLNK   2.3.0  78.244  LINK EDITOR
FORMAT COMPRESSED
PHASE 0,SYSGEN
INCLUDE :TXDATA/OBJ
INCLUDE :TASKDF/OBJ
INCLUDE :TXPART/OBJ
END
```

28. Enter the following information to execute SYSUTL to copy the system loader to the new system diskette and to designate the newly linked system as the current system.

```
PROGRAM: :SYSUTL/SYS*

SYSUTL   2.3.0 78.244   SYSTEM UTILITY

OP: BC,DSC.SF,:NEWSYS/SYS.TE.
```

29. Remove the TXDS system diskette from the right-hand unit. The newly linked system may then be loaded by repeating steps 2, 3, and 4. The system will then display the prompts as shown in Section XVI.

APPENDIX L

MPP 733 ASR DSR Documentation

L.1 Implementation of a TI 733 ASR Handler

This example illustrates the complete implementation of a device handler for the TI 733 ASR data terminal. It is a very thorough example containing a high degree of technical detail. Therefore, several modules from within the handler will be presented individually with some detail removed. At the end of this example, the entire implementation is presented as a complete Microprocessor Pascal System-compatible module with all technical detail included in Figure 13.

L.2 User Interface and Operation of 733 ASR

The TI 733 ASR data terminal is an example of a physical device which contains four logical devices: keyboard, printer, left cassette, and right cassette. This handler allows for any subset of these logical devices to be operated through the user's application and can therefore be used with many data terminals besides the TI 733 ASR.

The user application communicates to the 733 ASR through text files or sequential files having a component lenghth of 80 bytes (characters) or less. The keyboard can be connected to the user application for reading by performing a RESET on the file having the same name as the keyboard. As characters are entered at the keyboard they are echoed to the printer and may be edited via use of the backspace control character. A carrage return indicates an end of line and a control S (DC3) indicates an end of file. The printer is connected for writing by performing a REWRITE on the file having the same name as the printer. Printable characters are printed with a form feed (FF) being output as eight line feeds. Both cassettes may be opened for either reading or writing by performing a RESET or REWRITE on the file having the same name as the cassette drive. A read operation will read from the cassette until the read is satisfied or until an end of file is encountered. Write operations write lines of text files as records to the cassette in line format with a close operation writing an end of file. All cassette operations assume that the cassette has been manually located to the proper tape position to allow multiple files per cassette. Cassette operations also assume that the 733 ASR contains hardware options compatable with the TX operating system, and capable of disabling the keyboard and printer. The keyboard and printer can be connected to multiple files simultaneously while the cassettes may be connected to only one file at a time.

Due to the asynchronous properties of processor management, instructions which are successive in code space are not neccessarily executed by the processor in succession without the unexpected intervention of other instructions. The result is that the user must keep cognizant of the desired execution suquence of the code and ensure, through programming techniques, that a neccessary input/output is allowed to complete before a successive input/output is started.

An example driver is shown below. The function of this code is to prompt the user to enter information from the keyboard, and enter that information on cassette one. initial statements of this driver start the system clock routine (CLKINT), and name the specific 733 ASR logical devices to be utilized. In lines ten and eleven of the code a RESET, and CLOSE of PR01 is performend to ensure that I/O of the WRITELN function is completed before the system begins reading the keyboard, and writing to the cassette.

```
BEGIN (* HANDLER_TST *)
START CLKINT
ASR( #80, #4D0, 4, 'KB01     ', 'PR01     ', 'CAS01    ',
   '         ', TRUE );
REWRITE(PR01);
WRITELN(PR01,'TYPE YOUR MESSAGE, THEN <CONTROL S> WHEN DONE.');
WRITELN(PR01,'THE INFORMATION WILL THEN BE COPIED ONTO CASSETTE'):
WRITELN(PR01,'ONE FOR FUTURE REFERENCE.');
CLOSE(PR01);
RESET(PR01);
CLOSE(PR01);
RESET(KB01);
REWRITE(CAS01);
REPEAT
   READLN(KB01,MSG);
   WRITELN(CAS01,MSG);
UNTIL EOF(KB01);
CLOSE(KB01);
CLOSE(CAS01);
REWRITE(PR01);
WRITELN(PR01,'INFORMATION COPIED FROM KEYBOARD TO CASSETTE ONE');
CLOSE(PR01);
END(* HANDLER_TST *)
```

The handler is implemented by an interface process for each logical device as shown in Figure 1. An interrupt demultiplexer is necessary to signal the proper interface process when an interrupt occurs from the TMS 9902 Asynchronous Communicatins Controller. For more information about this serial interface refer to the "TMS 9902 Asynchronous Communications Controller Data Manual", MP004.

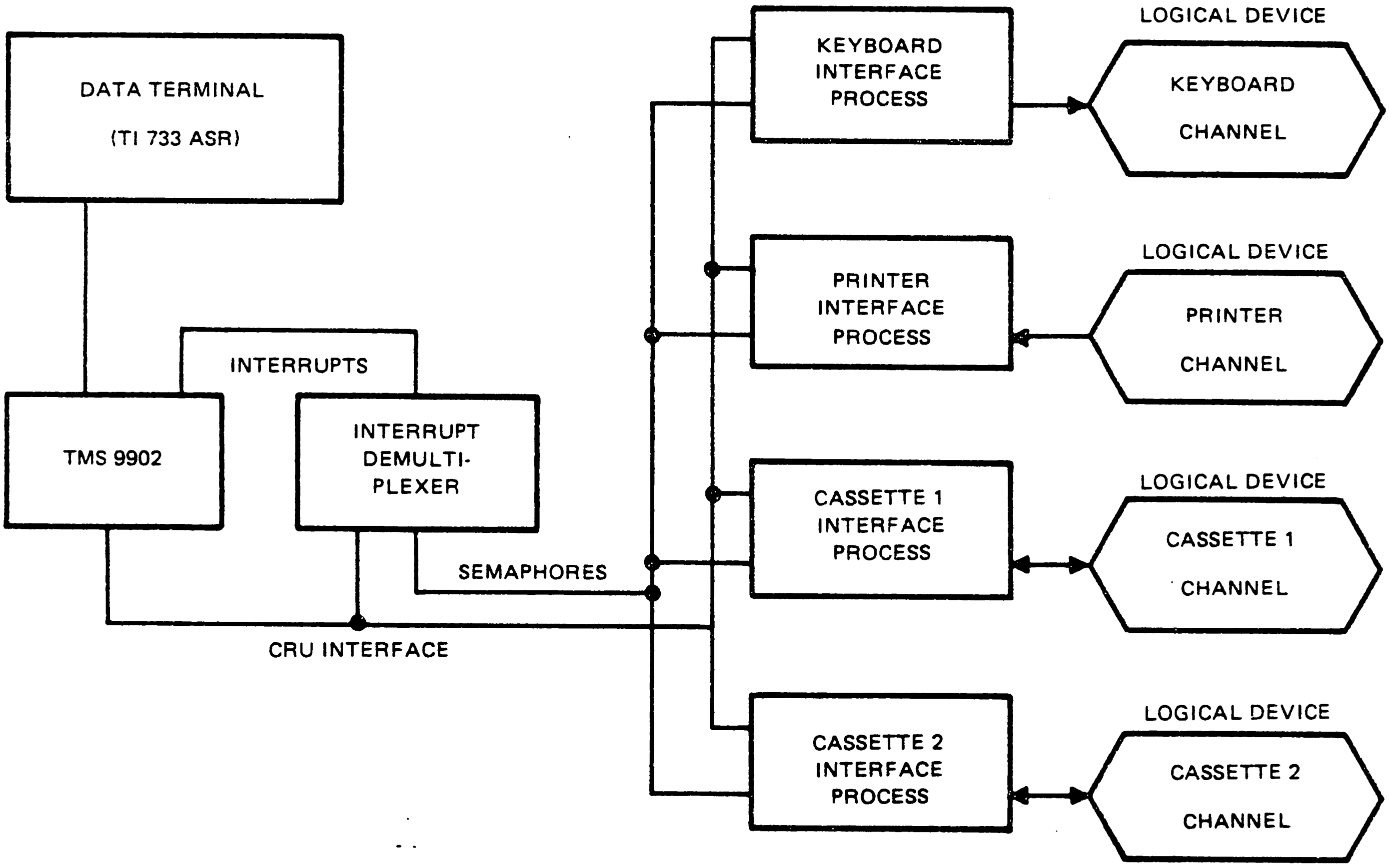


Figure 1. 733 ASR Handler Implementation

The 733 ASR physical device interface system is initialized by calling the procedure ASR$ which has the following calling sequence:

```
TYPE
  CRU_ADDRESS = 0..#1FFE;
  INTERRUPT_LEVEL = 0..15;
  ALPHA = PACKED ARRAY   1..8   OF CHAR;

PROCEDURE ASR$(
  BASE: CRU_ADDRESS;
  BAUD: INTEGER;
  LEVEL: INTERRUPT_LEVEL;
  KEYBOARD_NAME: ALPHA;
  PRINTER_NAME: ALPHA;
  CASSETTE_ONE_NAME: ALPHA;
  CASSETTE_TWO_NAME: ALPHA;
  FILE_ORIENTED: BOOLEAN);
  EXTERNAL;
```

A logical device can be excluded from the interface system by passing a device name consisting of eight spaces.

L.3 Implementation of Initialization Procedure

Figure 2 illustrates the implementation of the procedure ASR$ which is invoked from the user's application to initialize the device handler system.

```
PROCEDURE ASR$(
  BASE: CRU_ADDRESS;
  BAUD: INTEGER;
  LEVEL:  INTERRUPT_LEVEL;
  KEYBOARD_NAME: ALPHA;
  PRINTER_NAME: ALPHA;
  CASSETTE_ONE_NAME:  ALPHA;
  CASSETTE_TWO_NAME:  ALPHA;
  FILE_ORIENTED:  BOOLEAN);
VAR
  INITIALIZATION_COMPLETE:  SEMAPHORE;
BEGIN (* ASR$ *)
INITSEMAPHORE( INITIALIZATION_COMPLETE,0 );
START ASR$SUPERVISOR( BASE, BAUD, LEVEL, KEYBOARD_NAME,
  PRINTER_NAME, CASSETTE_ONE_NAME, CASSETTE_TWO_NAME,
  FILE_ORIENTED, INITIALIZATION_COMPLETE );
WAIT( INITIALIZATION_COMPLETE );
TERMSEMAPHORE( INITIALIZATION_COMPLETE );
END (* ASR$ *);
```

Figure 2. ASR Interface System Initialization Procedure

L.4 Implementation of Supervisor Program

The ASR physical device interface system supervisor program is responsible for initializing the device, starting the interface processes, and informing the initialization procedure that the initialization is complete. The supervisor program then becomes the interrupt demultiplexer for the interface processes. Figure 3 illustrates the implementation of the ASR supervisor program.

