TEXAS INSTRUMENTS

# 9900

# The Microprocessor Pascal System

MICROPROCESSOR SERIES™

User's Guide

[illegible]

TABLE OF CONTENTS

## SECTION 3. A SAMPLE MPP SYSTEM

## SECTION 4. SOURCE EDITOR GUIDE

## SECTION 6. HOST DEBUGGER GUIDE

## LIST OF ILLUSTRATIONS

## LIST OF TABLES

# SECTION 1

# MICROPROCESSOR PASCAL SYSTEM OVERVIEW

## 1.1 GENERAL

Microprocessor Pascal is a Texas Instrument´s product designed to support multi-tasking (concurrency) on Texas Instruments´ TM9900 microprocessors. Concurrency is one of the chief advantages of Microprocessor Pascal (MPP) because it provides a high level of computer-time saving by performing simultaneous integrated execution of a number of processes with a single processor.

### 1.1.1 Microprocessor Pascal System Features

The Microprocessor Pascal System is intended for applications to be executed on small computers that do not have a large, general-purpose operating system. Features supported by Microprocessor Pascal System are:

- Easy to learn
- Block structured language
- Compiler-enforced compatability checks
- User-defined data structures
- Interrupt handlers coded in Pascal or assembly language
- Concurrent execution of multiple tasks (processes)
- Process synchronization via semaphores
- Direct high-level language single-bit and byte I/O
- File level interprocess communication
- Dynamic creation and reclamation of a process´ resources
- Process scheduling according to a multiple-priority scheme
- Ability to customize applications to include only those those features of Executive Run-Time Support libraries that are required

## 1.2 WHAT IS MICROPROCESSOR PASCAL?

Pascal is a block-structured high-level language developed by Niklaus Wirth which enables the user to take a reliable, highly structured approach to designing and implementing an application program. High-level language means that each Microprocessor Pascal instruction corresponds to a group of machine level instructions, whereas, a low-level language instruction (e.g., 9900 Assembly Language) corresponds to a single instruction in machine code.

Texas Instruments' Microprocessor Pascal System is a software package development and execution of a superset of the Pascal language for use on the 9900 Family of microprocessors and TM990 microcomputer modules. Specifically, the Microprocessor Pascal System supports user development of an application on a host system and execution of that application on the 9900 or TM990 Target Systems.

The Microprocessor Pascal System can be divided into two groups: Host and Target. The Host is the system used specifically for the development of software. This group includes those tools necessary for entering and developing applications software such as the Editor, compiler, etc.

The Target group is the run-time support for executing the software and includes those tools necessary for implementing and maintaining a Target system environment, e.g., CPU scheduling routines, memory management routines, etc. These run-time support routines are grouped into libraries in such a manner that only those routines necessary for the execution of the application software can be selected by the user for inclusion in the executable object module. Selection and binding of the runtime-support software is performed by the link editor, and the object module produced will be the module that is executed on the Target System.

Tools provided in the Microprocessor Pascal System for Host software development consist of routines enabling the user to enter an application into a host computer system. Other tools are available for checking for errors in syntax, translating the source code into code that is executable by the central processor (CPU), and discovering and removing run-time errors that may occur during execution in the target system.

Routines available in the Microprocessor Pascal System for Target system run-time support provide the user with the means of implementing and maintaining a target system environment. The Target system environment controls software execution in the target system and includes CPU usage, system memory usage, routine calling conventions, data structures, etc. These run-time support routines are grouped together into libraries; only those routines required by the application will be included in the software executing on the target system. In addition, the run-time support provided allows the Microprocessor Pascal System package to be a concurrent (multitasking) system. Run-time routines support simultaneous sharing of a single processor by a number of routines (called processes) during execution.

In the Microprocessor Pascal System, these "processes" are separate sites of execution with their own environment.

## 1.3 WHY USE MICROPROCESSOR PASCAL?

Concurrency is only one of the advantages of Microprocessor Pascal. The following paragraph points out some of the other advantages.

An engineer wants to control his factory process with a dedicated microcomputer system. His factory process consists of five independent sub-processes. The engineer should design his process to satisfy the requirements of realtime logic flow, write his software so that the sequential execution of the code comprehends the relationship between the five independent sub-processes realtime logic flow and controls each sub-process independently. This approach requires a supervisor program to control which piece of code executes according to the realtime needs (priorities) of the factory process. Microprocessor Pascal System Run Time Support provides this supervisor. Furthermore, if the engineer writes his code in a block-structured and modular fashion, it will be more reliable and easier to add features to later. The Microprocessor Pascal System is such a language. Thus, the engineer can design and implement the control algorithm for his factory process in the same manner in which he perceives it.

Understanding concurrency and the features that support it in the Microprocessor Pascal System is a prerequisite to making efficient use of the system's resources. For this reason, the user should make sure that he understands the information defining these concepts which is presented in Section 2.

Execution of Microprocessor Pascal is in one of two modes, each supported by its own set of executive run-time libraries:

- Interpretive execution - Pseudo code (or PCODE) refers to the code resulting from execution of the Microprocessor Pascal compiler. Pseudo code is executed in the target via an interpreter (and for this reason, pseudo code may be referred to as interpretive code), a program that "looks at" each PCODE instruction in turn and executes a set piece of code to perform the task indicated.

- Native code - Native machine code (or object code) refers to the code resulting from execution of the Microprocessor Pascal computer and code generator. It is generated from the interpretive code produced by the Microprocessor Pascal compiler. Native code is executed directly by the microprocessor in the target environment.

This manual describes the run-time routines for Microprocessor Pascal interpretive execution (MPIX). Descriptions of run-time support routines for the native code user (MPX) are provided in the Microprocessor Pascal Executive User's Manual (MP385) included in this package.

The information that follows provides an overview look at the Microprocessor Pascal System describing in greater detail the system components introduced above. Figure 1-1 presents an overview of the Microprocessor Pascal System in diagrammatic form.

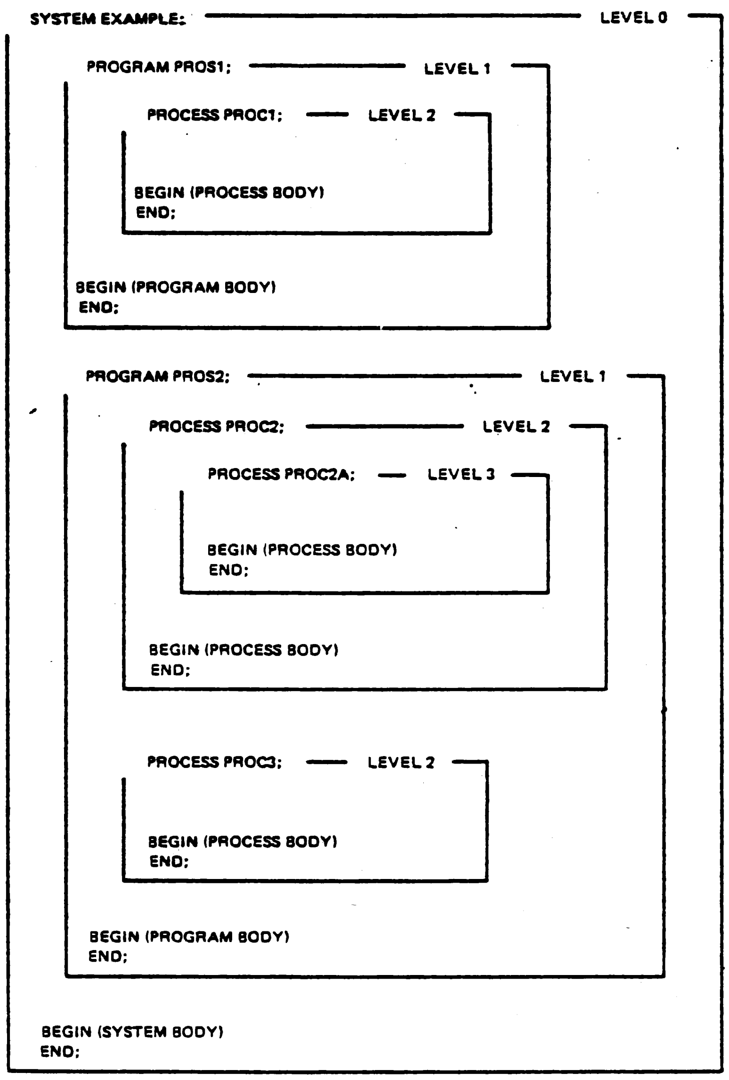


FIGURE 1-1.THE MICROPROCESSOR PASCAL SYSTEM.

## 1.4 HOST AND TARGET SYSTEMS

The user of the Microprocessor Pascal System develops software using one of the following host computer systems:

- Single-user FS990 (model 4 or 10) floppy disc development system with TX 990 system software
- Single-user TMAM 9000 table-top computer system with double-sided, double-density floppy disk drives and AMPLUS System software
- Multi-user DS990/10 hard disc minicomputer with DX system software

Once developed, the software can execute on the following target systems:

- 9900 16-bit Microprocessor chip set
- TM990 Microcomputer module with a memory expansion module

## 1.5 SOFTWARE DEVELOPMENT TOOLS

The Microprocessor Pascal System provides four major tools supporting software development on the host computer:

- An intelligent, interactive editor for source preparation which has syntax checking capability
- A compiler to generate interpretive code (PCODE) from source code
- A code generator to generate 9900 object code
- An interactive debugging interpreter

Each of these development tools is described briefly below.

### 1.5.1 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. In Compose mode, 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.

### 1.5.2 Pascal Compiler

The Microprocessor Pascal System provides a compiler that translates Microprocessor Pascal System source code into interpretive code. This code may be executed interpretively using the Host Debugger or using the interpreter in a target system. This code may also be used as input to the Native Code Generator (described in Subsection 1.3.3). Interpretive code is characterized by its compact size: a typical application generated in PCODE is about half the size of that same application generated in 9900 native code). One key advantage of interpretive code is the minimal time required to produce an executable system that can be debugged at a functional level. However, interpretive code runs somewhat slower than native code. The Microprocessor Pascal System Compiler processes the full Microprocessor Pascal System language and detects syntactic and semantic errors at the source level.

### 1.5.3 Native Code Generator

The Native Code Generator (NCG) translates the interpretive code from the compiler into 9900 native (object) code. Native code is less compact than interpretive code and executes five times faster. The native code option is provided to satisfy execution-time requirements that cannot be met by interpretive code.

### 1.5.4 Host Debugger

The Microprocessor Pascal System provides an interactive debugger which enables the user to debug application systems at a functional level (i.e., discover and remove errors that would occur if the application was executing on the target system). The debugger supports symbolic referencing of routines, files, etc. Statements can be referred to by Pascal statement numbers. Breakpoints can be used to stop the execution at any point by specifying the Pascal statement number of a particular routine. 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 variables for the process. Data can also be modified if desired. The execution of the system can be traced at various levels including the routine entry and exit level, or module statement flow level. Target hardware interfaces such as CRU references and interrupts may also be simulated in the debugging mode.

Please note: also supplied as an extension to the Microprocessor Pascal System are AMPL procs specifically designed to facilitate debug of a Microprocessor Pascal Target System. Information of use of these procs is provided in Section 19 of this manual.

## 1.6 EXECUTIVE RUN TIME SUPPORT

The executive components of the Microprocessor Pascal System are provided in two versions that correspond to the output generated by the Microprocessor Pascal System compiler and the output produced by the Native Code Generator. The Microprocessor Pascal Interpretive Executive (MPIX) supports interpretive execution of PCODE (produced by the compiler). It is generally used for applications for which program compaction achieved with interpretation is more important than the associated increase in execution speed. The native code run time support library introduced here as the Microprocessor Pascal Executive (MPX) supports execution of 9900 Native Code and is generally used for time-critical applications.

Actually, MPX is a set of routines that enables the user application to execute using Texas Instruments standalone executive called the Realtime Executive (Rx). The Realtime Executive User´s Manual 373) is included in the Microprocessor Pascal package. The Realtime Executive provides a run-time environment for any 9900-ba application regardless of application language.

Rx capabilities that are applicable to the Pascal user are documented in the Microprocessor Pascal Executive User´s Manual. Sections 9 through 14 of this manual document the RTS library for interpret code run-time support.

The relationship between Microprocessor Pascal and the Realtime Executive is displayed in Figure 1-2.

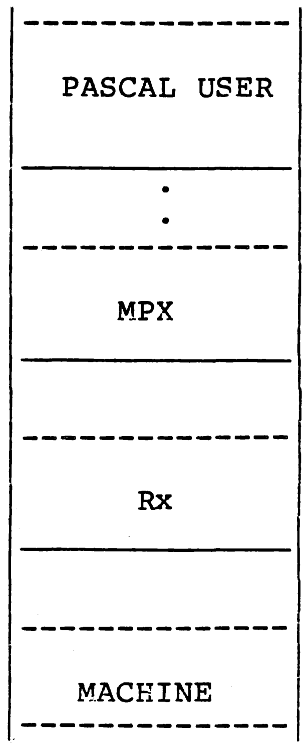


FIGURE 1-2. RELATIONSHIP BETWEEN THE PASCAL USER AND RX.

# SECTION 2

# MICROPROCESSOR PASCAL SYSTEM CONCEPTS

## 2.1 OVERVIEW

One measure of a system´s performance is the number of processes active at the same time, i.e., the level of concurrency within the system. (This may or may not be the same as the number of concurrent processes resident in the system.) Obviously, a system with 100 percent concurrency is most efficient regardless of the size of the job the system is performing. If everything can happen at once, the job will be completed in the briefest period of time.

Therefore, one goal in designing a computer system is to maximize the amount of concurrency in the system using one processor, but making the system appear as if it is doing everything at once. Obviously, with one processor, only one job can be performed at a time; but, if the appropriate constructs are set up to preserve a job´s environment, the processor can switch from job to job in an interleaving fashion, servicing each eligible task a little bit at a time. This is as close as one can get to doing everything at the same time in a single processor system. Achieving maximum concurrency is the goal of the Microprocessor Pascal System.

The constructs of the Microprocessor Pascal System are designed to support the multitasking concept of concurrency,i.e., one processor servicing many tasks or processes. This concept and the constructs in the Microprocessor Pascal System supporting it are described in the following paragraphs.

## 2.2 CONCURRENCY

To understand a concurrent processing system, simply compare it to the more familiar sequential programming system. Sequential programming means that one site of execution exists in a system at any one time. This program has the undivided attention of the processor and execution proceeds sequentially from instruction to instruction without interruption. When the program must wait for input or output, the processor is idle.

In a concurrent processing environment, several sites of execution may exist. These sites of execution are called processes in the Microprocessor Pascal System (see 2.3) and are simply separate sequential routines. When the execution of a process is blocked (waiting for input, output, etc.), the processor merely switches its attention to another available process that is not blocked.

The following example illustrates the increased efficiency realized when using concurrent rather than sequential processing. This example involves a processor controlling a soft drink bottling operation. There are several operations involved, each of which are performed at a different operating station.

(1) An empty bottle is positioned under a filler tube and filled.

(2) The filled bottle is then capped and checked for pressure leaks.

(3) The inspected bottle is then placed in carton until the carton is full.

(4) The full carton is then crated, and placed into inventory.

In a sequential operation, one bottle must proceed through all of the steps before the next bottle can begin. Thus the output rate is limited to the amount of time needed to perform all four steps in sequence. Figure 2-1 illustrates this sequential operation; time is plotted along the horizontal axis.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| STEP 1 | FILL 1 | | | | FILL 2 | | |
| STEP 2 | | CAP 1 | | | | CAP 2 | |
| STEP 3 | | | CARTON 1 | | | | ETC. |
| STEP 4 | | | | INVEN 1 | | | |
| | TIME 0 | TIME 1 | TIME 2 | TIME 3 | TIME 4 | TIME 5 | TIME 6 |

FIGURE 2-1. SEQUENTIALLY PERFORMED PROCESSES.

With concurrency (as illustrated in Figure 2-2), as a step is completed, the bottle is immediately passed to the next step; the just completed step is immediately repeated on a new bottle. Using concurrent processing, the output rate is one bottle per the time required for the slowest of the four steps.

| | | | | | | |
|---|---|---|---|---|---|---|
| STEP 1 | FILL 1 | FILL 2 | FILL 3 | FILL 4 | FILL 5 | FILL 6 |
| STEP 2 | | CAP 1 | CAP 2 | CAP 3 | CAP 4 | ..ETC |
| STEP 3 | | | CARTON 1 | CRTN 2 | CRTN 3 | ..ETC |
| STEP 4 | | | | INVEN. 1 | INVEN. 2 | INVEN 3 |
| | T0 | T1 | T2 | T3 | T4 | T5 |

FIGURE 2-2. CONCURRENTLY EXECUTING PROCESSES.

However, improved processor utilization is not the primary reason the Executive Run Time Support supports multitasking; the Executive Run Time Support 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 controls its behavior. This one-to-one correspondence between external activity and software control programs provides a powerful technique for the breakdown 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.

NOTE

In the terminology of Executive Run Time Support, a program is a special case of a process. A task is performed by a process, so "multiprocessing" would be a more appropriate term than "multitasking. However, "multiprocessing" has been used to describe systems utilizing multiple processors, so we will use the term "multitasking".

## 2.3 PROCESS

To permit multitasking, the concept of a process has been introduced into the Micprocessor Pascal System language. A process is a separately executable collection of instructions with data on which the instructions operate and with its own run time environment. Each process is independently scheduled for execution on a priority basis, and interacts with other processes and the executive as needed.

In a stand-alone environment having multiple processes (sites of execution), the ability to support priority scheduling and context switching are essential. In the example in Figure 2-2, a process is prepared for each of the four steps to be performed. Because the throughput of the system is limited to the speed of the slowest step, it would be desirable to give that step priority over any other step. In this way, it would never have to wait for the processor to be switched from one of the less urgent steps. The executive RTS provides for pre-emptive priority scheduling to serve this class of need, ensuring that the most urgent process that is ready to execute is the active process. A process priority is a user-assigned number that indicates the relative urgency of the process; the lower the number, the more urgent the process.

NOTE

The word "process" will be used many times in this document in a context applicable to a system, program, or process. Since 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 clearly made.

### 2.3.1 State Of The Process

When control is switched from a process, the current state of the process is saved. If the process were snapshot at some instant, its state would indicate the next instruction to be executed as well as the current values of all data variables which it can address. The state of the process must be resored before the process can resume its computation. The state of a process includes at least the machine context (workspace pointer, program counter, and status register) which is saved in a data structure called 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 a local memory region. Invariant code and relative data references provide reentrancy (See paragraph 2.7), and allows one copy of code to be in simultaneous use by more than one process.

### 2.3.2 Processes As Interrupt Handlers

Processes can also be created to act as hardware interrupt handlers. A process can be prepared to control each type of device in the system (e.g., an I/O device). When the device encounters an external condition that requires the attention of an internal process, it signals that fact by posting an interrupt request. Thus, the Executive Run Time Support allows the user to write processes in Pascal to service interrupts and devices

### 2.3.3 Interprocess Communication

Processes can communicate among themselves and synchronize with each other using the Executive Run Time Support. The system demonstrated in Figure 2-2 demonstrates this need. In the example, Step 2 must "WAIT" for Step 1 to "SIGNAL" completion of the filling operation before the capping operation can begin. Similarly, Step 2 must "SIGNAL" Step 3 that the checking operation is complete. This synchronization activity is supported in the executive RTS by the "Semaphore Management" set of procedures and functions. Interprocess files are also supported, which allow the sending and receiving of messages between processes. However, the implementation of these files is different in the interpretive mode and the native mode.

## 2.4 SOFTWARE ORGANIZATION OF MICROPROCESSOR PASCAL

The user application is built using a collection of processes nested at levels (referred to as "lexical levels") within the outermost Microprocessor Pascal construct which is the "System". Lexical levels indicate to the RTS the level at which a process in embedded in the System. Figure 2-3 illustrates this process organization.

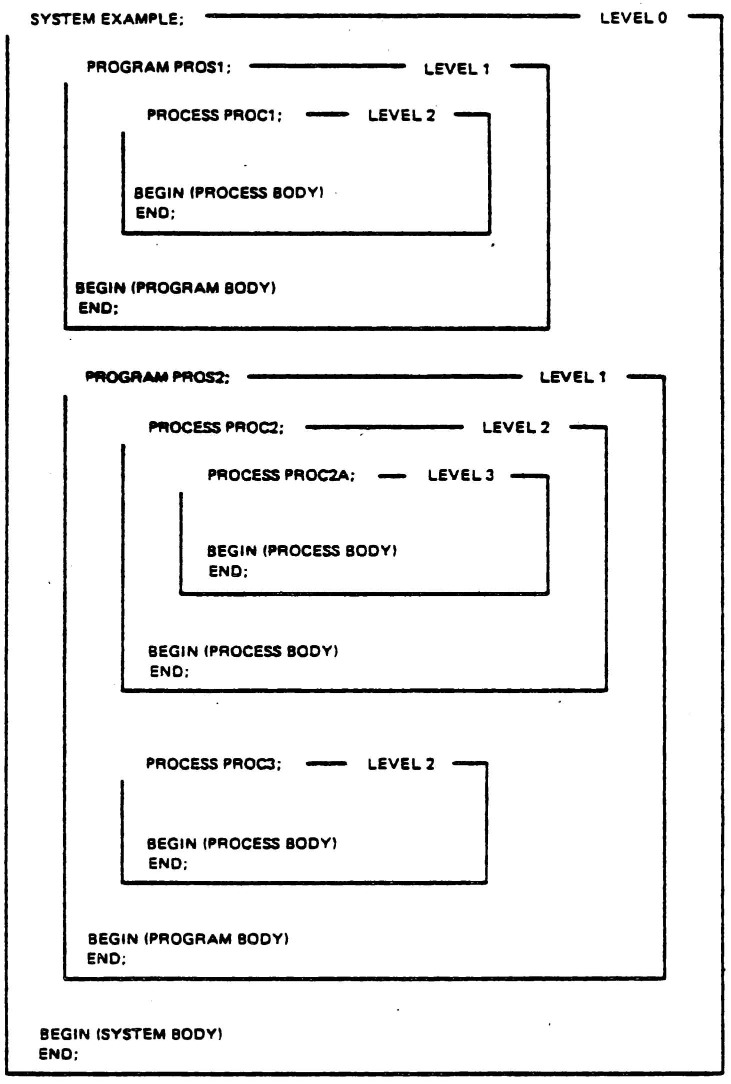


FIGURE 2-3. NESTING OF SYSTEM, PROGRAM, AND PROCESS DECLARATIONS.