```
PROGRAM ASR$SUPERVISOR(
  BASE: CRU_ADDRESS;
  BAUD: INTEGER;
  ASRSUPERVISOR_LEVEL:  INTERRUPT_LEVEL;
  KEYBOARD_NAME: ALPHA;
  PRINTER_NAME: ALPHA;
  CASSETTE_ONE_NAME:  ALPHA;
  CASSETTE_TWO_NAME:  ALPHA;
  FILE_ORIENTED:  BOOLEAN;
  INITIALIZATION_COMPLETE: SEMAPHORE );
VAR
  PARTIAL_COMPLETION: SEMAPHORE;
  EXCLUSIVE_ACCESS: SEMAPHORE;
  ACKNOWLEDGE: SEMAPHORE;
  READ_READY: SEMAPHORE;
  WRITE_READY:  SEMAPHORE;
  DSR_READY: SEMAPHORE;
  INTERRUPT:  SEMAPHORE;
  CASSETTE_MODE: CHAR;
  STATUS: CASSETTE_STATUS;
  WANTED: BOOLEAN;

PROCEDURE INPUT9CHAR( VAR CH: CHAR);
  ...
END (* INPUT9CHAR *);


PROCEDURE OUTPUT9CHAR( CH: CHAR; DEVICE: DEVICE_TYPE );
  ...
END  (* OUTPUT9CHAR *);



PROCEDURE SEND9CONTROL( CONTROL_CHAR: CHAR );
  ...
END (* SEND CONTROL *);



PROCEDURE SET9MODE( NEW_MODE: MODE; CASSETTE: CASSETTE_SIDE );
  ...
END  (* SET9MODE *);
```

```
PROCESS KBDINTERFACE(
  INFILE: TEXT );
  ...
END  (* KBDINTERFACE *);



PROCESS PRTINTERFACE(
  OUTFILE: TEXT );
  ...
END  (* PRTINTERFACE *);



PROCESS CASINTERFACE(
  ...
END  (* CASINTERFACE *);



BEGIN (* ASRSUPERVISOR *)
(*# PRIORITY = ASRSUPERVISOR_LEVEL;
    STACKSIZE = ASRSUPERVISOR_STACK;
    HEAPSIZE = ASRSUPERVISOR_HEAP *)
CRUBASE( TMS9901 )  (* SET CRU BASE TO 9901 *);
SBZ( CONTROL_BIT )  (* SET TO INTERRUPT MODE *);
CRUBASE( TMS9901 +( 2* ASRSUPERVISOR_LEVEL ) );
LDCR( 1, 1 )       (* ENABLE INTERRUPT MASK *);
CRUBASE( BASE )      (* SET CRU BASE TO 9902*);
SBO( DEVICE_RESET )         (* RESET 9902 *);
LDCR( 8, CONTROL_REG )      (* LOAD CONTROL REGISTER *);
SBZ( INT_REG_LOAD_FLAG )    (* SKIP INTERVAL REGISTER LOAD *);
LDCR( 12, BAUD)       (* LOAD TRANS/RECEV DATA RATE REGISTER *);
CASSETTE_MODE := 'Z';
INITSEMAPHORE( PARTIAL_COMPLETION, 0 );
INITSEMAPHORE( EXCLUSIVE_ACCESS, 1 );
INITSEMAPHORE( ACKNOWLEDGE, 0 );
INITSEMAPHORE( READ_READY, 0 );
INITSEMAPHORE( WRITE_READY, 0 );
INITSEMAPHORE( DSR_READY, 0 );
INITSEMAPHORE( INTERRUPT, 0 );
EXTERNALEVENT( INTERRUPT, ASRSUPERVISOR_LEVEL );
IF KEYBOARD_NAME <> NULL THEN BEGIN
  START KBDINTERFACE( FILENAMED( KEYBOARD_NAME ) );
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF PRINTER_NAME <> NULL THEN BEGIN
  START PRTINTERFACE( FILENAMED( PRINTER_NAME ) );
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF CASSETTE_ONE_NAME <> NULL THEN BEGIN
  START CASINTERFACE( FILENAMED( CASSETTE_ONE_NAME ), LEFT_CASSETTE )
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF CASSETTE_TWO_NAME <> NULL THEN BEGIN
  START CASINTERFACE( FILENAMED( CASSETTE_TWO_NAME ), RIGHT_CASSETTE
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
SIGNAL( INITIALIZATION_COMPLETE );
TERMSEMAPHORE( PARTIAL_COMPLETION );
WHILE TRUE DO BEGIN
  WAIT( INTERRUPT );
  IF TB( TRANSMIT_INTERRUPT ) THEN BEGIN
    SBZ(TRANSMIT_INTERRUPT_ENABLE);
    SIGNAL( WRITE_READY );
    WAIT( ACKNOWLEDGE );
    END (* IF TB( TRANSMIT_INTERRUPT ) *)
    ELSE IF TB( RECEIVE_INTERRUPT ) THEN BEGIN
      SBZ( RECEIVER_INTERRUPT_ENABLE );
      SIGNAL( READ_READY );
      WAIT( ACKNOWLEDGE );
      END  (* IF TB( RECEIVE_INTERRUPT ) *)
      ELSE IF TB( STATUS_CHANGE_INTERRUPT ) THEN BEGIN
        REPEAT CSIGNAL( DSR_READY, WANTED )
          UNTIL NOT WANTED;
        SBZ( STATUS_CHANGE_INTERRUPT_ENABLE );
        END  (* IF TB( STATUS_CHANGE_INTERRUPT ) *)
        ELSE EXCEPTION( USER_ERROR, 1 );
  END  (* WHILE TRUE DO *);
END  (* ASRSUPERVISOR *);
```

Figure 3. ASR Interface System Supervisor Program

The procedures INPUT_CHAR and OUTPUT_CHAR do the device dependent input and output to the 733 ASR. The procedure SEND_CONTROL is used to output remote control charaters to the ASR while the procedure SET_MODE is used to set the proper cassette mode for recording or playback of cassettes. The concurrent characteristics for this program specify that its priority is to be the same as the interrupt level of the device being serviced. The constants ASR$SUPERVISOR_LEVEL and ASR$SUPERVISOR_HEAP are constants defined at the system level. The program first initializes the TMS 9901 to allow interrupts to pass from the TMS 9902 at the proper level. The supervisor program then initializes the TMS 9902 to the proper baud rate and character format. The cassette mode is initialized to a non-active state. The semaphores used by the interrupt demultiplexer are then initialized to the proper initial values. The PARTIAL_COMPLETION semaphore is used to await partial system initializations. The EXCLUSIVE_ACCESS semaphore is used to allow only one interface process access to the physical device at a time. The semaphore INTERRUPT is initialized to correspond to the external interrupt from the physical device. The supervisor program next starts the necessary interface processes for the logical devices requested. Once these have been started, the supervisor program becomes the interrupt demultiplexer. When an interrupt occurs, the demultiplexer polls the device to acertain which type of interrupt occured and signals the proper waiting process. If an invalid interrupt has occured, a user exception is caused with a reason code of 1.

L.5 Implementation of ASR Keyboard Logical Device Process

The ASR keyboard logical device process has the responsibility for reading, editing and echoing ASCII text from the ASR keyboard to the file INFILE via the CRU interface. Figure 4 illustrates the implementation of this process.

```
PROCESS KBD$INTERFACE(
  INFILE: TEXT );

PROCEDURE KBD$WORK;
 ...
END;

PROCEDURE KBD$EXCEPTION;
 ...
END;

BEGIN  (* KBD$INTERFACE *)
(*# PRIORITY = KEYBOARD_LEVEL;
    STACKSIZE = KEYBOARD_STACK;
    HEAPSIZE = KEYBOARD_HEAP *)
F$MASTER( INFILE );
IF FILE_ORIENTED THEN F$XACCESS( INFILE );
F$STLENGTH( INFILE, 80 );
F$ULENGTH( INFILE );
F$STMODE( INFILE, WRITING );
F$STEOC( INFILE );
F$CREATECHANNEL( INFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( KBD$EXCEPTION ) );
KBD$WORK;
END  (* KBD$INTERFACE *);
```

Figure 4. ASR Keyboard Logical Device Process

The nested procedure KBD$WORK does all the necessary work after the data structures have been initialized. The nested procedure KBD$EXCEPTION is the exception handler for this process. The channel characteristics of the keyboard device channel are initialized through the file INFILE. INFILE is established as channel master by the routine F$MASTER. If the keyboard is to be file oriented (FILE_ORIENTED=TRUE), the routine F$XACCESS is called to set the maximum number of users to one. The routine F$STLENGTH is called to set the maximum line length to 80 characters and F$ULENGTH is called to allow user files of a shorter length. The mode of the device channel is set to writing by a call to F$STMODE. The routine F$STEOC is called to detect when end of consumption occurs on the device channel. F$CREATECHANNEL is then called to create the channel. The semaphore PARTIAL_COMPLETION is then signaled to indicate to the supervisor program that the initialization of the keyboard channel is complete. The procedure ONEXCEPTION is called to establish KBD$EXCEPTION as the exception handler and the work procedure KBD$WORK is entered. The implementation of the KBD$WORK procedure is illustrated in Figure 5.