As illustrated in Figure 2-3, the System is found at lexical level 0. Programs (special cases of processes) started (spawned) by the System reside at lexical level 1. Any processes spawned by programs at lexical level 1 reside at lexical level 2, etc. A Program always resides at lexical level 1; its parent is always the System. Processes are spawned by "Programs" and by other processes and thus processes reside at lexical level 2 or lower. (A process that spawns any other process is referred to as that process´ "lexical parent" or "ancestor". Conversly, a process spawned by another process is referred to as that process´ "child".) Processes are made up of standard Pascal procedures and functions.

MPP System architectural terminology is explained in paragraphs 2.4.1 to 2.4.3.2. Refer to Section 8 for a description of how to declare and implement each construct, using the Microprocessor Pascal System language.

### 2.4.1 SYSTEM

The SYSTEM is the outermost level of declarations and executable statements in a Microprocessor Pascal System; all other modules are contained within it (i.e., programs are nested within the SYSTEM, and processes are nested within programs or within other processes). As previously mentioned, Figure 2-3 illustrates this concept of nesting.

A SYSTEM is the process in which execution begins. The SYSTEM initializes global parameters and starts the programs which it contains. A SYSTEM must not have any variables with the possible exception of variables in COMMONs. (See paragraph 8.5.3.4)

### 2.4.2 PROGRAM

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

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 sequential Pascal which allows only single program environments.

## 2.4.3 PROCESS

A process is a specific entity that "owns" a set of resources and performs one or a series of computations.

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.

### 2.4.3.1 Procedure

A procedure is a statement or group of statements linked to a name. For example: A program consists of a main program labeled: PROGRAM_MAIN, and x number of subroutines (processes). Assume that PROGRAM_MAIN must search a list of values for the value 0. To accomplishg this, the program would contain the following process:

```
ZERO_SEARCH
  REPEAT
    READ (x)
  UNTIL x = 0
END
```

One need only to include a call to ZERO_SEARCH in PROGRAM_MAIN to accomplish the task.

### 2.4.3.2 Function

A function also links a name to a statement or group of statements; however, a function has a
value, while a procedure has an effect. A function is especially useful when a particular calculation is performed repeatedly using different data. For example:

```
PROGRAM_MAIN;
      .  .
      .  .
      .  .
   BEGIN
      .  .
      .  .
     SUMXY = Y + X
      .  .
      .  .
     SUMAB = A + B
      .  .
      .  .
   END
```

BECOMES:

```
          PROGRAM_MAIN;
               .  .
               .  .
               .  .
           FUNCTION SUM(I,J:INTEGER):INTEGER;
             BEGIN
               SUM: = I + J
             END
           BEGIN
               .  .
               .  .
             SUMXY: = SUM(X,Y);
               .  .
               .  .
             SUMAB: = SUM(A,B);
               .  .
               .  .
           END;
```

## 2.5 LANGUAGE EXTENSIONS TO SUPPORT PROCESSES

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 is a site of execution. 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 compiler.

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

- 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.

## 2.6 MEMORY

Each program or process has two concepts 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.

NOTE

> In order to use NEW and DISPOSE, a variable that will point to the variable to be created in the Heap must be declared in the Stack.

### 2.6.1 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.

### 2.6.2 Stack

A stack is implemented by using a block of storage called a stack region, out of which stack frames are allocated upon process entry and deallocated upon process 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, temporaries, and an administration area. See Figure 2-4.

```
+----------------+-----------+-------------+
| ADMINISTRATION | VARIABLES | TEMPORARIES |
+----------------+-----------+-------------+
```

Figure 2-4. TYPICAL STACK FRAME.

### 2.6.3 Heap

A heap is an area of memory which may be allocated in arbitrarily 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 program 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).

```
+--------------+--------+--------+--------+--------+-------+
|ADMINISTRATION| PACKET | UNUSED | PACKET | PACKET | UNUSED|
+--------------+--------+--------+--------+--------+-------+
```

FIGURE 2-5. HEAP STRUCTURE.

## 2.7 REENTRANCY

Reentrancy is a property of code (of which Microprocessor Pascal System code is an 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 physically using 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. As an example, this allows 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.

## 2.8 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), or when a calling routine calls another routine which in turn calls the first calling routine (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. A typical example 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 symbolically 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 3

# A SAMPLE MPP SYSTEM

## 3.1 OVERVIEW

Much attention has been given to methods for producing good quality software. The solution to this problem remains a highly subjective one, as any two software designers may use very different methods to achieve the same result, namely, a reliable, maintainable software product that performs the desired function. However, though ideas of method may differ, one point is constant: the software design process must become ever more disciplined.

Software systems must be simple, adaptable, and reliable if they are to achieve a long lifetime of use. The purpose of this section is to offer a sample of the disciplined software development cycle by presenting a simple software system in a step-by-step example which shows 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. Any 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 THE SAMPLE SYSTEM

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(Figure 3-1). 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, or if the buffer is not available for exclusive access. 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.

```
|----------|        |---------|        |-----------|
|PRODUCER  | <----> | BUFFER  | <----> | CONSUMER  |
|----------|        |---------|        |-----------|
```

FIGURE 3-1. DIAGRAM OF INPUT/OUTPUT.

In Figure 3-2 on the following pages, 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 (see Section 9 for additional information on semaphores). 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.

```
DX Microprocessor Pascal System Compiler    3.0  6/22/81 17:54:32  PAGE 1

   0
   0  {$DEBUG,MAP}
   0
   0 SYSTEM  TUTORIAL;
   0
   0   CONST
   0     Number_of_slots = 10; { Maximum number of slots in buffer }
   0
   0   TYPE
   0     Slot_index = 1..Number_of_slots;
   0     Alphabetic = 'A'..'Z';
   0     Buffer     = RECORD
   0                    Next_in           : Slot_index;
   0                    Next_out          : Slot_index;
   0                    Not_empty         : SEMAPHORE;
   0                    Not_full          : SEMAPHORE;
   0                    Exclusive_access  : SEMAPHORE;
   0                    Slots             : ARRAY [Slot_index] OF Alphabetic;
   0                  END;
   0
   0
   0
   0   COMMON
   0     Message_buffer : Buffer;
   0
   0
   0   ACCESS
   0     Message_buffer;
   0
   0
   0   PROCEDURE  INITSEMAPHORE (  VAR Sema  : SEMAPHORE;
   2                                   Count : INTEGER );          EXTERNAL;
   4
   0   PROCEDURE  SIGNAL        (      Sema  : SEMAPHORE );        EXTERNAL;
   2
   0   PROCEDURE  WAIT          (      Sema  : SEMAPHORE );        EXTERNAL;
   2
   0   PROCEDURE  SWAP;                                            EXTERNAL;
   0
   0
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUCER PROCESSES.
(1 of 5)

```
DX Microprocessor Pascal System Compiler   3.0   6/22/81 17:54:32  PAGE 2
   0 {$PAGE}
   0
   0   PROGRAM  PRODUCER;                                    { Produce messages }
   0
   0     VAR
   0       Item : Alphabetic;
   2       Line : PACKED ARRAY [1..16] OF CHAR;
  18
  18     ACCESS
  18       Message_buffer;
  18
   1     BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100  }
   1        { Initialize item so that first message will be ´A´ }
   1       Item := ´Z´;
   2        { Initialize message to inform user of "PRODUCTION" }
   2       Line := ´Item produced:  ´;
   3       WITH  M = Message_buffer  DO
   4         WHILE  TRUE  { i.e. do forever }
   5            DO  BEGIN
   5                 { Set item to be ´PRODUCED´ }
   5                IF  Item = ´Z´  THEN  Item := ´A´
   7                                ELSE  Item := SUCC ( Item );
   8                 { Wait on an empty buffer slot }
   8                WAIT ( M.Not_full );
   9                 { Wait on exclusive access to the message buffer }
   9                WAIT ( M.Exclusive_access );
  10                 { Move message to next available slot in buffer }
  10                M.Slots [ M.Next_in ] := Item;
  11                 { Set pointer to next free slot }
  11                M.Next_in := SUCC ( M.Next_in MOD Number_of_slots );
  12                 { MOD function produces a value 0..(Number_of_slots-1),
  12                   Ie. 0..9.  If the slot just used was 10 then MOD
  12                   will give 0, and SUCC(0) is 1, which is what we want }
  12                 { Relinquish exclusive access of message buffer }
  12                SIGNAL ( M.Exclusive_access );
  13                 { Signal that another message was ´PRODUCED´ }
  13                SIGNAL ( M.Not_empty );
  14                 { Set output message to indicate what was ´PRODUCED´ }
  14                Line[16] := Item;
  15                 { Output the message to the user }
  15                MESSAGE ( Line );
  16                 { Give other processes at this priority a
  16                   chance to execute }
  16                SWAP;
  17            END;
  17        { End of PRODUCER program }
  17     END;
  17
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES.
(2 of 5)

```
DX Microprocessor Pascal System Compiler   3.0   6/22/81 17:54:32  PAGE 3
 17 {$PAGE}
 17
  0   PROGRAM  CONSUMER;                                      { Consume messages }
  0
  0     VAR
  0       Item : Alphabetic;
  2       Line : PACKED ARRAY [1..36] OF CHAR;
 38
 38     ACCESS
 38       Message_buffer;
 38
  1     BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100  }
  1
  1        { Initialize message to inform user of "CONSUMPTION".
  1          NOTE: This message has 20 leading blanks to make it print out
  1          in a different column to the ´item produced´ messages. }
  1       Line := ´                    Item consumed:  ´;
  2       WITH  M = Message_buffer  DO
  3         WHILE  TRUE  { i.e. do forever }
  4            DO  BEGIN
  4                 { Wait on an full buffer slot }
  4                WAIT ( M.Not_empty );
  5                 { Wait on exclusive access to the message buffer }
  5                WAIT ( M.Exclusive_access );
  6                 { Get message from slot in buffer }
  6                Item := M.Slots [ M.Next_out ];
  7                 { Set pointer to next free slot }
  7                M.Next_out := SUCC ( M.Next_out MOD Number_of_slots );
  8                 { MOD function produces a value 0..(Number_of_slots-1),
  8                   Ie. 0..9.  If the slot just used was 10 then MOD
  8                   will give 0, and SUCC(0) is 1, which is what we want }
  8                 { Relinquish exclusive access of message buffer }
  8                SIGNAL ( M.Exclusive_access );
  9                 { Signal that another message was ´CONSUMED´ }
  9                SIGNAL ( M.Not_full );
 10                 { Set output message to indicate what was ´CONSUMED´ }
 10                Line[36] := Item;
 11                 { Output the message to the user }
 11                MESSAGE ( Line );
 12                 { Give other processes at this priority a
 12                   chance to execute }
 12                SWAP;
 13
 13           END;
 13
 13        { End of CONSUMER program }
 13     END;
 13
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES.
(3 of 5)

```
DX Microprocessor Pascal System Compiler   3.0   6/22/81 17:54:32  PAGE 4
 13 {$PAGE}
 13
 13
  1   BEGIN  {# STACKSIZE = 300;  HEAPSIZE = 500 }
  1          { Code for SYSTEM Tutorial }
  1
  1     { Initialize message buffer }
  1    WITH  M = Message_buffer
  2      DO  BEGIN
  2           { Initialize first in and first out to the same slot. So
  2             long as they are the same and within range any initial
  2             value will work }
  2          M.Next_in  := 1;
  3          M.Next_out := 1;
  4           { Initialize the exclusive access semaphore for only
  4             one access at a time }
  4          INITSEMAPHORE ( M.Exclusive_access, 1 );
  5           { Initialize the not empty semaphore for no messages
  5             currently in buffer }
  5          INITSEMAPHORE ( M.Not_empty, 0 );
  6           { Initialize the not full semaphore for all messages
  6             currently empty }
  6          INITSEMAPHORE ( M.Not_full, Number_of_slots );
  7      END;
  7
  7
  7     { Initialization complete. Start the producer and consumer }
  7    START  Producer;
  8    START  Consumer;
  9
  9   END.