```
PROCEDURE KBDWORK;
VAR
  CH: CHAR;
BEGIN  (* KBDWORK *)
WHILE TRUE DO BEGIN
  REWRITE( INFILE );
  WHILE NOT F$EOC( INFILE ) DO BEGIN
    WAIT( EXCLUSIVE_ACCESS );
    INPUT9CHAR( CH );
    WHILE CH <> DC3 AND CH <> CR DO BEGIN
      IF ( CH = BS OR EOLN( INFILE ) ) AND COLUMN( INFILE ) > 0 THEN
        BEGIN
        OUTPUT9CHAR( BS, PRINT );
        F$BSPACE( INFILE );
        END  (* IF BS OR EOLN AND COLUMN > 0 *);
      IF CH = HT THEN BEGIN
       WRITE( INFILE, HT );
       OUTPUT9CHAR( SPACE, PRINT );
       END  (* IF CH = HT *)
       ELSE IF CH = LF THEN
         OUTPUT9CHAR( LF, PRINT )
         ELSE IF CH >= SPACE THEN BEGIN
           OUTPUT9CHAR( CH, PRINT );
           WRITE( INFILE, CH);
           END  (* IF CH >= SPACE *);
    INPUT9CHAR( CH );
    END  (* WHILE CH<> DC3 AND CH <> CR *);
    OUTPUT9CHAR( CR, PRINT );
    OUTPUT9CHAR( LF, PRINT );
    SIGNAL( EXCLUSIVE_ACCESS );
    IF CH = CR THEN WRITELN( INFILE )
      ELSE IF CH = DC3 THEN REWRITE( INFILE )
    END  (* WHILE NOT F$EOC( INFILE ) *);
  END  (* WHILE TRUE DO *);
END  (* KBDWORK *);
```

Figure 5. Implementation of Keyboard Work Procedure

The code within the outer-most WHILE loop copies a single file sequence from the keyboard to the file INFILE. The REWRITE opens the file INFILE for writing and waits for the user file to connect to the channel. The code within the WHILE NOT F$EOC(INFILE) loop copies characters from the keyboard until the user disconnects from the channel. Once exclusive access has been obtained, a WHILE loop is entered to transfer characters until an end of line or end of file occurs. A backspace character causes the printer and INFILE to be backed up one character. If at the end of a line in the user's file, the print carrage and user file are backspaced. A tab character (HT) is placed in the user file and echoed as a space to the printer. A line feed (LF) is echoed but not placed into the user file. Otherwise, the character is echoed to the printer and placed into the user file if it is a valid non-control character. When an end of line or end of file is entered a carrage return and line feed are echoed to the printer. An end of line causes a WRITELN to the file while an end of file causes a REWRITE.

Aborting the logical device channel for the keyboard causes all file variables connected to the channel to become disconnected. Any subsequent WRITE of a disconnected user file variable will result in an exception until the file is re-opened. Any files suspended on the channel being aborted are activated with an exception. This type of exception will occur for the process KBD$INTERFACE if INFILE is ever aborted since INFILE is the channel master.

Any exception of KBD$INTERFACE causes an implicit escape from KBD$WORK and an implicit invocation of KBD$EXCEPTION. The implementation of KBD$EXCEPTION is illustrated in Figure 6.

```
PROCEDURE KBDEXCEPTION;
BEGIN  (* KBDEXCEPTION *)
IF ERR$CLASS = FCLASS_FILE_ERROR THEN BEGIN
  ERR$CLEAR;
  KBDWORK;
  END  (* IF ERR$CLASS = FCLASS_FILE_ERROR *);
END  (* KBDEXCEPTION *);
```

Figure 6. Implementation of Keyboard Exception Handler

The only exception handled by this routine is one of class F$$CLASS_FILE_ERROR (file error). All other exceptions are ignored and cause a termination of KBD$INTERFACE with the exception still present. If the reason for the exception is a channel abort, the exception is cleared and KBD$WORK is invoked to restart the transfer. Since the keyboard device channel has a master, it is not destroyed when aborted. Therefore, it is not necessary to initialize the channel again. The REWRITE in KBD$WORK will first close INFILE and will then connect INFILE to the channel again. At this time, user files are allowed to connect to the channel.

L.6 Implementation of ASR Printer Logical Device Process

The ASR printer logical device process has the responsibility of transfering ASCII text from the file OUTFILE to the ASR printer via the CRU interface. Figure 7 illustrates the implementation of this process.

```
PROCESS PRT$INTERFACE(
  OUTFILE: TEXT );

PROCEDURE PRT$WORK;
 ...
END;

PROCEDURE PRT$EXCEPTION;
 ...
END;

BEGIN  (* PRT$INTERFACE *)
(*# PRIORITY = PRINTER_LEVEL;
    STACKSIZE = PRINTER_STACK;
    HEAPSIZE = PRINTER_HEAP *)
F$MASTER( OUTFILE );
IF FILE_ORIENTED THEN F$XACCESS( OUTFILE );
F$STLENGTH( OUTFILE, 80 );
F$ULENGTH( OUTFILE );
F$STMODE( OUTFILE, READING );
F$CREATECHANNEL( OUTFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( PRT$EXCEPTION ) );
PRT$WORK;
END  (* PRT$INTERFACE *);
```

Figure 7. ASR Printer Logical Device Process

The nested procedure PRT$WORK does the majority of the work after the necessary structures are initialized. The nested procedure PRT$EXCEPTION is the exception handler for the process. The channel characteristics of the printer device channel are initialized through the file OUTFILE. The initialization of this channel is similar to the initialization of the keyboard channel with differences in the channel mode and not requesting end of consumption flag detection. The printer channel mode is READING instead of WRITING as is the keyboard channel. After the channel is created, the semaphore PARTIAL_COMPLETION is signaled to indicate to the supervisor program that initialization of the device channel is complete. The procedure PRT$EXCEPTION is established as the exception handler and the procedure PRT$WORK is invoked to perform the transfer of information from the file OUTFILE to the printer. The implementation of PRT$WORK is illustrated in Figure 8.

```
PROCEDURE PRTWORK;
VAR
  CH: CHAR;
BEGIN  (* PRTWORK *)
WHILE TRUE DO BEGIN
  RESET( OUTFILE );
  WHILE NOT EOF( OUTFILE ) DO BEGIN
    WAIT( EXCLUSIVE_ACCESS );
    WHILE NOT EOLN( OUTFILE ) DO BEGIN
      READ( OUTFILE, CH );
      IF CH = HT THEN
        OUTPUT9CHAR( SPACE, PRINT )
        ELSE IF CH = FF THEN
          FOR I := 1 TO 8 DO
            OUTPUT9CHAR( LF, PRINT )
          ELSE IF CH = BELL OR
                  CH = BS OR
                  CH = LF OR
                  CH = CR OR
                  CH >= SPACE THEN
            OUTPUT9CHAR( CH, PRINT );
      END  (* WHILE NOT EOLN *);
    IF CH >= SPACE THEN BEGIN
      OUTPUT9CHAR( CR, PRINT);
      OUTPUT9CHAR( LF, PRINT);
      END  (* IF CH >= SPACE *);
    SIGNAL( EXCLUSIVE_ACCESS );
    READLN( OUTFILE );
    END  (* WHILE NOT EOF *);
  END  (* WHILE TRUE DO *);
END  (* PRTWORK *);
```

Figure 8. Implementation of ASR Printer Work Procedure

The code within the outer_most WHILE loop outputs a single file to the printer. The RESET opens the file OUTFILE for reading and connects it to the printer device channel. The code within the WHILE NOT EOF(OUTFILE) loop copies an entire line to the printer. Exclusive access to the physical device is gained by waiting on the semaphore EXCLUSIVE_ACCESS. Once this exclusive access is obtained, the WHILE NOT EOLN(OUTFILE) loop is entered to transfer characters to the printer. A tab character (HT) is ouput as a space and a form feed (FF) is ouput as eight line feeds. All other valid non-control characters are copied to the printer until the entire line has been output. If the last character is not a control character, then a carrage return and line feed are output. Then exclusive access is released by signaling the semaphore EXCLUSIVE_ACCESS until the next line is ready for output.

The exception handler for the printer process is very similar to the exception handler for the keyboard process. When a channel abortion occurs, it clears the exception and invokes the PRT$WORK procedure. For other exceptions, no corrective action is taken.

L.7 Implementation of ASR Cassette Logical Device Process

The ASR cassette logical device process has the responsibility for reading and writing ASCII text to and from a cassette drive of the ASR data terminal via the CRU interface. Figure 9 illustrates the implementation of this process.

```
PROCESS CAS$INTERFACE(
  IOFILE: TEXT;
  CASSETTE: CASSETTE_SIDE );
VAR
  USERS_MODE: CHANNEL_MODE;

PROCEDURE CAS$WORK;
 ...
END;

PROCEDURE CAS$EXCEPTION;
 ...
END;

BEGIN  (* CAS$INTERFACE *)
(*# PRIORITY = CASSETTE_LEVEL;
    STACKSIZE = CASSETTE_STACK;
    HEAPSIZE = CASSETTE_HEAP *)
F$MASTER( IOFILE );
F$XACCESS( IOFILE );
F$STLENGTH( IOFILE, 80 );
F$ULENGTH( IOFILE );
F$STMODE( IOFILE, USERMODE );
F$STEOC( IOFILE );
F$CREATECHANNEL( IOFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( CAS$EXCEPTION ) );
CAS$WORK;
END  (* CAS$INTERFACE *);
```

Figure 9. Implementation of ASR Cassette Logical Device Process

The nested procedure CAS$WORK does the majority of the work after the necessary structures are initialized. The nested procedure CAS$EXCEPTION is the exception handler for this process. The channel characteristics of the cassette device channels are initialized through the file IOFILE. The initialization of this channel is similar to the initialization of the keyboard and printer channels with some slight differences. Because the cassette channels are always assumed to be file oriented, the maximum number of users is always set to one by a call to the routine F$XACCESS. The channel mode is set to user mode with a call to F$STMODE. The routine F$STEOC is also called to detect end of consumption when reading from the cassette. After the channel is created, the semaphore PARTIAL_COMPLETION is signaled to indicate to the supervisor program that the initialization of the device channel is complete. The procedure CAS$EXCEPTION is established as the exception handler and the procedure CAS$WORK is invoked to perform the necessary transfers between IOFILE and the ASR cassette drive. The implementation of CAS$WORK is illustrated in Figure 10.

```
PROCEDURE CAS$WORK;

PROCEDURE CAS$_OUT_WORK;
  ...
END;

PROCEDURE CAS$_IN_WORK;
  ...
END;

BEGIN  (* CAS$WORK *)
WHILE TRUE DO BEGIN
  F$WAIT( IOFILE, USERS_MODE );
  CASE USERS_MODE OF
    READING:  CAS$_IN_WORK;
    WRITING:  CAS$_OUT_WORK
    END  (* CASE *);
  END  (* WHILE TRUE DO *);
END  (* CAS$WORK *);
```

Figure 10. Implementation of ASR Cassette Work Procedure

The code within the WHILE loop waits for a user file to connect to the cassette channel and then calls the appropriate transfer procedure for the file mode selected. The nested procedure CAS$_OUT_WORK transfers ASCII characters from the file IOFILE to the cassette drive while the nested procedure CAS$_IN_WORK transfers from the cassette drive to the file IOFILE. The implementation of the procedure CAS$_OUT_WORK is illustrated in Figure 11.