SYSTEM TUTORIAL;
    STACK SIZE = 0000

          COMMON     TYPE      SIZE
          MESSAGE_   RECORD    30

          FIELD      DISP      TYPE      SIZE
          NEXT_IN    0000      SUBRANGE  2
          NEXT_OUT   0002      SUBRANGE  2
          NOT_EMPT   0004      SEMAPHORE 2
          NOT_FULL   0006      SEMAPHORE 2
          EXCLUSIV   0008      SEMAPHORE 2
          SLOTS      000A      ARRAY     20
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES.
(4 of 5)

```
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 = 0012

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

PROGRAM CONSUMER;
    STACK SIZE = 0026

          VARIABLE  DISP      TYPE      SIZE
          ITEM      0000      SUBRANGE  2
          LINE      0002      STRING    36
```

FIGURE 3-2. COMPILER LISTING OF PRODUCER/CONSUMER PROCESSES.
(5 of 5)

## 3.4 SAMPLE DEBUG SESSION

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

```
 HOST DEBUGGER            6/22/81  17:55:19

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

  System heap size = 5 (K)bytes                     } These lines appear in
  Most recently compiled system will be loaded.     } the "LOG" file only.

<>DEBUG(PRODUCER)

<>DEBUG(CONSUMER)

<>GO
     run-time support now initialized

<>GO
     *** Process Created *** PRODUC(2)

<>GO
     *** Process Created *** CONSUM(3)

<>SDP(PRODUCER)

<>AB(PRODUCER,12)

<>SDP(CONSUMER)

<>AB(CONSUMER,8)

<>SC(MESSAGE)
     common MESSAG
 E5C0 (0000) 0001 0001 F0A0 F096 F0AA 0000 0000 0000  (................)
 E5D0 (0010) 0000 0000 0000 0000 0000 0000 0000       (..............  )

<>GO
     *** Breakpoint ***       PRODUC(2).PRODUC    Statement 12

<>SC(MESSAGE)
     common MESSAG
 E5C0 (0000) 0002 0001 F0A0 F096 F0AA 0041 0000 0000  (...........A....)
 E5D0 (0010) 0000 0000 0000 0000 0000 0000 0000       (..............  )

<>GO
Item produced: A
     *** Breakpoint ***       CONSUM(3).CONSUM    Statement 8

<>SC(MESSAGE)
     common MESSAG
 E5C0 (0000) 0002 0002 F0A0 F096 F0AA 0041 0000 0000  (...........A....)
 E5D0 (0010) 0000 0000 0000 0000 0000 0000 0000       (..............  )

<>GO
                     Item consumed: A
     *** Breakpoint ***       PRODUC(2).PRODUC    Statement 12

<>SC(MESSAGE)
     common MESSAG
 E5C0  (0000) 0003 0002 F0A0 F096 F0AA 0041 0042 0000   (...........A.B..)
 E5D0  (0010) 0000 0000 0000 0000 0000 0000 0000        (..............  )

<>GO
Item produced: B
     *** Breakpoint ***       CONSUM(3).CONSUM    Statement 8

<>SC(MESSAGE)
     common MESSAG
 E5C0  (0000) 0003 0003 F0A0 F096 F0AA 0041 0042 0000   (...........A.B..)
 E5D0  (0010) 0000 0000 0000 0000 0000 0000 0000        (..............  )

<>GO
                    Item consumed: B
     *** Breakpoint ***       PRODUC(2).PRODUC    Statement 12

<>HP(CONSUMER)

<>DAB(PRODUCER)

<>GO
Item produced: C
Item produced: D
Item produced: E
Item produced: F
Item produced: G
Item produced: H
Item produced: I
Item produced: J
Item produced: K
Item produced: L
     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)
     common MESSAG
 E5C0  (0000) 0003 0003 F0A0 F096 F0AA 004B 004C 0043   (...........K.L.C)
 E5D0  (0010) 0044 0045 0046 0047 0048 0049 004A        (.D.E.F.G.H.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 8

<>SC(MESSAGE)
     common MESSAG
 E5C0 (0000) 0003 0004 F0A0 F096 F0AA 004B 004C 0043   (...........K.L.C)
 E5D0 (0010) 0044 0045 0046 0047 0048 0049 004A        (.D.E.F.G.H.I.J  )

<>GO
                         Item consumed: C
     *** Breakpoint ***       CONSUM(3).CONSUM    Statement 8

<>SC(MESSAGE)
     common MESSAG
 E5C0 (0000) 0003 0005 F0A0 F096 F0AA 004B 004C 0043   (...........K.L.C)
 E5D0 (0010) 0044 0045 0046 0047 0048 0049 004A        (.D.E.F.G.H.I.J  )

<>DAB(CONSUMER)

<>GO
                         Item consumed: D
                         Item consumed: E
                         Item consumed: F
                         Item consumed: G
                         Item consumed: H
                         Item consumed: I
                         Item consumed: J
                         Item consumed: K
                         Item consumed: L
     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)
```

```
     common MESSAG
 E5C0 (0000) 0003 0004 F0A0 F096 F0AA 004B 004C 0043   (...........K.L.C)
 E5D0 (0010) 0044 0045 0046 0047 0048 0049 004A        (.D.E.F.G.H.I.J  )

<>RP(PRODUCER)

<>GO
Item produced: M
                         Item consumed: M
Item produced: N
                         Item consumed: N
Item produced: O
                         Item consumed: O
Item produced: P
                         Item consumed: P
Item produced: Q
                         Item consumed: Q
Item produced: R
                         Item consumed: R
Item produced: S
                         Item consumed: S
Item produced: T
                         Item consumed: T
Item produced: U
                         Item consumed: U
Item produced: V
                         Item consumed: V
Item produced: W
                         Item consumed: W
Item produced: X
                         Item consumed: X
Item produced: Y
                         Item consumed: Y
Item produced: Z
                         Item consumed: Z
Item produced: A
                         Item consumed: A
Item produced: B
                         Item consumed: B
Item produced: C
                         Item consumed: C
Item produced: D
                         Item consumed: D
Item produced: E
                         Item consumed: E
Item produced: F
                         Item consumed: F
Item produced: G
                         Item consumed: G
Item produced: H
                         Item consumed: H
Item produced: I
                         Item consumed: I
Item produced: J
```

```
                    Item consumed: J
Item produced: L
                    Item consumed: L

                {....   NOTE. At this point the CMD key was struck   ....}
                {....   causing an anonymous breakpoint to occur.    ....}
                {....   The program would have executed forever if   ....}
                {....   we had not done this.                        ....}

     *** Anonymous Bkpt ***

<>QUIT

Execution Terminated
Memory Used (bytes)  Maximum = 4046  Current = 2148
```

In the sample debugging session, the message buffer is displayed at various points using a "Show Common" (SC) command. Notice that a length of 30 (>1E) bytes starting at displacement 0 is specified each time the common is displayed. From the compiler map, it can be seen that the last 20 (>14) bytes of the Common comprise the message buffer slots. The first four bytes of the Common form "next_in" and the "next_out" fields. 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., an item is generally consumed as soon as it is produced. To make the example more interesting, the consumer is held using the "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, so the The session was 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 4

## THE MPP SOURCE EDITOR

### 4.1 OVERVIEW

The Source Editor allows the user to create new source files and to modify existing source files which may be input to the Microprocessor Pascal Compiler. The Source Editor can be invoked and operated from a Texas Instruments 911, 913, or TMAM 9000 Video Display Terminal (VDT). Information on operating these terminals can be found in Volume I of the Model 990 Computer DX10 Operating System Reference Manual (part number 946250-9701) for the 911 and 913 VDTs, and in the AMPLUS Software System User´s Manual (part number 1603142-9701) for the TMAM 9000 VDT.

In this section, only the 911 terminal commands and responses will be addressed, although the function response to the correct command input on the 913 and the TMAM 9000 will be the same. See Table 4-2 for equivalent command inputs on these two VDT´s.

#### 4.1.1 The Video Display

Editing occurs on a page basis; a page is 23 lines on the 911. Any line displayed on the screen may be edited by positioning the cursor at any character on 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 the file for display is accomplished by the use of the Roll Up, Roll Down, Cursor Up, and Cursor Down functions as well as by the Relative Positioning, Top, Bottom, and Find commands.

#### 4.1.2 Microprocessor Pascal Source File Definition

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

- o A module header with any portion of its declaration section and its associated body,
- o 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

Table 4-1 is a summary of the editor commands and functions as input from a 911 VDT. Table 4-2 is a summary of editor commands and functions as input from 913 and TMAm 9000 VDTs. A detailed description of each command/function is given in Section 4.3. The commands and functions of the Source Editor are divided into five separate classes for your convenience.

TABLE 4-1. MPP SOURCE EDITOR COMMANDS AND FUNCTIONS: 911 VDT.

| COMMAND/FUNCTION | 911 VDT |
|---|---|
| <u>Setup and Termination</u> | |
| Help | CMD/"HELP" |
| Edit/Compose Toggle | F7 |
| Syntax Check | CMD/"CHECK" |
| Quit | CMD/"QUIT" |
| Abort | CMD/"ABORT" |
| Save | CMD/"SAVE" |
| Input | CMD/"INPUT" |
| <u>Cursor Positioning</u> | |
| Roll Up | F1 |
| Roll Down | F2 |
| New Line | RETURN Key |
| Tab | SHIFT/TAB Key |
| Back Tab | FIELD Key |
| Set Tab | CMD/TAB <incr> |
| Cursor Up | Up Arrow |
| Cursor Down | Down Arrow |
| Cursor Right | Right Arrow |
| Cursor Left | Left Arrow |
| Home (Return to beg. of line) | HOME Key |
| Find | CMD/"FIND" <parm> |
| Relative Positioning | CMD/number |
| Top | CMD/"TOP" |
| Bottom | CMD/"BOTTOM" |
| <u>Program Modification</u> | |
| Insert Line | Blank Gray Key |
| Duplicate Line | F4 |
| Delete Line | ERASE INPUT |
| Skip | TAB/Skip Key |
| Insert Character | INS CHAR Key |
| Delete Character | DEL CHAR Key |
| Clear Line | ERASE FIELD Key |
| Replace | CMD/"REPLACE"<parm> |
| Split Line | F8 |
| Insert | CMD/"INSERT" |
| <u>Block Commands</u> | |
| Start Block | F5 |
| End Block | F6 |
| Copy | CMD/"COPY" |
| Move | CMD/"MOVE" |
| Delete | CMD/"DELETE" |
| Put | CMD/"PUT" |
| <u>Show Command</u> | |
| Show | CMD/"SHOW" |

TABLE 4-2. MPP SOURCE EDITOR COMMANDS/FUNCTIONS: 913 & TMAM 9000 VDT

| COMMAND/FUNCTION | 913 VDT | TMAM 9000 VDT |
|---|---|---|
| Setup and Termination | | |
| Help | HELP/"HELP" | F9/"HELP" |
| Edit/Compose Toggle | F7 | F7 |
| Syntax Check | HELP/"CHECK" | F9/"CHECK" |
| Quit | HELP/"QUIT" | F9/"QUIT" |
| Abort | HELP/"ABORT" | F9/"ABORT" |
| Save | HELP/"SAVE" | F9/"SAVE" |
| Input | HELP/"INPUT" | F9/"INPUT" |
| Cursor Positioning | | |
| Roll Up | ROLL UP | F1 |
| Roll Down | ROLL DOWN | F2 |
| New Line | NEW LINE | RETURN |
| Tab | TAB | SHIFT TAB |
| Back Tab | BACK TAB | BACK TAB |
| Set Tab | HELP/"TAB" <incr> | F9/"TAB" <incr> |
| 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 | HELP/"FIND" parm> | F9/"FIND" <parm> |
| Relative Positioning | HELP/number | F9/number |
| Top | HELP/"TOP" | F9/" OP" |
| Bottom | HELP/"BOTTOM" | F9/"BOTOM" |
| Program Modification | | |
| Insert Line | INSERT LINE | CONTROL O |
| Duplicate Line | F4 | F4 |
| Delete Line | DELETE LINE | ERASE INPUT |
| Skip | SKIP | TAB SKIP |
| Insert Character | INS CHAR | INS CHAR |
| Delete Character | DEL CHAR | DEL CHAR |
| Clear Line | CLEAR | ERASE FIELD |
| Replace | CMD/"REPLACE"<parm> | F90/"REPLACE" <parm> |
| Split Line | F0 | F8 |
| Insert | HELP/"INSERT" | F9/"INSERT" |
| Block Commands | | |
| Start Block | F5 | F5 |
| End Block | F6 | F6 |
| Copy | HELP/"COPY" | F9/"COPY" |
| Move | HELP/"MOVE" | F9/"MOVE" |
| Delete | HELP/"DELETE" | F9/"DELETE" |
| Put | HELP/"PUT" | F9/"PUT" |
| Show Command | | |
| Show | HELP/"SHOW" | F9/"SHOW" |

NOTE

Lower-case letters and some special symbols (braces) cannot be represented on the TI 913 VDT. In a file created on a 911 (or TMAM 9000) but displayed on a 913, lower-case letters appear as upper-case and unrepresentable symbols are replaced by other (displayable) symbols. Editing this same file on a 913 results in the altered positions of text containing only those characters that are supported on the 913 (upper-case letters, symbols, etc.)

---

## 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.

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 intended to be part of the source program in columns 73 through 80. Furthermore, if a file that has data in columns 73 through 80 in edited using this editor, the information in those columns will be lost in all lines that are modified during an edit session.

Unless specifically requested,the compiler will ignore everything in columns 73 through 80.

### 4.2.1 Creating a File

The following procedure applies to the creation of a new file using the Source Editor.

To invoke the Source Editor, the user enters the command EDIT after the Microprocessor Pascal System has been loaded (see Host Systems user´s manual for information on loading Microprocessor Pascal under the DX/10, TX/4 and AMPLUS operating systems respectively.)

In response to the EDIT command, the following prompt is displayed:

```
INPUT FILE ACCESS NAME:
```

If a previously edited file name is displayed, clear the field with the TAB/SKIP key and press RETURN. This indicates that a new source file is to be created. The screen is cleared, The cusor and *EOF is displayed in the upper left-hand corner of the screen, and COMPOSE MODE is displayed in the lower right-hand corner of the screen, The cursor position is at line 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 the RETURN key. Note that line one is now a blank line, the end-of-file marker is on line two, and the cursor is positioned at line one, column one. You may now begin entering a source file by simply typing the data and pressing the RETURN key whenever you wish to enter another line.

## 4.2.2 Editing An Existing File

Invoke the Source Editor by entering the EDIT command and the name of file to be edited after the INPUT FILE ACCESS NAME: prompt. The file will be displayed, beginning at the top of the file.

Typographical errors corrections, additions and deletions may be made by using the edit key and commands as described in section 4.3.

Following is a sample of how some of the editing techniques are used. For this example session, the file in Figure 4-1 will be used as the input file.

Upon invocation, the following is displayed:

```
LINE                    (Cursor Position)

1                        []
2                        PROGRAM EXAMPLE;
3                        VAR <-------------------------- (4) Add SECOND
4                          FIRST: INTEGER;
5                        RESET(INPUT);
6      (1)Insert     ---->WHILE NOT EOF DO
7         BEGIN     /            READLN(FIRST,SECOND);
8                                IF FIRST = SECOND
9                                THEN WRITELN('EQUAL',FIRST);<--(2) Delete
10                               ELSE WRITELN(FIRST,'NOT EQUAL',SECOND)
11                             END;
12     (3) Delete  -------> END.
13                     *EOF
```

FIGURE 4-1. INPUT FILE EXAMPLE CONTAINING ERRORS.

The screen is cleared and the file is displayed, beginning in line one of the screen, EDIT MODE is displayed in the lower right hand corner, and *EOF is displayed on line 13, after the last line of the source file. The cursor is in column one, line one.

#### 4.2.2.1 Correcting The Errors

Refer to Figure 4-1.

(1) Upon examination of the program to be edited, notice that in the WHILE statement, the keyword BEGIN, which should follow on the next line, has been omitted. In order to insert BEGIN, press the down-arrow key seven times, then press the blank grey (INSERT LINE) key, which inserts a blank line between lines 6 and 7. The cursor is in column one of the blank line. The user then types BEGIN in the proper character spaces.

At this point the user may think that the program is syntactically complete. To verify this, press the CMD key, enter the word: CHECK, and press the RETURN key.

(2) 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" (line 7). Press the up-arrow key, the HOME key, then the FIELD key to position the cursor on the semicolon following the THEN clause. (The same result may be achieved by pressing the up-arrow key, then using the right-arrow key to position the cursor.) Remove this semicolon, by pressing the space bar, causing the semicolon to be replaced by a blank.

Again, verify that the program is syntactically correct by pressing the CMD key, and the RETURN key in response to the CHECK which is already present at the bottom of the screen (from the original entry).

(3) The error message:

"MODULE EXPECTED"

is displayed on the bottom line of the screen and the cursor is positioned at the keyword "END" immediately preceeding the *EOF. Press the ERASE INPUT key to delete the line containing the extra "END;" statement, then again check for syntactical correctness by pressing the CMD key and the RETURN key in response to the CHECK already present at the bottom of the screen.

The message:

"NO SYNTAX ERRORS FOUND"

is displayed on the bottom line of the screen. The user then saves the file as described in Paragraph 4.2.3.

(4) Notice that the syntax checker did not detect that the variable "SECOND" was not declared. This is an example of the kind of a semantic error which is not detected by the syntax checker; they are the responsibility of the programmer, but will be identified by the compiler if overlooked.

### 4.2.3 Saving The File

After a file has been created or an existing file has been edited, the file may be saved by one of two methods:

#### 4.2.3.1 To Quit Editing Operations

If the user does not want to edit another source file, Press the CMD key and type in the word: QUIT. The user will then be prompted as follows:

OUTPUT FILE ACCESS NAME:

Respond by typing the pathname of the file to which the edited source file is to be written and press the RETURN key. The user will then be prompted as follows:

REPLACE?:

Respond by typing in the letter "Y" to specify that the data is to be placed in the file specified by the pathname entered as the output file. The editor is then exited.

If the user responds with an 'N' and a carriage return, he specifies that the existing file entered in response to the OUTPUT FILE ACCESS NAME prompt should not be replaced or have the editing changes incorporated. In order to finish the QUIT procedure, the user must either:

(1) Enter a new output file access name where the file and the editing will be saved. (The original input file will remain intact, unedited), or,

(2) End the procedure with an ABORT command. In this case, the file will remain under the original input file access name, without the editing just done.

If either an invalid pathname is given or an invalid replacement option is specified, a file I/O error will occur and an appropriate error message will be generated. If no pathname is specified for the output file and the RETURN key is pressed, no action is performed and the editor prompts for another command.

#### 4.2.3.2 <u>To Coninue Editing Operations</u>

If the user wants to edit another file after saving the file just edited, press the CMD key and type in the word: SAVE. 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 key. The user will then be prompted as follows:

REPLACE?:

If the user responds with an ´N´ and a carriage return, he specifies that the existing file entered in response to the OUTPUT FILE ACCESS NAME prompt should not be replaced or have the editing changes incorporated. In order to finish the QUIT procedure, the user must either:

(1) Enter a new input file access name where the file and the editing will be saved. (The original input file will remain intact, unedited), or,

(2) End the procedure with an ABORT command. In this case, the file will remain under the original input file access name, without the editing just done.

If the user responds with a ´Y´, the updated file is then saved in the file specified by the pathname entered in the OUTPUT FILE ACCESS NAME prompt, and the user is then prompted for the pathname of the next file to be edited as follows:

INPUT FILE ACCESS NAME: (file name of the previously edited file)

In response, the user may:

(1) Press the RETURN to re-edit the file just just saved,

(2) Press TAB/SKIP to clear the file pathname on the screen, and press RETURN to create a new file,

(3) Enter the pathname of another existing file to be edited.

If either an invalid pathname is given or an invalid replacement option be specified,
a file I/O error
will occur and an appropriate error message will be generated.
If no pathname is specified for the output file
when the RETURN key is pressed,
no action is performed and the editor prompts for another command.

## 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 these groups:

1) Setup and Termination

2) Cursor Positioning

3) Program Modification

4) Block Commands

6) Show Command

### 4.3.1 Parameters

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

- o Integer Constant - An integer constant parameter is a non-negative number less than or equal to 32767
- o Identifier - An identifier parameter is either a Microprocessor Pascal identifier or Microprocessor Pascal System reserved word
- o Pathname - A pathname parameter is a valid DX10, TX4, or AMPLUS pathname, depending on which operating system is being used.
- o String - A string parameter is a character string enclosed in double quotes ( a double quote is represented by two double quotes inside a string)

```
|------------STRING---------------|
|                                 |
V                                 V
("xxxxxx","xxx","xxxxxxxx","xx",etc)

     ^      ^        ^       ^
     |______|________|_______|
         |STRING PARAMETERS|
         |-----------------|
```

FIGURE 4-2. A STRING.

### 4.3.2 Optional Parameters

If a parameter is optional, it can simply be omitted, allowing 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 key is pressed during an edit session by placing a "+------+" in columns 73 through 80.

If the current line already contains a start or end block marker in columns 73 through 80 a "+------>" or a "<------+" will result in columns 73 through 80 when the CMD (HELP) key is pressed. If both a start and end block already appear on the current cursor line, no change will occur in columns 73 - 80.

### 4.3.4 CMD Key

The CMD 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 key will cause the editor to return to command mode. However, if the user presses the CMD key in response 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 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. 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 key. When operating in edit mode, pressing RETURN causes the cursor to move to the next line; in compose mode, pressing RETURN causes a blank line to be inserted after the current line and the cursor to be positioned on the new line on the first non-blank character of the previous line (column 1, if the previous line is blank).

When creating a new file the editor is invoked in compose mode. The user may switch back and forth between edit and compose modes by pressing F7, which acts as a toggle switch. The mode of the editor is displayed in the lower right-hand corner of the screen.