```
PROCEDURE CAS9OUT9WORK;
VAR
  CH: CHAR;
BEGIN  (* CAS9OUT9WORK *);
RESET( IOFILE );
WHILE NOT EOF( IOFILE ) DO BEGIN
  WAIT( EXCLUSIVE_ACCESS );
  SET9MODE( RECORDING, CASSETTE );
  IF NOT STATUS.RECORD_READY THEN
    EXCEPTION( USER_ERROR, 0 );
  OUTPUT9CHAR( DC2, PRINT );
  WHILE NOT EOLN( IOFILE ) DO BEGIN
    READ( IOFILE, CH );
    IF CH = CR THEN OUTPUT9CHAR( ETB, TAPE )
    ELSE IF CH = HT OR
            CH = FF OR
            CH = BELL OR
            CH = BS OR
            CH >= SPACE THEN OUTPUT9CHAR( CH, TAPE );
    END  (* WHILE NOT EOLN *);
  OUTPUT9CHAR( CR, TAPE );
  OUTPUT9CHAR( LF, TAPE );
  OUTPUT9CHAR( DC4, TAPE );
  OUTPUT9CHAR( DEL, PRINT );
  SIGNAL( EXCLUSIVE_ACCESS );
  READLN( IOFILE );
  END  (* WHILE NOT EOF *);
WAIT( EXCLUSIVE_ACCESS );
SET9MODE( RECORDING, CASSETTE );
IF NOT STATUS.RECORD_READY THEN
  EXCEPTION( USER_ERROR, 0 );
OUTPUT9CHAR( DC2, PRINT);
OUTPUT9CHAR( DC3, TAPE );
OUTPUT9CHAR( CR, TAPE );
OUTPUT9CHAR( DC4, TAPE );
OUTPUT9CHAR( DEL, PRINT );
SIGNAL( EXCLUSIVE_ACCESS );
CLOSE( IOFILE )
END  (* CAS9OUT9WORK *);
```

Figure 11. Implementation of ASR Cassette Output Procedure

The RESET opens the file IOFILE for reading. The user file has already connected to the channel inside the procedure CAS$WORK during the call to F$WAIT. The code within the WHILE NOT EOF(IOFILE) loop transfers the entire user file to the cassette drive. After waiting for exclusive access to the physical device, the ASR is set into the record cassette mode for the proper cassette drive. The status is checked and a user exception with a reason code of

0 is generated if the ASR did not change into the proper mode. The cassette drive is then started by outputting a DC2 control character and the WHILE NOT EOLN(IOFILE) loop is entered to transfer the individual characters of the cassette record. Carrage return characters are output as ETB characters and all other valid non-control characters are transfered directly from the cassette channel to the cassette drive. The cassette record is terminated by a CR, LF, DC4, DEL character sequence. When the entire file has been transferred, another record containing a DC3, CR, DC4, DEL end of file character sequence is output to the cassette. Once the end of file mark has been written, the file IOFILE is closed and the procedure CAS$WORK is re-entered to wait for the next user file to connect. The implementation of the CAS$_IN_WORK procedure is illustrated in Figure 12.

```
PROCEDURE CAS9IN9WORK;
VAR
  CH: CHAR;
  END_OF_FILE: BOOLEAN;
BEGIN  (* CAS9IN9WORK *)
REWRITE( IOFILE );
END_OF_FILE := FALSE;
WHILE NOT F$EOC( IOFILE ) AND NOT END_OF_FILE DO BEGIN
  WAIT( EXCLUSIVE_ACCESS );
  SET9MODE( PLAYBACK, CASSETTE );
  IF NOT STATUS.PLAY_READY THEN
    EXCEPTION( USER_ERROR, 0 );
  SEND9CONTROL( BLOCK_FORWARD );
  REPEAT
    INPUT9CHAR( CH );
    UNTIL CH <> LF OR CH <> DEL;
  IF CH <> DC3 THEN BEGIN
    WHILE CH <> CR AND NOT EOLN( IOFILE ) DO BEGIN
      IF CH = ETB THEN WRITE( IOFILE, CR )
      ELSE IF CH = HT OR
              CH = FF OR
              CH = BELL OR
              CH = BS OR
              CH >= SPACE THEN WRITE( IOFILE, CH )
      ELSE IF CH = DC3 THEN SEND9CONTROL( BLOCK_FORWARD );
      INPUT9CHAR( CH );
      END  (* WHILE CH <> CR AND NOT EOLN *);
    END  (* IF CH <> DC3 *)
    ELSE END_OF_FILE := TRUE;
  SIGNAL( EXCLUSIVE_ACCESS );
  IF NOT END_OF_FILE THEN WRITELN( IOFILE );
  END  (* WHILE NOT F$EOC OR END_OF-FILE *);
CLOSE( IOFILE );
END  (* CAS9IN9WORK *);
```

Figure 12. Implementation of ASR Cassette Input Procedure

The REWRITE opens the file IOFILE for writing. The user file has already connected to the channel during the call to F$WAIT within the procedure CAS$WORK. END_OF_FILE is initialized to false and a WHILE loop is entered to transfer cassette records from the cassette drive to the cassette channel until no more records are requested or until the cassette encounters an end of file mark. After exclusive access is gained for the physical device, the proper cassette is placed into the playback mode. If the ASR status does not indicate that the cassette is available for playback, then a user exception with a reason code of 0 is caused. The cassette drive is started by sending a block forward control sequence and a REPEAT loop is entered to read any beginning line feed (LF) or delete (DEL) characters. These line feed and delete characters are not transferred to the cassette channel. If the first actual character of the record is not a DC3 (indicating end of file), then characters are transferred until a carrage return is encountered or the user record is full. ETB characters are transferred as carrage returns (CR) to the cassette channel and DC3's are detected which cause a restart of the cassette drive. If an end of file mark has been encountered, END_OF_FILE is set to true. If not end of file, the record is transfered to the cassette channel by a WRITELN. The file IOFILE is closed when the transfer is complete and the CAS$WORK procedure is re-entered.

The exception handler for the cassette logical process is very similar to the exception handlers for the keyboard and printer processes. Channel abortions result in a re-start of the channel and no other exceptions are handled. The entire ASR physical device interface system is presented in Figure 13.

```
 SYSTEM ASRHANDLER;

   (* HIERARCHICAL STRUCTURE OF ASRHANDLER IS:

        ASRHANDLER  . . . . .  NULL-BODY SYSTEM ENVIRONMENT
          ASRSUPERVISOR. . . . PHYSICAL DEVICE SUPERVISOR PROGRAM
            INPUT9CHAR  . . . . DEVICE-DEPENDENT INPUT ROUTINE
            OUTPUT9CHAR . . . . DEVICE-DEPENDENT OUTPUT ROUTINE
            SEND9CONTROL. . . . DEVICE-DEPENDENT CONTROL ROUTINE
            SET9MODE . . . . .  DEVICE-DEPENDENT MODE ROUTINE
            KBDINTERFACE  . . . KEYBOARD LOGICAL PROCESS
              KBDWORK  . . . .  KEYBOARD WORK PROCEDURE
              KBDEXCEPTION. . . KEYBOARD EXCEPTION HANDLER
            PRTINTERFACE  . . . PRINTER LOGICAL PROCESS
              PRTWORK  . . . .  PRINTER WORK PROCEDURE
              PRTEXCEPTION. . . PRINTER EXCEPTION HANDLER
            CASINTERFACE  . . . CASSETTE LOGICAL PROCESS
              CASWORK  . . . .  CASSETTE WORK PROCEDURE
                CAS9OUT9WORK .  CASSETTE OUTPUT PROCEDURE
                CAS9IN9WORK  .  CASSETTE INPUT PROCEDURE
              CASEXCEPTION. . . CASSETTE EXCEPTION HANDLER
          ASR . . . . . . . .  PHYSICAL DEVIVE INITIALIZATION

 CONST
   ASRSUPERVISOR_STACK = 220;
   ASRSUPERVISOR_HEAP = 1100;
   KEYBOARD_LEVEL = 62;
   KEYBOARD_STACK = 140;
   KEYBOARD_HEAP = 000;
   PRINTER_LEVEL = 61;
```

```
  PRINTER_STACK = 200;
  PRINTER_HEAP = 000;
  CASSETTE_LEVEL = 60;
  CASSETTE_STACK = 170;
  CASSETTE_HEAP = 000;

  MAXINT = 32767;
  FCLASS_FILE_ERROR = 7;
  USER_ERROR = 1;
  NULL = '        ';

  TMS9901 = #100;
  CONTROL_BIT = 0;

  DATA_SET_READY = 27;
  RECEIVE_ERROR = 9;
  REQUEST_TO_SEND = 16;
  DEVICE_RESET = 31;
  INT_REG_LOAD_FLAG = 13;
  RECEIVER_INTERRUPT_ENABLE = 18;
  TRANSMIT_INTERRUPT_ENABLE = 19;
  STATUS_CHANGE_INTERRUPT_ENABLE = 21;
  TRANSMIT_INTERRUPT = 17;
  RECEIVE_INTERRUPT = 16;
  STATUS_CHANGE_INTERRUPT = 20;
  CONTROL_REG = #62;
  ASR_RATE = #1A0;

  BELL = '#07';
  BS = '#08';
  HT = '#09';
  LF = '#0A';
  VT = '#0B';
  FF = '#0C';
  CR = '#0D';
  DLE = '#10';
  DC2 = '#12';
  DC3 = '#13';
  DC4 = '#14';
  ETB = '#17';
  SPACE = '#20';
  DEL = '#7F';
  REQUEST_STATUS = '>';
  BLOCK_FORWARD = '7';

TYPE
  INTERRUPT_LEVEL = 0..15;
  ALPHA = PACKED ARRAY   1..8   OF CHAR;
  NONNEG = 0..MAXINT;
  CHANNEL_MODE = ( READING, WRITING, USERMODE );
  DEVICE_TYPE = ( PRINT, TAPE );
  CRU_ADDRESS = 0..#1FFE;
  CASSETTE_SIDE = ( LEFT_CASSETTE, RIGHT_CASSETTE );
  MODE = ( PLAYBACK, RECORDING );
  CASSETTE_STATUS = PACKED RECORD
          B1,B2,B3,B4,B5,B6,B7,B8: BOOLEAN;
          PARITY,PLAYBACK_OFF,PRINTER_READY,RECORD_READY,
          CLEAR_LEADER_TWO,CLEAR_LEADER_ONE,PLAY_ERROR,PLAY_READY:
          BOOLEAN END;