4.3.5.2 HELP Command. Pressing CMD and typing "HELP" causes the display of a list of available commands, along with short descriptions of each. The HELP command also displays the tab increment amount.

4.3.5.3 CHECK (Syntax) Command. Pressing CMD and typing "CHECK" instructs the editor to perform a syntax check of the module being edited and allows the user to correct errors as they are discovered. This feature has the advantage of helping the user detect common syntax errors before a (possibly) time-consuming compile is attempted. This command is typically used 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, allowing the user to correct it.

NOTE: syntax checking halts on finding the first syntax error. When the error is corrected, the CHECK command should be issued again to check for other errors. (For example, see 4.2.2.1)

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 and typing in "QUIT". The following prompt is then displayed:

```
    OUTPUT FILE ACCESS NAME: Pathname
```

Respond by typing the pathname of the file to which the edited source file is to be written and press the RETURN key. The user will then be prompted as follows:

```
                              REPLACE?:
```

Respond by typing in the letter "Y" to specify that the data is to be placed in the file specified by the pathname entered as the output file. The editor is then exited.

If the user responds with an ´N´ and a carriage return, he specifies that the existing file entered in response to the OUTPUT FILE ACCESS NAME prompt should not be replaced or have the editing changes incorporated. In order to finish the QUIT procedure, the user must either:

(1) Enter a new output file access name where the file and the editing will be saved. (The original input file will remain intact, unedited), or,

(2) End the procedure with an ABORT command. In this case, the file will remain under the original input file access name, without the editing just done.

If either an invalid pathname is given or an invalid replacement option be specified, a file I/O error will occur and an appropriate error message will be generated. If no pathname is specified for the output file when the RETURN key is pressed, no action is performed and the editor prompts for another command.

4.3.5.5 <u>ABORT Command</u>. The ABORT command is used to exit the editor without saving the results of the current edit session. This command is entered by pressing the CMD key and typing the word "ABORT".

4.3.5.6 <u>SAVE Command</u>. The SAVE command is used to save the results of the most recent edit session and begin the editing of a new file. This command is entered by pressing the CMD key and typing "SAVE". The following prompt is then displayed:

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 key. The user will then be prompted as follows:

REPLACE?:

If the user responds with an ´N´ and a carriage return, he specifies that the existing file entered in response to the OUTPUT FILE ACCESS NAME prompt should not be replaced or have the editing changes incorporated. In order to finish the QUIT procedure, the user must either:

(1) Enter a new input file access name where the file and the editing will be saved. (The original input file will remain intact, unedited), or,

(2) End the procedure with an ABORT command. In this case, the

file will remain under the original input file access name, without the editing just done.

If the user responds with a 'Y', the updated file is then saved in the file specified by the pathname entered in the OUTPUT FILE ACCESS NAME prompt, and the user is then prompted for the pathname of the next file to be edited as follows:

INPUT FILE ACCESS NAME: (name of the previously edited file)

In response, the user may:

(1) Press the RETURN to re-edit the file just just saved,

(2) Press TAB SKIP to clear the file pathname on the screen, and press RETURN to create a new file,

(3) Enter the pathname of an existing file to be edited.

If either an invalid pathname is given or an invalid replacement option be specified, a file I/O error will occur and an appropriate error message will be generated. If no pathname is specified for the output file when the RETURN key is pressed, no action is performed and the editor prompts for another command. If a legal pathnamed is supplied and a valid replacement option is specified, the file will be saved in its final, edited form and The following message will be displayed:

SAVE COMMAND EXECUTING

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

INPUT FILE ACCESS NAME: (name of file previously edited)

The response to the prompt is the pathname of the file which is to be edited next. If no pathname is specified (by clearing the field with the TAB/Skip key, then pressing the RETURN key, a new file will be created.

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, typing "INPUT", and pressing the RETURN key. the following prompt is displayed:

INPUT FILE ACCESS NAME: (name of file previously edited)

The response to the prompt is to type the pathname of the next file to be edited. If no pathname is specified when the RETURN key is presssed (by clearing the pathname field with the TAB/SKIP key), a new file is created. The cursor is positioned at the first column of the first line of the file.

The difference between the use of INPUT and ABORT (described in 4.3.5.5) is that INPUT prompts for an input file name after terminating the editing of the current edit; ABORT exits the source editor.

## 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 <u>Roll-Up Function:</u> The Roll-Up function is called by pressing the F1 key. This function advances the file 23 lines from its present location. The cursor remains positioned at the same character of the new line as it was on the previous line.

4.3.6.2 <u>Roll-Down Function:</u> The Roll-Down function is called by pressing the F2 key. This function reverses the file 23 lines from its present location. The cursor remains positioned at the same character of the new line as it was on the previous line.

4.3.6.3 <u>New Line Function:</u> The New Line function is called by pressing the RETURN 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 (blank) line. The cursor is positioned on a new line at the same indentation level as the first token on the previous line. If the code to be entered should start at a different nesting level, the TAB, Field (backtab), Space Bar, or right and left arrow keys can be used to move the cursor to the proper place.

4.3.6.4 <u>Tab Function:</u> The Tab function is called by simultaneously pressing the SHIFT key and TAB/SKIP key. If the cursor is positioned on a blank line, the cursor moves one indentation level to the right. Otherwise, the cursor moves to the start of the next token. If the cursor is at the last token on a line, the cursor moves one indentation level to the right. If the cursor is at column 72, the cursor is positioned at the beginning of the line.

4.3.6.5 <u>Back Tab Function:</u> The Back Tab function is called by pressing the FIELD key. If there are no characters to the left of the cursor, the cursor moves 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 set or change the increment amount used for tabs and back tabs. The syntax of this command is:

TAB(Increment)

NOTE: 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 grey 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 grey 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 grey right-arrow key. This function causes the cursor to be moved one position to the right. If the cursor is in column 72 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 grey left-arrow key. This function causes the cursor to be moved one position to the left. If the cursor is in column one 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 following the current cursor position. The command is entered by pressing the CMD 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). The search begins at the first character following the cursor. If the specified number of occurrences is found, the cursor is positioned so that it is on the first character of the last occurrence. The line in which the identifier or string is found occupies the middle row of the screen. If the specified number is not found, the cursor position remains unchanged; and a message:

n OCCURRENCE(S) NOT FOUND

indicates that the identifier or string was not found the specified number of times.

4.3.6.13 Relative Positioning: A file may be advanced or reversed by an arbitrary number of lines relative to the current file position on display. This is accomplished by first pressing the CMD key followed by the number (integer) 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 skip will stop at the file's beginning or end, depending on the direction of the jump, i.e., the cursor is positioned at column one at the top of the file, or, if the line is the end-of-file, the cursor is displayed as the last line on the screen. Otherwise, the cursor line is displayed as the middle line 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 in the file being edited. The command is entered by pressing CMD, typing "TOP", and pressing the RETURN key.

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, typing "BOTTOM", and pressing the RETURN key. The cursor is positioned at column one of the end-of-file marker.

### 4.3.7 Program Modification

The functions and commands described in this section are used to modify source files. When a line is modified, any data which may be in columns 73 through 80 are 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 (INSERT LINE) key. When this function is called, a blank line is inserted immediately before the line on which the cursor is presently located. The cursor is placed on the new line and remains in the same column.

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 on which the cursor is currently positioned to a new line immediately following the current line. The cursor is moved to the new line and remains in the same column.

4.3.7.3 Clear Line Function: The Clear Line function is called by pressing the ERASE FIELD key. This function causes the line containing the cursor to be cleared and repositions the cursor at the beginning of the line.

4.3.7.4 Delete Line Function: The Delete Line function is called by pressing the ERASE INPUT 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 on the line following the deleted line.

4.3.7.5 Skip Function: The Skip function is called by pressing the TAB/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.

4.3.7.6 Insert Character Function: The Insert Character function is called by pressing the INS CHAR key and typing the new character(s) to be inserted into the file. Characters are never lost at the right margin; therefore, if a non-blank character is present at the right margin, no additional characters can be inserted on the line and the beeper is sounded if this is attempted. In this case, the "split line" command must be used for breaking up long lines. (See 4.3.7.9)

4.3.7.7 Delete Character Function: The Delete Character function occurs each time the DEL CHAR key is depressed. This causes the character in the current cursor position to be deleted. Characters to the right of the cursor position are shifted one character position to the left and a blank is inserted in column 72.

4.3.7.8 REPLACE Command: The REPLACE command searches for the next "n" occurrences of an identifier (or string) and replaces each occurence with the identifier (or string) specified. The syntax of this command is:

```
REPLACE (<patternl>, <pattern2>, <number of occurences>)
```

where <patternl> is to be replaced by <pattern2> and both may be identifiers or strings enclosed in double quotes.

If the command is executed the number of times specified by <repeat count> without an error, the cursor is positioned at the first character of the last occurrence of <pattern2>. If <patternl> is not found the specified number of times, the cursor is returned to its position prior to execution of the command; a message:

```
n OCCURRENCE(S) NOT FOUND
```

is displayed, indicating the number of occurrences of <patternl> that were not replaced. If the replacing of an occurrence of <patternl> by <pattern2> results in characters being lost (pushed off the end of a line), the command is halted and the cursor is positioned at the beginning of the occurance causing the halt.

4.3.7.9 Split Line Function: The Split Line function is called by pressing the F8 key. This function causes the current line to be split into two lines such that the cursor position indicates the first character position 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 that was split. The cursor position is not changed.

4.3.7.10 <u>INSERT Command:</u> 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. The INSERT command is called by pressing CMD, typing "INSERT", and pressing the RETURN key. 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 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 manipulating designated blocks of lines instead of single characters or single lines.

4.3.8.1 <u>Start and End Block Functions:</u> These functions place markers to bracket file sections to be manipulated by the COPY, MOVE, DELETE, and PUT commands.

A start block marker is set by pressing the F5 key. Result is a beep and the placement of "<-------" in columns 73 through 80.

An end block marker is set by pressing the F6 key. Result is a beep and the placement of "------->" in columns 73 through 80.

If both the start and end block markers are set on the same line, (if you wish to move only one line of text, for example), "<------>" is placed in columns 73 through 80 of that line.

For both functions, the cursor position is used as the location of the marker.

4.3.8.2 <u>COPY Command:</u> The COPY command causes a copy of the block designated by the start and end block markers to be inserted between the line on which cursor is positioned and the following line. 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 marker does not exist, a message is displayed and no action is taken. To resume EDIT, the user must hit the CMD key and enter any command including a blank command.

4.3.8.3 <u>MOVE Command:</u> The MOVE command causes the block designated by the start and end block markers to be moved to the line following the line on which the cursor is positioned. The order of procedure is:

(1) Position the cursor at the beginning of the first line of the block to be moved. Press F5 (Start-block arrow set).

(2) Position the cursor at the beginning of the last line of the block to be moved. Press F6 (End-block arrow set).

(3) Position the cursor at the line preceeding the line where the block is to be inserted.

(4) Press CMD key and enter: MOVE (Ret)

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 cannot be moved to a location that is contained within itself.
If the end block marker precedes the start block marker, or either markers does not exist, a message is displayed and no action is taken. To resume EDIT, the user must hit the CMD key and enter any command including a blank command.

4.3.8.4 <u>DELETE Command:</u> 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. To resume EDIT, the user must hit the CMD key and enter any command including a blank command.

4.3.8.5 <u>PUT Command:</u> The PUT command causes a copy of the block designated by the start and end block markers to be copied to another file specified. After the PUT command has been entered, the user will be prompted for the pathname of the destination file.

```
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 key. (If no pathname is specified and the RETURN key is pressed, no action is performed and the editor prompts for another command.) The user will then be prompted as follows:

```
REPLACE?:
```

If the user responds by typing in letter N to specify that an existing file with the entered pathname should not be replaced, no action is taken. The message:

```
FILE EXISTS AND NO REPLACEMENT IS SPECIFIED
```

is displayed. the RETURN or COMD key must be pressed and another pathname or a Y response to the REPLACE? prompt entered in order to complete the PUT procedure. If the file does not exist, the file will be created and the data placed in it.

If the letter Y is entered, the requested replacement takes place. NOTE: the old data in the specified file will be lost. To save the data in the specified output access file, use the INSERT command instead of PUT. (See 4.3.7.10)

To resume the EDIT, enter any command including the blank command. If a legal pathname is given and the 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 an invalid pathname or an invalid replace response be incorrect, a file I/O error will occur and an appropriate error message will be displayed.

### 4.3.9 SHOW Command

The SHOW command causes the display of a file other than the one being edited during the edit session. After the SHOW command is entered, this prompt will be displayed asking for the pathname of the file to be shown:

```
SHOW FILE ACCESS NAME:
```

Respond by entering the pathname of the file to be displayed. Paging through the file may be accomplished by use of the F1 and F2 keys, or by using the relative positioning function (see 4.3.6.13). Pressing the CMD key terminates the show file and returns the file being edited to the display.

## 4.4 ERROR MESSAGES

The error messages generated by the Command Processor and Syntax Checker are described in the following paragraphs.

### 4.4.1 Command Syntax Errors

When a command or its parameter are improperly formed or recognized by the Source Editor, one of the following error messages is returned:

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.

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

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 during execution of an EDIT command:

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
Replacement of a string or name in a line would cause characters to be lost off the right hand side of a line.

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 MOVE
The designated block in a MOVE command cannot be moved to a location within itself.

### 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, SHOW, QUIT, INSERT, or PUT 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 the User´s Manuals associated with the relavent system.

BAD DISK NAME/DISK VOLUME NOT INSTALLED
: 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
: The file access name given by the user in response to an editor prompt does not exist.

FILE EXISTS AND REPLACE NOT SPECIFIED
: The user requests that a file be saved but not replaced, and a file of that name already exists, or replace was specified and the file is write or delete protected.

BAD PATHNAME SYNTAX
: The syntax of the file access name entered is invalid.

UNABLE TO GRANT REQUESTED ACCESS PRIVILEGES
: The user has requested a file in response to a prompt which can not be accessed by the editor, i.e., it is already in use.

### 4.4.4 Syntax Checking Error Messages

The CHECK routine begins at the first line of the file and checks each line in sequence. The cursor will stop at the first line where a syntax error is detected and an error message will display. Once corrective action is taken and the CMD RETURN keys are pressed, the routine will continue checking the file until another error is detected or EOF is reached.

The following error messages that may be generated by the syntax checking routine of the editor are:

1 STATEMENT SEPARATOR EXPECTED
: Statements must be separated by ";", "END", "ELSE", "OTHERWISE", 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 BEGIN 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 "," 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 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 is syntactically incorrect.

92 END OF STRING EXPECTED
AN "'" was expected to terminate a string within the 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 that is not allowed in the type of number found. It may 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 file using the CHECK command. If the problem persists, contact your Texas Instruments service representative.

# SECTION 5

# COMPILER AND NATIVE CODE GENERATOR

## 5.1 OVERVIEW

The Microprocessor Pascal System produces both interpretive code and native code. The Microprocessor Pascal Compiler takes the source code for a Microprocessor Pascal System as input and produces interpretive code for a hypothetical stack computer. The compiler checks for syntax and semantic errors. The error messages generated are listed and defined in Paragraph 5.2.6.1.

The Native Code Generator generates 9900 native code from the interpretive code produced by the compiler. Although native code is larger than interpretive code, certain types of applications may require either faster execution speed or critical timing dependencies which cannot be met by interpretive code.

Descriptions of the compiler and the native code generator are presented below.

## 5.2 COMPILER

Presented below is information concerning compiler printouts, compiler options, use of the COPY statement, separate compilations, and segment saving.

### 5.2.1 Compiler Printouts

The compiler printouts below are execution messages, a compiler listing, and a variable map.

5.2.1.1 Compiler Execution Messages. As the compiler executes, messages are output indicating how much of the system has been compiled. These messages are output to a file usually sent to the user´s display. For Example:

```
EXECUTION BEGINS
SCANNER IS FINISHED
FACTORIA
EXAMPLE1
NO ERRORS IN COMPILATION
Stack used =   2706  Heap used =  2672
Execution Ends.
```

```
     COMPILATION COMPLETE
```

The first line appears when execution begins. The last three lines indicate normal termination of the compiler and the amount of memory (in bytes) used in the compilation. The remaining lines in the message are generated by the compiler. An explanation of these lines follows:

The second line in the message appears when the first pass of the compiler is completed. (The compiler executes in two passes - scanner and parser). The third and fourth lines contain the names (first eight characters only) of modules in the system. The name of the module is output as its parsing is completed by the compiler.

The next line indicates no errors were discovered in the compilation.

When errors are found in the compilation, the message file is displayed as follows:

```
     Execution Begins
     SCANNER IS FINISHED
     ERRORS IN MODULE
     ERROR
     ERRORS IN MODULE
     EXAMPLE2
     FATAL ERRORS IN COMPILATION
     Stack used =   2532  Heap used =  2722
     Execution ENDS.
     ERRORS IN COMPILATION.
```

For each module containing errors, an "ERRORS IN MODULE" line is output before the module´s name. The message "FATAL ERRORS IN THE COMPILATION" is output following the name of the last module in the system and the last line reads "ERRORS IN COMPILATION". If all the errors found were non-fatal, the message "ERRORS IN COMPILATION" is output and the last line reads "COMPILATION COMPLETE".

5.2.1.2 <u>Compiler Listing</u>. The compiler listing is produced by the second pass of the compiler. An example of this listing is shown as follows:

```
DX  Microprocessor Pascal System Compiler   3.0  06/22/81 10:10:38  PAGE
    0 PROGRAM EXAMPLE1;
    0 VAR
    0   N: INTEGER;
    2   M: INTEGER;
    4
    0      FUNCTION FACTORIAL(I:INTEGER): LONGINT;
    1      BEGIN                                              { FACTORIA }
    1        IF I = 1
    2        THEN FACTORIAL := 1
    3        ELSE FACTORIAL := I * FACTORIAL(I-1)
    4      END;                                               { FACTORIA }
    4
    1 BEGIN                                                   { EXAMPLE1 }
    1   N := 5;
    2   M := FACTORIAL(N);
    3   WRITELN(N:2,' FACTORIAL = ',M);
    4   WRITELN('NORMAL PROGRAM TERMINATION')
    5 END.                                                    { EXAMPLE1 }
```

At the top of the listing, data appears identifying the compiler, version of compiler, and date and time of compilation. The source listing is provided next.

The LIST option is the default. When NO LIST is specified in the source, any lines appearing between it and the next "LIST" entry will not appear unless they contain errors (see options).

Each line consists of a number followed by the first 72 columns of the source line. In a line in the declaration section, the number ndicates the byte displacement for the variables within the procedures' local stack frame. In the body, the number indicates the statement number of the first statement appearing on the line. The above information is required for debugging.

Syntax and semantic errors are also included in the listing as illustrated next:

A sample listing with error messages:

```
DX  Microprocessor Pascal System Compiler   3.0  06/22/81 10:12:23  PAGE

   0 PROGRAM EXAMPLE2;
   0 CONST
   0   TWO = 2;
   0   TEN = 10;
   0 VAR
   0   X,XTWO,XTEN: REAL;
  12   RESULT: REAL;
  16
   0      PROCEDURE ERROR(ERR:INTEGER);
   1      BEGIN                                                 { ERROR }
   1        IF ERR <> 0
   2        THEN WRITELN('ERROR ##',ERR:2);
   3        ELSE WRITELN('NO ERRORS - NORMAL TERMINATION')
****        !41
   3      END;                                                  { ERROR }
   3
   1 BEGIN                                                  { EXAMPLE 2 }
   1   X := 133.726;
   2   XTWO := X * TWO
   3   XTEN := X * TEN;
****   !14
   3   RESULT := X/((X/XTWO)*XTEN/5.0);
   4   IF RESULT < 0
   5   THEN ERROR(1)
   6   ELSE
   6     IF RESULT = 0
   7     THENERROR(2)
****     !52       !104
   8     ELSE ERROR(0)
   9 END.                                                   { EXAMPLE 2 }
```

Four consecutive asterisks are placed in the left-hand margin under a line containing errors (setting the line off from the rest of the listing. (In cases where a token expected to terminate the previous line was not found, the error will be indicated on the first token in the next line.) An exclamation point "!" is placed beneath the incorrect token. Following the exclamation point is a numeric value associated with the error. The meanings associated with these numeric values are defined in Appendix E and section 5.2.6.1.

One error can cause several error messages separated by commas.

5.2.1.3 <u>Variable Map</u>. The variable map provides information regarding the various declarations in an application and does not address the executable code. The MAP compiler 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 (i.e., 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 with the MAP option specified is shown below:

```
DX  Microprocessor Pascal System Compiler   3.0  06/22/81 10:14:21  PAGE

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

```
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 in the listing begins with a module header. This header indicates the module name and whether the module is a system, program, process, procedure, or function. Any parameters associated with the module are listed after the module name and surrounded by parentheses. For each parameter, the name and type classification are provided. Reference parameters are preceeded by the keyword VAR. When the module is a function, the type of the result is provided following the parentheses. When the module is external, the word "EXTERNAL" comes at the end of the header. When the module is not external, the stack frame size in bytes is provided.

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

The common section follows the variable section. For each COMMON, its name, type, and size in bytes is given. Again, if the module does not declare any COMMONS, this section is omitted.

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 omitted.

## 5.2.2 Compiler Options

Various options are available for controlling compiler execution and output. These options are listed and defined in Table 5-1. Before presenting the table however, information is provided describing options and how they work. Options are Boolean objects, each of which may have the value TRUE or FALSE independent of the values of 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 values 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 option in this class is MAP.

"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 between the BEGIN and the first statement following the BEGIN. The options in this class are DEBUG, NULLBODY, and STATMAP. The remaining class of options may legally be set to new values at any point in a system where a comment could occur. Table 5-1 below lists all available options and defines their default values and meanings.

---

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 the first 72 columns of the source are scanned. This option does not obey the normal scope rules so it must be explicitely turned 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 paragraph 5.2.1.3. (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) |

(Continued)

TABLE 5-1. LISTING CONTROL OPTIONS (CONTINUED).

| OPTION | DEFAULT | MEANING |
|---|---|---|
| STATMAP | FALSE | This option indicates that a map of the displacement for each statement in the object module is to be generated by the Native Code Generator. (ROUTINE SENSITIVE) |
| DEBUG | FALSE | This option should be used if the code is to be debugged in any of the host or target debuggers. Statement level breakpoints may then be used to debug the routine. (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) |
| CKINDEX | FALSE | This option is used to enable run-time checks 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) |

---

### 5.2.3 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 the 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 previously given in Figure 3-2, Section 3, is used in the following sample (pages 5-10 to 5-13) to illustrate possible uses of the ?COPY statement:

```
 {$DEBUG,MAP}

SYSTEM  TUTORIAL;

?COPY USER.SYSDECL

?COPY USER.PRODUCE

?COPY USER.CONSUME


  BEGIN  {#  STACKSIZE = 300;  HEAPSIZE = 500 }
         { Code for SYSTEM Tutorial }

     { Initialize message buffer }
    WITH  M = Message_buffer
      DO  BEGIN
           { Initialize first in and first out to the same slot. So
             long as they are the same and within range any initial
             value will work }
          M.Next_in  := 1;
          M.Next_out := 1;
           { Initialize the exclusive access semaphore for only
             one access at a time }
          INITSEMAPHORE ( M.Exclusive_access, 1 );
           { Initialize the not empty semaphore for no messages
             currently in buffer }
          INITSEMAPHORE ( M.Not_empty, 0 );
           { Initialize the not full semaphore for all messages
             currently empty }
          INITSEMAPHORE ( M.Not_full, Number_of_slots );
      END;


     { Initialization complete. Start the producer and consumer }
    START  Producer;
    START  Consumer;

  END.