PROCEDURE F$BSPACE( VAR F: TEXT ); EXTERNAL;
PROCEDURE F$CREATECHANNEL( VAR F: ANYFILE ); EXTERNAL;
PROCEDURE F$MASTER( VAR F: ANYFILE ); EXTERNAL;
PROCEDURE F$STMODE( VAR F: ANYFILE; MODE: CHANNEL_MODE ); EXTERNAL;
PROCEDURE F$STLENGTH( VAR F: ANYFILE; LENGTH: INTEGER ); EXTERNAL;
PROCEDURE F$ULENGTH( VAR F: ANYFILE ); EXTERNAL;
PROCEDURE F$STEOC( VAR F: ANYFILE ); EXTERNAL;
FUNCTION F$EOC( VAR F: ANYFILE ): BOOLEAN; EXTERNAL;
PROCEDURE F$XACCESS( VAR F: ANYFILE ); EXTERNAL;
PROCEDURE F$WAIT( VAR F: ANYFILE; VAR M: CHANNEL_MODE ); EXTERNAL;
PROCEDURE CLOSE( VAR F: ANYFILE ); EXTERNAL;
FUNCTION COLUMN( VAR F: TEXT ): INTEGER; EXTERNAL;

PROCEDURE ERR$CLEAR; EXTERNAL;
FUNCTION ERR$CLASS: INTEGER; EXTERNAL;

PROCEDURE ONEXCEPTION( HANDLER_LOCATION: INTEGER ); EXTERNAL;
PROCEDURE EXCEPTION( CLASS_CODE, REASON_CODE: INTEGER ); EXTERNAL;

PROCEDURE INITSEMAPHORE( VAR S: SEMAPHORE; N: NONNEG ); EXTERNAL;
PROCEDURE TERMSEMAPHORE( VAR S: SEMAPHORE ); EXTERNAL;
PROCEDURE EXTERNALEVENT( S: SEMAPHORE; LEVEL: INTERRUPT_LEVEL );
  EXTERNAL;
PROCEDURE SIGNAL( S: SEMAPHORE ); EXTERNAL;
PROCEDURE WAIT( S: SEMAPHORE ); EXTERNAL;
PROCEDURE CSIGNAL( S: SEMAPHORE; VAR WAITER: BOOLEAN ); EXTERNAL;

PROCEDURE DELAY( INT: LONGINT ); EXTERNAL;

PROGRAM CLKINT; EXTERNAL;

PROGRAM ASRSUPERVISOR(
  BASE: CRU_ADDRESS;
  BAUD: INTEGER;
  ASRSUPERVISOR_LEVEL:  INTERRUPT_LEVEL;
  KEYBOARD_NAME: ALPHA;
  PRINTER_NAME: ALPHA;
  CASSETTE_ONE_NAME:  ALPHA;
  CASSETTE_TWO_NAME:  ALPHA;
  FILE_ORIENTED:  BOOLEAN;
  INITIALIZATION_COMPLETE: SEMAPHORE );
VAR
  PARTIAL_COMPLETION: SEMAPHORE;
  EXCLUSIVE_ACCESS: SEMAPHORE;
  ACKNOWLEDGE: SEMAPHORE;
  READ_READY: SEMAPHORE;
  WRITE_READY:  SEMAPHORE;
  DSR_READY: SEMAPHORE;
  INTERRUPT:  SEMAPHORE;
  CASSETTE_MODE: CHAR;
  STATUS: CASSETTE_STATUS;
  WANTED: BOOLEAN;

PROCEDURE INPUT9CHAR( VAR CH: CHAR);
BEGIN (* INPUT9CHAR *)
CRUBASE( BASE );
IF NOT TB( DATA_SET_READY ) THEN BEGIN
  SBO( STATUS_CHANGE_INTERRUPT_ENABLE );
  WAIT( DSR_READY );
  END  (* IF TB( DATA_SET_READY ) *);
SBO( RECEIVER_INTERRUPT_ENABLE );
WAIT( READ_READY );
IF TB( RECEIVE_ERROR ) THEN BEGIN
  SIGNAL( ACKNOWLEDGE );
  EXCEPTION( USER_ERROR, 0);
  END  (* IF TB( RECEIVE_ERROR ) *);
STCR( 8, CH::INTEGER );
SIGNAL( ACKNOWLEDGE );
END (* INPUT9CHAR *);

PROCEDURE OUTPUT9CHAR( CH: CHAR; DEVICE: DEVICE_TYPE );
BEGIN  (* OUTPUT9CHAR *)
CRUBASE( BASE );
IF NOT TB( DATA_SET_READY ) THEN BEGIN
  SBO( STATUS_CHANGE_INTERRUPT_ENABLE );
  WAIT( DSR_READY );
  END  (* IF TB( DATA_SET_READY ) *);
SBO( TRANSMIT_INTERRUPT_ENABLE );
WAIT( WRITE_READY );
SBO( REQUEST_TO_SEND );
LDCR( 8, CH::INTEGER );
SBZ( REQUEST_TO_SEND );
SIGNAL( ACKNOWLEDGE );
IF DEVICE = PRINT THEN
  IF CH = CR AND BAUD >= ASR_RATE THEN DELAY( 200 )
  ELSE IF BAUD = ASR_RATE THEN DELAY ( 25 );
END  (* OUTPUT9CHAR *);

PROCEDURE SEND9CONTROL( CONTROL_CHAR: CHAR );
BEGIN  (* SEND9CONTROL *)
OUTPUT9CHAR( DLE, PRINT );
OUTPUT9CHAR( CONTROL_CHAR, TAPE );
END (* SEND CONTROL *);

PROCEDURE SET9MODE( NEW_MODE: MODE; CASSETTE: CASSETTE_SIDE );
VAR
```

```
  OLD_MODE: CHAR;
BEGIN  (* SET9MODE *)
OLD_MODE := CASSETTE_MODE;
IF ( NEW_MODE = PLAYBACK AND CASSETTE = LEFT_CASSETTE ) OR
   ( NEW_MODE = RECORDING AND CASSETTE = RIGHT_CASSETTE ) THEN
    CASSETTE_MODE := '6'
    ELSE CASSETTE_MODE := '5';
IF CASSETTE_MODE <> OLD_MODE THEN BEGIN
  DELAY( 900 );
  SEND9CONTROL( CASSETTE_MODE );
  END  (* IF CASSETTE_MODE <> OLD_MODE *);
SEND9CONTROL( REQUEST_STATUS );
INPUT9CHAR( STATUS::CHAR );
END  (* SET9MODE *);

PROCESS KBDINTERFACE(
  INFILE: TEXT );

PROCEDURE KBDWORK;
VAR
  CH: CHAR;
BEGIN  (* KBDWORK *)
WHILE TRUE DO BEGIN
  REWRITE( INFILE );
  WHILE NOT F$EOC( INFILE ) DO BEGIN
    WAIT( EXCLUSIVE_ACCESS );
    INPUT9CHAR( CH );
    WHILE CH <> DC3 AND CH <> CR DO BEGIN
      IF ( CH = BS OR EOLN( INFILE ) ) AND COLUMN( INFILE ) > 0 THEN
        BEGIN
        OUTPUT9CHAR( BS, PRINT );
        F$BSPACE( INFILE );
        END  (* IF BS OR EOLN AND COLUMN > 0 *);
      IF CH = HT THEN BEGIN
       WRITE( INFILE, HT );
       OUTPUT9CHAR( SPACE, PRINT );
       END  (* IF CH = HT *)
       ELSE IF CH = LF THEN
         OUTPUT9CHAR( LF, PRINT )
         ELSE IF CH >= SPACE THEN BEGIN
           OUTPUT9CHAR( CH, PRINT );
           WRITE( INFILE, CH);
           END  (* IF CH >= SPACE *);
    INPUT9CHAR( CH );
    END  (* WHILE CH <> DC3 AND CH <> CR *);
    OUTPUT9CHAR( CR, PRINT );
    OUTPUT9CHAR( LF, PRINT );
    SIGNAL( EXCLUSIVE_ACCESS );
    IF CH = CR THEN WRITELN( INFILE )
      ELSE IF CH = DC3 THEN REWRITE( INFILE )
    END  (* WHILE NOT F$EOC( INFILE ) *);
  END  (* WHILE TRUE DO *);
END  (* KBDWORK *);

PROCEDURE KBDEXCEPTION;
BEGIN  (* KBDEXCEPTION *)
IF ERR$CLASS = FCLASS_FILE_ERROR THEN BEGIN
  ERR$CLEAR;
  KBDWORK;
  END  (* IF ERR$CLASS = FCLASS_FILE_ERROR *);
END  (* KBDEXCEPTION *);
BEGIN  (* KBDINTERFACE *)
(*# PRIORITY = KEYBOARD_LEVEL;
    STACKSIZE = KEYBOARD_STACK;
    HEAPSIZE = KEYBOARD_HEAP *)
F$MASTER( INFILE );
IF FILE_ORIENTED THEN F$XACCESS( INFILE );
F$STLENGTH( INFILE, 80 );
F$ULENGTH( INFILE );
F$STMODE( INFILE, WRITING );
F$STEOC( INFILE );
F$CREATECHANNEL( INFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( KBDEXCEPTION ) );
KBDWORK;
END  (* KBDINTERFACE *);

PROCESS PRTINTERFACE(
  OUTFILE: TEXT );

PROCEDURE PRTWORK;
VAR
  CH: CHAR;
BEGIN  (* PRTWORK *)
WHILE TRUE DO BEGIN
  RESET( OUTFILE );
  WHILE NOT EOF( OUTFILE ) DO BEGIN
    WAIT( EXCLUSIVE_ACCESS );
    WHILE NOT EOLN( OUTFILE ) DO BEGIN
      READ( OUTFILE, CH );
      IF CH = HT THEN
        OUTPUT9CHAR( SPACE, PRINT )
        ELSE IF CH = FF THEN
          FOR I := 1 TO 8 DO
            OUTPUT9CHAR( LF, PRINT )
          ELSE IF CH = BELL OR
                  CH = BS OR
                  CH = LF OR
                  CH = CR OR
                  CH >= SPACE THEN
            OUTPUT9CHAR( CH, PRINT );
      END  (* WHILE NOT EOLN *);
    IF CH>= SPACE THEN BEGIN
      OUTPUT9CHAR( CR, PRINT);
      OUTPUT9CHAR( LF, PRINT);
      END  (* IF CH >= SPACE *);
    SIGNAL( EXCLUSIVE_ACCESS );
    READLN( OUTFILE );
    END  (* WHILE NOT EOF *);
  END  (* WHILE TRUE DO *);
END  (* PRTWORK *);

PROCEDURE PRTEXCEPTION;
BEGIN  (* PRTEXCEPTION *)
IF ERR$CLASS = FCLASS_FILE_ERROR THEN BEGIN
  ERR$CLEAR;
  PRTWORK;
  END  (* IF ERR$CLASS = FCLASS_FILE_ERROR *);
END  (* PRTEXCEPTION *);
BEGIN  (* PRTINTERFACE *)
(*# PRIORITY = PRINTER_LEVEL;
    STACKSIZE = PRINTER_STACK;
    HEAPSIZE = PRINTER_HEAP *)
F$MASTER( OUTFILE );
IF FILE_ORIENTED THEN F$XACCESS( OUTFILE );
F$STLENGTH( OUTFILE, 80 );
F$ULENGTH( OUTFILE );
F$STMODE( OUTFILE, READING );
F$CREATECHANNEL( OUTFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( PRTEXCEPTION ) );
PRTWORK;
END  (* PRTINTERFACE *);

PROCESS CASINTERFACE(
  IOFILE: TEXT;
  CASSETTE: CASSETTE_SIDE );
VAR
  USERS_MODE: CHANNEL_MODE;

PROCEDURE CASWORK;

PROCEDURE CAS9OUT9WORK;
VAR
  CH: CHAR;
BEGIN  (* CAS9OUT9WORK *);
RESET( IOFILE );
WHILE NOT EOF( IOFILE ) DO BEGIN
  WAIT( EXCLUSIVE_ACCESS );
  SET9MODE( RECORDING, CASSETTE );
  IF NOT STATUS.RECORD_READY THEN
    EXCEPTION( USER_ERROR, 0 );
  OUTPUT9CHAR( DC2, PRINT );
  WHILE NOT EOLN( IOFILE ) DO BEGIN
    READ( IOFILE, CH );
    IF CH = CR THEN OUTPUT9CHAR( ETB, TAPE )
    ELSE IF CH = HT OR
            CH = FF OR
            CH = BELL OR
            CH = BS OR
            CH >= SPACE THEN OUTPUT9CHAR( CH, TAPE );
    END  (* WHILE NOT EOLN *);
  OUTPUT9CHAR( CR, TAPE );
  OUTPUT9CHAR( LF, TAPE );
  OUTPUT9CHAR( DC4, TAPE );
  OUTPUT9CHAR( DEL, PRINT );
  SIGNAL( EXCLUSIVE_ACCESS );
  READLN( IOFILE );
  END  (* WHILE NOT EOF *);
WAIT( EXCLUSIVE_ACCESS );
SET9MODE( RECORDING, CASSETTE );
IF NOT STATUS.RECORD_READY THEN
  EXCEPTION( USER_ERROR, 0 );
OUTPUT9CHAR( DC2, PRINT);
OUTPUT9CHAR( DC3, TAPE );
OUTPUT9CHAR( CR, TAPE );
OUTPUT9CHAR( DC4, TAPE );
OUTPUT9CHAR( DEL, PRINT );
SIGNAL( EXCLUSIVE_ACCESS );
CLOSE( IOFILE )
END  (* CAS9OUT9WORK *);

PROCEDURE CAS9IN9WORK;
VAR
  CH: CHAR;
  END_OF_FILE: BOOLEAN;
BEGIN  (* CAS9IN9WORK *)
REWRITE( IOFILE );
END_OF_FILE := FALSE;
WHILE NOT F$EOC( IOFILE ) AND NOT END_OF_FILE DO BEGIN
  WAIT( EXCLUSIVE_ACCESS );
  SET9MODE( PLAYBACK, CASSETTE );
  IF NOT STATUS.PLAY_READY THEN
    EXCEPTION( USER_ERROR, 0 );
  SEND9CONTROL( BLOCK_FORWARD );
  REPEAT
    INPUT9CHAR( CH );
    UNTIL CH<> LF OR CH<> DEL;
  IF CH    DC3 THEN BEGIN
    WHILE CH<> CR AND NOT EOLN( IOFILE ) DO BEGIN
      IF CH = ETB THEN WRITE( IOFILE, CR )
      ELSE IF CH = HT OR
              CH = FF OR
              CH = BELL OR
              CH = BS OR
              CH >= SPACE THEN WRITE( IOFILE, CH )
      ELSE IF CH = DC3 THEN SEND9CONTROL( BLOCK_FORWARD );
      INPUT9CHAR( CH );
      END  (* WHILE CH <> CR AND NOT EOLN *);
    END  (* IF CH<> DC3 *)
    ELSE END_OF_FILE := TRUE;
  SIGNAL( EXCLUSIVE_ACCESS );
  IF NOT END_OF_FILE THEN WRITELN( IOFILE );
  END  (* WHILE NOT F$EOC OR END_OF-FILE *);
CLOSE( IOFILE );
END  (* CAS9IN9WORK *);

BEGIN  (* CASWORK *)
WHILE TRUE DO BEGIN
  F$WAIT( IOFILE, USERS_MODE );
  CASE USERS_MODE OF
    READING:  CAS9IN9WORK;
    WRITING:  CAS9OUT9WORK
    END  (* CASE *);
  END  (* WHILE TRUE DO *);
END  (* CASWORK *);

PROCEDURE CASEXCEPTION;
BEGIN  (* CASEXCEPTION *)
IF ERR$CLASS = FCLASS_FILE_ERROR THEN BEGIN
  ERR$CLEAR;
  CASWORK;
  END  (* IF ERR$CLASS = FCLASS_FILE_ERROR *);
END  (* CASEXCEPTION *);

BEGIN  (* CASINTERFACE *)
(*# PRIORITY = CASSETTE_LEVEL;
    STACKSIZE = CASSETTE_STACK;
    HEAPSIZE = CASSETTE_HEAP *)
F$MASTER( IOFILE );
F$XACCESS( IOFILE );
F$STLENGTH( IOFILE, 80 );
F$ULENGTH( IOFILE );
F$STMODE( IOFILE, USERMODE );
F$STEOC( IOFILE );
F$CREATECHANNEL( IOFILE );
SIGNAL( PARTIAL_COMPLETION );
ONEXCEPTION( LOCATION( CASEXCEPTION ) );
CASWORK;
END  (* CASINTERFACE *);

BEGIN (* ASRSUPERVISOR *)
(*# PRIORITY = ASRSUPERVISOR_LEVEL;
    STACKSIZE = ASRSUPERVISOR_STACK;
    HEAPSIZE = ASRSUPERVISOR_HEAP *)
CRUBASE( TMS9901 )  (* SET CRU BASE TO 9901 *);
SBZ( CONTROL_BIT )  (* SET TO INTERRUPT MODE *);
CRUBASE( TMS9901 +( 2* ASRSUPERVISOR_LEVEL ) );
LDCR( 1, 1 )      (* ENABLE INTERRUPT MASK *);
CRUBASE( BASE )     (* SET CRU BASE TO 9902*);
```

```
SBO( DEVICE_RESET )        (* RESET 9902 *);
LDCR( 8, CONTROL_REG )     (* LOAD CONTROL REGISTER *);
SBZ( INT_REG_LOAD_FLAG )   (* SKIP INTERVAL REGISTER LOAD *);
LDCR( 12, BAUD)     (* LOAD TRANS/RECEV DATA RATE REGISTER *);
CASSETTE_MODE := 'Z';
INITSEMAPHORE( PARTIAL_COMPLETION, 0 );
INITSEMAPHORE( EXCLUSIVE_ACCESS, 1 );
INITSEMAPHORE( ACKNOWLEDGE, 0 );
INITSEMAPHORE( READ_READY, 0 );
INITSEMAPHORE( WRITE_READY, 0 );
INITSEMAPHORE( DSR_READY, 0 );
INITSEMAPHORE( INTERRUPT, 0 );
EXTERNALEVENT( INTERRUPT, ASRSUPERVISOR_LEVEL );
IF KEYBOARD_NAME <> NULL THEN BEGIN
  START KBDINTERFACE( FILENAMED( KEYBOARD_NAME ) );
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF PRINTER_NAME <> NULL THEN BEGIN
  START PRTINTERFACE( FILENAMED( PRINTER_NAME ) );
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF CASSETTE_ONE_NAME <> NULL THEN BEGIN
  START CASINTERFACE( FILENAMED( CASSETTE_ONE_NAME ), LEFT_CASSETTE )
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
IF CASSETTE_TWO_NAME <> NULL THEN BEGIN
  START CASINTERFACE( FILENAMED( CASSETTE_TWO_NAME ), RIGHT_CASSETTE
  WAIT( PARTIAL_COMPLETION );
  END  (* IF *);
SIGNAL( INITIALIZATION_COMPLETE );
TERMSEMAPHORE( PARTIAL_COMPLETION );
WHILE TRUE DO BEGIN
  WAIT( INTERRUPT );
  IF TB( TRANSMIT_INTERRUPT ) THEN BEGIN
    SBZ(TRANSMIT_INTERRUPT_ENABLE);
    SIGNAL( WRITE_READY );
    WAIT( ACKNOWLEDGE );
    END (* IF TB( TRANSMIT_INTERRUPT ) *)
    ELSE IF TB( RECEIVE_INTERRUPT ) THEN BEGIN
      SBZ( RECEIVER_INTERRUPT_ENABLE );
      SIGNAL( READ_READY );
      WAIT( ACKNOWLEDGE );
      END  (* IF TB( RECEIVE_INTERRUPT ) *)
      ELSE IF TB( STATUS_CHANGE_INTERRUPT ) THEN BEGIN
        REPEAT CSIGNAL( DSR_READY, WANTED )
          UNTIL NOT WANTED;
        SBZ( STATUS_CHANGE_INTERRUPT_ENABLE );
        END  (* IF TB( STATUS_CHANGE_INTERRUPT ) *)
        ELSE EXCEPTION( USER_ERROR, 1 );
  END  (* WHILE TRUE DO *);
END  (* ASRSUPERVISOR *);
```

```
PROCEDURE ASR(
  BASE: CRU_ADDRESS;
  BAUD: INTEGER;
  LEVEL:  INTERRUPT_LEVEL;
  KEYBOARD_NAME: ALPHA;
  PRINTER_NAME: ALPHA;
  CASSETTE_ONE_NAME:  ALPHA;
  CASSETTE_TWO_NAME:  ALPHA;
  FILE_ORIENTED:  BOOLEAN);
VAR
  INITIALIZATION_COMPLETE:  SEMAPHORE;
BEGIN (* ASR *)
INITSEMAPHORE( INITIALIZATION_COMPLETE,0 );
START ASRSUPERVISOR( BASE, BAUD, LEVEL, KEYBOARD_NAME,
  PRINTER_NAME, CASSETTE_ONE_NAME, CASSETTE_TWO_NAME,
  FILE_ORIENTED, INITIALIZATION_COMPLETE );
WAIT( INITIALIZATION_COMPLETE );
TERMSEMAPHORE( INITIALIZATION_COMPLETE );
END (* ASR *);

BEGIN (* USER$APPL *)
  ...
END(* USER$APPL *);

BEGIN  (* $NULLBODY *)
END.
```

Figure 13. ASR Physical Device Interface System

APPENDIX M

RTS Mailbox Manager

Microprocessor Pascal System Executive RTS provides a mailbox manager to facilitate inter-process communication. The mailbox routines manage a message queue and free the user from details of process synchronization.

```
        CONST
           MSGSIZ = 80;

        TYPE
           MSGPTR = @MSG;
           MBPTR  = @MAILBOX;

           MSG    = RECORD
                  NEXTMSG : INTEGER;
                  RESPONSE : SEMAPHORE;
                  MSGSIZE : INTEGER;
                  CMD: ( R, W );
                  MSGTEXT : PACKED ARRAY (.1..MSGSIZE.) OF CHAR;
                  END;

           MAILBOX = RECORD
                   MAILPRESENT, MUTEX : SEMAPHORE;
                   MSGHEAD, MSGTAIL : MSGPTR;
                   END;
```

These TYPEs are fundamental to the operation of the message handler, and must be declared by the user. For two or more processes to communicate through this system, the user must then declare a mailbox, a message record, and pointers to these records with these TYPEs.

The user is also responsible for initializing MAILPRESENT and MUTEX (via INITSEMAPHORE). MAILPRESENT should be initialized with an event count of zero, and MUTEX should be initialized with a count of one. In addition, MSGHEAD should be initially set to NIL. The use of the RESPONSE, MSGSIZE, and CMD fields of MSG is at the discretion of the user.

```
        procedure sndmsg(m: msgptr; addressee: mbptr);
                     external;
```

This procedure enters a message into a mailbox queue. M is a pointer to the message, and ADDRESSEE is a pointer to the mailbox.

```
        procedure rcvmsg(var m: msgptr; addressee: mbptr);
                     external;
```

This procedure receives a message from a mailbox and removes it from the queue. M is a pointer that (upon return from RCVMSG) points to the received message, and ADDRESSEE is a pointer to the mailbox from which the message is to be received. If no messages exist at call time, the process is suspended until a message is entered into the mailbox queue (via SNDMSG).

```
        procedure delmsg(m: msgptr; addressee: mbptr);
                     external;
```

This procedure deletes a message from a mailbox queue. M is a pointer to the message to be deleted, and ADDRESSEE is a pointer to the mailbox queue.

The source code for the mailbox manager is shown below:

```
(*$ MAP, DEBUG *)

PROGRAM MAILBOXES;

CONST
  MSGSIZ = 80;

TYPE
  MSGPTR = @MSG;
  MBPTR  = @MAILBOX;

  MSG    = RECORD
         NEXTMSG : MSGPTR;
         RESPONSE : SEMAPHORE;
         MSGSIZE : INTEGER;
         CMD : ( R, W );
         MSGTEXT : PACKED ARRAY (.1..MSGSIZ.) OF CHAR;
         END;

  MAILBOX = RECORD
          MAILPRESENT, MUTEX : SEMAPHORE;
          MSGHEAD, MSGTAIL : MSGPTR;
          END;

PROCEDURE SIGNAL ( S : SEMAPHORE ); EXTERNAL;
PROCEDURE WAIT   ( S : SEMAPHORE ); EXTERNAL;
PROCEDURE INITSEMAPHORE ( VAR S : SEMAPHORE ; VALUE : INTEGER);
          EXTERNAL;
PROCEDURE TERMSEMAPHORE ( VAR S : SEMAPHORE ); EXTERNAL;

PROCEDURE SNDMSG ( M : MSGPTR ; ADDRESSEE : MBPTR ); FORWARD;
PROCEDURE RCVMSG ( VAR M : MSGPTR ; ADDRESSEE : MBPTR ); FORWARD;
PROCEDURE DELMSG ( M : MSGPTR ; ADDRESSEE : MBPTR ); FORWARD;
PROCEDURE SNDMSG (* M : MSGPTR ; ADDRESSEE : MBPTR *);
(*-----------------------------------------------------------------------
PURPOSE:
  ENTER A MESSAGE INTO A MAILBOX QUEUE.
INPUTS:
  M         : POINTER TO THE MESSAGE.
  ADDRESSEE : POINTER TO THE MAILBOX.
PROCEDURES CALLED:
  WAIT, SIGNAL.
OUTPUTS:
  NONE.
EXCEPTIONS:
  NONE.
HISTORY:
  04/08/79: ORIGINAL.
------------------------------------------------------------------------

BEGIN
  WITH M@ DO BEGIN
    NEXTMSG := NIL;
    WITH ADDRESSEE@ DO BEGIN
      WAIT ( MUTEX );
      IF MSGHEAD <> NIL
        THEN MSGTAIL@.NEXTMSG := M
        ELSE MSGHEAD := M;
      MSGTAIL := M ;
      SIGNAL ( MAILPRESENT );
      SIGNAL ( MUTEX );
    END; (* WITH ADDRESSEE@ *)
  END; (* WITH M@ *)
END; (* SNDMSG *)
```

```
PROCEDURE RCVMSG (* VAR M : MSGPTR ; ADDRESSEE : MBPTR *);
(*-------------------------------------------------------------------
PURPOSE:
  RECEIVE A MESSAGE FROM A MAILBOX AND REMOVE IT FROM THE QUEUE.
INPUTS:
  ADDRESSEE : POINTER TO THE MAILBOX FROM WHICH A MESSAGE IS
  TO BE RECEIVED.
PROCEDURES CALLED:
  WAIT, SIGNAL.
OUTPUTS:
  M : A POINTER TO THE RECEIVED MESSAGE.
EXCEPTIONS:
  THE PROCESS IS SUSPENDED UNTIL A MESSAGE IS ENTERED INTO THE
  QUEUE IF NONE EXIST AT CALL TIME.
HISTORY:
  04/08/79: ORIGINAL.
---------------------------------------------------------------------

BEGIN
  WITH ADDRESSEE@ DO BEGIN
    WAIT ( MAILPRESENT );
    WAIT ( MUTEX );
    M := MSGHEAD;
    IF MSGTAIL = MSGHEAD
      THEN MSGTAIL := NIL;
    MSGHEAD := M@.NEXTMSG;
    SIGNAL ( MUTEX );
  END; (* WITH ADDRESSEE@ *)
END; (* RCVMSG *)
```

```
PROCEDURE DELMSG (* M : MSGPTR ; ADDRESSEE : MBPTR *);
(*-----------------------------------------------------------------------
PURPOSE:
  DELETE A MESSAGE FROM A MAILBOX QUEUE.
INPUTS:
  M         : POINTER TO THE MESSAGE TO BE DELETED.
  ADDRESSEE : POINTER TO THE MAILBOX CONTAINING THE QUEUE
  FROM WHICH THE MESSAGE IS TO BE DELETED.
PROCEDURES CALLED:
  WAIT, SIGNAL.
OUTPUTS:
  NONE.
EXCEPTIONS:
  NONE.
HISTORY:
  04/08/79: ORIGINAL.
------------------------------------------------------------------------

VAR
  LAST  : MSGPTR;
  FOUND : BOOLEAN;

BEGIN
  FOUND := FALSE;
  WITH ADDRESSEE@ DO BEGIN
    WAIT ( MUTEX );
    IF MSGHEAD = M
      THEN BEGIN
        MSGHEAD := M@.NEXTMSG;
        FOUND := TRUE;
      END  (* IF MSGHEAD THEN *)
      ELSE BEGIN
        LAST := MSGHEAD;
        WHILE LAST <> NIL AND LAST@.NEXTMSG <> M
          DO LAST := LAST@.NEXTMSG;
        IF LAST    NIL THEN BEGIN
          LAST@.NEXTMSG := M@.NEXTMSG;
          FOUND := TRUE;
          IF LAST@.NEXTMSG = NIL
            THEN MSGTAIL := LAST;
        END; (* IF LAST *)
      END; (* IF MSGHEAD ELSE *)
    IF FOUND
      THEN WAIT ( MAILPRESENT );
    SIGNAL ( MUTEX );
  END; (* WITH ADDRESSEE@ *)
END; (* DELMSG *)


BEGIN (* MAILBOXES *)
  (* NULLBODY *)
END.  (* MAILBOXES *)
```

APPENDIX N

RTS Clock Interrupt Handler

N.1 OVERVIEW

Microprocessor Pascal Executive RTS supplies several clock handling routines which use the TMS 9901 interval timer to provide a timed wait facility for Microprocessor Pascal processes. The clock handler also provides for time-slicing for non-interrupt priority processes.

```
        program clkint; external;
```

This program initializes the clock handling routines and sets the TMS 9901 to generate clock interrupts at a given rate. For MPX, the 9901 generates an interrupt every 10.048 milliseconds; for MPIX the interval is 100.02 milliseconds.

The program maintains a common CLOCK containing an elapsed time counter which registers the number of milliseconds which have elapsed since system startup. This counter will wrap around every 24.8 days (31 bits).

The program implements time-slicing for non-interrupt priority processes by calling the procedure SWAP once every five interrupts. This corresponds to once every 50.24 milliseconds for MPX and once every 500.1 milliseconds for MPIX. Time slicing ensures that among non-device processes of equal priority no one process will maintain exclusive control of the processor; SWAP reorders the scheduling queue so that other processes may execute. Note that time-slicing has no effect if the most urgent non-device process is the only process of a given priority.

```
        procedure twait(var s: semaphore; ms: longint; var b:
        boolean); external;
```

This procedure performs a timed wait on semaphore S for MS milliseconds. Variable B is set to TRUE if the semaphore is signalled before MS milliseconds elapse; B is set to FALSE if the time-out occurs before S is signalled.

The procedure inserts semaphore S into a delay queue and then uses the elapsed-time counter initialized by CLKINT to check when the semaphore is signalled.

```
        procedure delay(ms: longint); external;
```

This procedure provides a timed delay of execution for

Microprocessor Pascal System processes. MS is the number of milliseconds that a process is to be delayed.

DELAY operates by initializing a semaphore and then calling TWAIT to perform a timed wait on this semaphore.

N.2 IMPLEMENTATION

The source listing of the clock handler (MPIX version) is given below. If necessary, the constant values defined in CLKINT may be changed to reflect individual installation-dependent factors. These constants are as follows:

CLKCRU - The CRU base used for communication with the TMS 9901 Systems Interface. This value should be #0100 for both MPX and MPIX.

CLKLVL - The clock interrupt level. The TMS 9901 uses level three for clock interrupts.

INTBIT - The value of the CRU control bit. Zero indicates interrupt mode.

CLKBIT - Enables an interrupt at this level by setting the corresponding CRU bit to one.

COUNTS - The value used to program the TMS 9901 for the proper clock interrupt interval. This interval (in milliseconds) is given by (COUNTS * 21.33)/1000. The value 21.33 is the clock resoluton of the 9901, which operates at 3 MHz. COUNTS may be changed to effect a different clock interval.

MILLIS - This value is the whole millisecond part of the clock interval. It will need to be changed whenever COUNTS is changed.

MICROS - This value is the microsecond part of the clock interval. It will need to be changed whenever COUNTS is changed.

QUANTUM - This value indicates the number of clock interrupts that are to be serviced in each time slice; it may be changed.

```
(*$ MAP, DEBUG *)

SYSTEM  OUTSIDE;

TYPE

  INTLVL = 0..15;

  PROCESSPTR = @ INTEGER;

  TELT         = RECORD
                 TNXT : @ TELT;
                 TSEM : SEMAPHORE;
                 TMS  : LONGINT;
                 END;

COMMON

  CLOCK        : RECORD
                 ELAPSED : LONGINT;
                 TIMEQ : @ TELT;
                 END;


PROCEDURE SWAP ; EXTERNAL;
PROCEDURE MASK ; EXTERNAL;
PROCEDURE UNMASK ; EXTERNAL;
PROCEDURE TERMSEMAPHORE ( VAR S : SEMAPHORE );
          EXTERNAL;
PROCEDURE INITSEMAPHORE ( VAR S : SEMAPHORE ; VALUE : INTEGER);
          EXTERNAL;
PROCEDURE EXTERNALEVENT ( S : SEMAPHORE ; LEVEL : INTLVL );
          EXTERNAL;
PROCEDURE SIGNAL ( S : SEMAPHORE ); EXTERNAL;
PROCEDURE WAIT ( S : SEMAPHORE ); EXTERNAL;
PROCEDURE TWAIT ( VAR S : SEMAPHORE; MS : LONGINT; VAR B : BOOLEAN );
(*-------------------------------------------------------------------
PURPOSE:
  PROVIDE A TIME-OUT ON SEMAPHORE WAIT OPERATION.
INPUTS:
  S  : SEMAPHORE ON WHICH THE TIMED WAIT IS PERFORMED.
  MS : TIME-OUT COUNT IN MILLISECONDS.
PROCEDURES CALLED:
  MASK, WAIT, UNMASK.
OUTPUTS:
  B : BOOLEAN.  SET TO TRUE WHEN SEMAPHORE IS SIGNALLED BEFORE
                TIMING OUT.
                SET TO FALSE IF TIME-OUT OCCURS BEFORE SEMAPHORE
                IS SIGNALLED.
EXCEPTIONS:
  NONE.
HISTORY:
  06/11/79: REVISION.
--------------------------------------------------------------------

VAR
  T      : TELT;
  Q,P    : @TELT;

ACCESS
  CLOCK;

BEGIN
  WITH T DO BEGIN
    TSEM::INTEGER := S::INTEGER;
    WITH CLOCK DO BEGIN
      TMS := ELAPSED + MS;
      MASK;
      P := TIMEQ;
      IF P = NIL OR P@.TMS > TMS
        THEN TIMEQ::INTEGER := LOCATION ( T )
        ELSE BEGIN
          REPEAT
            Q := P;
            P := P@.TNXT;
          UNTIL P = NIL OR P@.TMS > TMS;
          Q@.TNXT::INTEGER := LOCATION ( T );
        END; (* IF ELSE *)

      TNXT := P;

      WAIT ( S );

      B := NOT ODD ( TNXT::INTEGER );
      IF B
        THEN BEGIN
          P := TIMEQ;
          IF P::INTEGER = LOCATION ( T )
            THEN TIMEQ := TNXT
            ELSE BEGIN
              REPEAT
                Q := P;
                P := P@.TNXT;
              UNTIL P::INTEGER = LOCATION ( T );
              Q@.TNXT := TNXT;
            END; (* IF P ELSE *)
        END; (* IF B THEN *)
      UNMASK;
    END; (* WITH CLOCK *)
  END; (* WITH T *)
END; (* TWAIT *)
```

```
PROCEDURE DELAY ( MS : LONGINT );
(*-----------------------------------------------------------------------
PURPOSE:
  PROVIDE TIMED DELAY OF EXECUTION FOR MPIX PROCESSES.
INPUTS:
  MS : TIME OF DELAY REQUESTED IN MILLISECONDS.
PROCEDURES CALLED:
  INITSEMAPHORE, TWAIT, TERMSEMAPHORE.
OUTPUTS:
  NONE.
EXCEPTIONS:
  NONE.
HISTORY:
  06/11/79: REVISION.
------------------------------------------------------------------------

VAR
  S : SEMAPHORE;
  B : BOOLEAN;

BEGIN
  INITSEMAPHORE ( S , 0 );
  TWAIT ( S, MS, B );
  TERMSEMAPHORE ( S );
END; (* DELAY *)
```

```
PROGRAM CLKINT;
(*-----------------------------------------------------------------------
PURPOSE:
  PROVIDE REAL TIME CLOCK FOR MPIX :
  INITIALIZE TMS 9901 FOR 100.02 MS CLOCK INTERRUPTS, MANAGE TIME
  DELAY QUEUE, AND PERFORM TIME SLICING ( @ 500.1 MS PER SLICE ).
INPUTS:
  NONE.
PROCEDURES CALLED:
  INITSEMAPHORE, EXTERNALEVENT, NEW, WAIT, SIGNAL, SWAP.
OUTPUTS:
  NONE.
EXCEPTIONS:
  NONE.
HISTORY:
  06/11/79: REVISION.
------------------------------------------------------------------------

CONST

  CLKCRU  = #0100;
  CLKLVL  =     3;
  INTBIT  =     0;
  CLKBIT  =     3;
  COUNTS  =  4689;
  MILLIS  =   100;
  MICROS  =    20;
  QUANTUM =     5;

VAR

  SLOP  : INTEGER;
  TIMER : INTEGER;
  TQ    : @ TELT;
  INTERRUPT : SEMAPHORE;

ACCESS
  CLOCK;
```

```
BEGIN    (* PROGRAM CLKINT *)

(*# STACKSIZE = 256; HEAPSIZE = 64; PRIORITY = CLKLVL *)

  INITSEMAPHORE ( INTERRUPT, 0 );
  EXTERNALEVENT( INTERRUPT, CLKLVL );

  WITH CLOCK DO BEGIN
    ELAPSED := 0;
    TIMEQ := NIL;
    SLOP := 0;
    TIMER := QUANTUM;

    CRUBASE ( CLKCRU  );
    LDCR( 15, COUNTS * 2 + 1 );

    WHILE TRUE DO BEGIN
      SBZ ( INTBIT );       (* SET INTERRUPT ON *)
      SBO ( CLKBIT );       (* ENABLE CLOCK INTERRUPT *)
      WAIT ( INTERRUPT );
      ELAPSED := ELAPSED + MILLIS;
      SLOP := SLOP + MICROS;
      IF SLOP >= 1000
        THEN BEGIN
          SLOP := SLOP - 1000;
          ELAPSED := ELAPSED + 1;
        END; (* IF SLOP *)

      WHILE TIMEQ <> NIL AND TIMEQ@.TMS <= ELAPSED DO BEGIN
        SIGNAL ( TIMEQ@.TSEM );
        TQ := TIMEQ@.TNXT;
        TIMEQ@.TNXT::INTEGER := TQ::INTEGER + 1;
        TIMEQ := TQ;
      END; (* WHILE *)

      TIMER := TIMER - 1;
      IF TIMER = 0
        THEN BEGIN
          TIMER := QUANTUM;
          SWAP;
        END; (* IF TIMER THEN *)

    END; (* WHILE TRUE *)
  END; (* WITH CLOCK *)
END; (* CLKINT *)
```

```
BEGIN (* OUTSIDE *)
(*# STACKSIZE = 256 *)
  START CLKINT;
END. (* OUTSIDE *)
```

INDEX

SOFTWARE TROUBLE REPORT

STR NUMBER ____________________
(for TI use)

SUBMITTED BY ____ DATE ______________
COMPANY ____ TEL. NO. ______________
MAILING ADDRESS ____
REPORTED TO ____ TEL. NO. ______________

CUSTOMER INFORMATION:
SOFTWARE/DOCUMENT NAME ____
TEXAS INSTRUMENTS PART NO. ____________ REVISION ________ VERSION ________
PROBLEM TYPE: (CIRCLE ONE)
SOFTWARE DOCUMENTATION OTHER (explain) ____________________
PROBLEM SUMMARY: ____

EQUIPEMNT DESCRIPTION:
MEMORY SIZE:
RAM __________
EPROM __________

	MANUFACTURER	MODEL
MICROCOMPUTER BOARD	____________________	__________
EXPANSION MEMORY BOARD(S)	____________________	__________
	____________________	__________
ADDITIONAL BOARDS IN SYSTEM	____________________	__________
	____________________	__________
	____________________	__________
	____________________	__________

PLEASE ATTACH ALL NECESSARY INFORMATION TO DUPLICATE THE ABOVE REPORTED DISCREPANCE. USE ADDITIONAL PAGES IF NECESSARY.

MAIL TO:

TEXAS INSTRUMENTS INCORPORATED
P.O. BOX 1443, M/S 6407
HOUSTON, TEXAS 77001

TEXAS INSTRUMENTS
INCORPORATED

Post Office Box 1443 / Houston, Texas 77001
Semiconductor Group

Printed in U.S.A.