  CONST
  Number_of_slots = 10; { Maximum number of slots in buffer }

TYPE
  Slot_index = 1..Number_of_slots;
  Alphabetic = 'A'..'Z';
  Buffer     = RECORD
                  Next_in          : Slot_index;
                  Next_out         : Slot_index;
                  Not_empty        : SEMAPHORE;
                  Not_full         : SEMAPHORE;
                  Exclusive_access : SEMAPHORE;
                  Slots            : ARRAY [Slot_index] OF Alphabetic;
                END;


COMMON
  Message_buffer : Buffer;


ACCESS
  Message_buffer;


PROCEDURE  INITSEMAPHORE (  VAR Sema  : SEMAPHORE;
                                Count : INTEGER );           EXTERNAL;

PROCEDURE  SIGNAL        (      Sema  : SEMAPHORE );         EXTERNAL;

PROCEDURE  WAIT          (      Sema  : SEMAPHORE );         EXTERNAL;

PROCEDURE  SWAP;                                             EXTERNAL;


PROGRAM  PRODUCER;                               { Produce messages }

  VAR
    Item : Alphabetic;
    Line : PACKED ARRAY [1..16] OF CHAR;

  ACCESS
    Message_buffer;


  BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100  }

     { Initialize item so that first message will be 'A' }
    Item := 'Z';
     { Initialize message to inform user of "PRODUCTION" }
    Line := 'Item produced:  ';

    WITH  M = Message_buffer  DO
```

```
      WHILE  TRUE  { i.e. do forever }
         DO  BEGIN
              { Set item to be ´PRODUCED´ }
             IF  Item = ´Z´  THEN  Item := ´A´
                                ELSE  Item := SUCC ( Item );
              { Wait on an empty buffer slot }
             WAIT ( M.Not_full );
              { Wait on exclusive access to the message buffer }
             WAIT ( M.Exclusive_access );
              { Move message to next available slot in buffer }
             M.Slots [ M.Next_in ] := Item;
              { Set pointer to next free slot }
             M.Next_in := SUCC ( M.Next_in MOD Number_of_slots );
              { MOD function produces a value 0..(Number_of_slots-1),
                Ie. 0..9.  If the slot just used was 10 then MOD
                will give 0, and SUCC(0) is 1, which is what we want }

              { Relinquish exclusive access of message buffer }
             SIGNAL ( M.Exclusive_access );
              { Signal that another message was ´PRODUCED´ }
             SIGNAL ( M.Not_empty );
              { Set output message to indicate what was ´PRODUCED´ }
             Line[16] := Item;
              { Output the message to the user }
             MESSAGE ( Line );
              { Give other processes at this priority a
                chance to execute }
             SWAP;
         END;
    { End of PRODUCER program }
  END;


PROGRAM  CONSUMER;                                    { Consume messages }

  VAR
    Item : Alphabetic;
    Line : PACKED ARRAY [1..36] OF CHAR;

  ACCESS
    Message_buffer;


  BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100  }

     { Initialize message to inform user of "CONSUMPTION".
       NOTE: This message has 20 leading blanks to make it print out
       in a different column to the ´item produced´ messages. }
    Line := ´                    Item consumed:  ´;

    WITH  M = Message_buffer  DO
      WHILE  TRUE  { i.e. do forever }
```

```
            DO  BEGIN
                  { Wait on an full buffer slot }
                WAIT ( M.Not_empty );
                  { Wait on exclusive access to the message buffer }
                WAIT ( M.Exclusive_access );
                  { Get message from slot in buffer }
                Item := M.Slots [ M.Next_out ];
                  { Set pointer to next free slot }
                M.Next_out := SUCC ( M.Next_out MOD Number_of_slots );
                  { MOD function produces a value 0..(Number_of_slots-1),
                    Ie. 0..9.  If the slot just used was 10 then MOD
                    will give 0, and SUCC(0) is 1, which is what we want }

                  { Relinquish exclusive access of message buffer }
                SIGNAL ( M.Exclusive_access );
                  { Signal that another message was 'CONSUMED' }
                SIGNAL ( M.Not_full );
                  { Set output message to indicate what was 'CONSUMED' }
                Line[36] := Item;
                  { Output the message to the user }
                MESSAGE ( Line );
                  { Give other processes at this priority a
                    chance to execute }
                SWAP;

          END;

        { End of CONSUMER program }
    END;
```

### 5.2.4 Separate Compilations

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 that are to be compiled together. This segment may then be saved in the form of a standard 9900 object module or MPP Pcode module for later use in debugging of the complete system. All separately compiled segments must be compiled with the same global declaration environment so that they access the same global variables. Any modules which are referenced by but not included in a segment must be declared EXTERNAL. Any global modules that are required only because of their declarations must have null bodies and must also be declared EXTERNAL. Any module declared as having a null body in a 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 on pages 5-10 through 5-13 can be divided into segments as follows:

(SEGMENT 1 - SYSTEM BODY)

```
 {$DEBUG,MAP}

SYSTEM  TUTORIAL;

?COPY USER.SYSDECL

PROGRAM  Producer;  EXTERNAL;

PROGRAM  Consumer;  EXTERNAL;

  BEGIN  {#  STACKSIZE = 300;  HEAPSIZE = 500 }
         { Code for SYSTEM Tutorial }

     { Initialize message buffer }
    WITH  M = Message_buffer
      DO  BEGIN
           { Initialize first in and first out to the same slot. So
             long as they are the same and within range any initial
             value will work }
          M.Next_in  := 1;
          M.Next_out := 1;
           { Initialize the exclusive access semaphore for only
             one access at a time }
          INITSEMAPHORE ( M.Exclusive_access, 1 );
           { Initialize the not empty semaphore for no messages
             currently in buffer }
          INITSEMAPHORE ( M.Not_empty, 0 );
           { Initialize the not full semaphore for all messages
             currently empty }
          INITSEMAPHORE ( M.Not_full, Number_of_slots );
      END;


     { Initialization complete. Start the producer and consumer }
    START  Producer;
    START  Consumer;

  END.
```

(SEGMENT 2 - PRODUCER)

```
 {$DEBUG,MAP}

SYSTEM  TUTORIAL;
```

```
?COPY USER.SYSDECL

  PROGRAM  PRODUCER;                                         { Produce messages }

    VAR
      Item : Alphabetic;
      Line : PACKED ARRAY [1..16] OF CHAR;

    ACCESS
      Message_buffer;


    BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100  }

       { Initialize item so that first message will be 'A' }
      Item := 'Z';
       { Initialize message to inform user of "PRODUCTION" }
      Line := 'Item produced:  ';

      WITH  M = Message_buffer  DO
        WHILE  TRUE  { i.e. do forever }
           DO  BEGIN
                { Set item to be 'PRODUCED' }
               IF  Item = 'Z'  THEN  Item := 'A'
                                ELSE  Item := SUCC ( Item );
                { Wait on an empty buffer slot }
               WAIT ( M.Not_full );
                { Wait on exclusive access to the message buffer }
               WAIT ( M.Exclusive_access );
                { Move message to next available slot in buffer }
               M.Slots [ M.Next_in ] := Item;
                { Set pointer to next free slot }
               M.Next_in := SUCC ( M.Next_in MOD Number_of_slots );
                { MOD function produces a value 0..(Number_of_slots-1),
                  Ie. 0..9.  If the slot just used was 10 then MOD
                  will give 0, and SUCC(0) is 1, which is what we want }

                { Relinquish exclusive access of message buffer }
               SIGNAL ( M.Exclusive_access );
                { Signal that another message was 'PRODUCED' }
               SIGNAL ( M.Not_empty );
                { Set output message to indicate what was 'PRODUCED' }
               Line[16] := Item;
                { Output the message to the user }
               MESSAGE ( Line );
                { Give other processes at this priority a
                  chance to execute }
               SWAP;
          END;
      { End of PRODUCER program }
    END;
  BEGIN  {$NULLBODY}
  END.
```

(SEGMENT 3 - CONSUMER)

```
 {$DEBUG,MAP}

SYSTEM  TUTORIAL;

?COPY USER.SYSDECL


  PROGRAM  CONSUMER;                                    { Consume messages }

    VAR
      Item : Alphabetic;
      Line : PACKED ARRAY [1..36] OF CHAR;

    ACCESS
      Message_buffer;


    BEGIN  {#  PRIORITY = 20;  STACKSIZE = 100   }

       { Initialize message to inform user of "CONSUMPTION".
         NOTE: This message has 20 leading blanks to make it print out
         in a different column to the ´item produced´ messages. }
      Line := ´                    Item consumed:  ´;

      WITH  M = Message_buffer  DO
        WHILE  TRUE  { i.e. do forever }
           DO  BEGIN
                { Wait on an full buffer slot }
               WAIT ( M.Not_empty );
                { Wait on exclusive access to the message buffer }
               WAIT ( M.Exclusive_access );
                { Get message from slot in buffer }
               Item := M.Slots [ M.Next_out ];
                { Set pointer to next free slot }
               M.Next_out := SUCC ( M.Next_out MOD Number_of_slots );
                { MOD function produces a value 0..(Number_of_slots-1),
                  Ie. 0..9.  If the slot just used was 10 then MOD
                  will give 0, and SUCC(0) is 1, which is what we want }

                { Relinquish exclusive access of message buffer }
               SIGNAL ( M.Exclusive_access );
                { Signal that another message was ´CONSUMED´ }
               SIGNAL ( M.Not_full );
                { Set output message to indicate what was ´CONSUMED´ }
                Line[36] := Item;
                 { Output the message to the user }
                MESSAGE ( Line );
                 { Give other processes at this priority a
                   chance to execute }
                SWAP;

           END;

       { End of CONSUMER program }
    END;



  BEGIN  {$NULLBODY}
  END.
```

In the example 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.

5.2.4.1 Mechanisms To Obtain Valid System Code

If the system is constructed via separate compilations and certain variables are being accessed via scope across "compilation boundaries", it is absolutely imperative that equivalent variables reside in the same "dynamic" location. Two mechanisms, the MPP compiler and the MPP code generator, must be carefully controlled in order to prevent system failures during execution.

5.2.4.1.1 The MPP Compiler: This compiler does not automatically save the display pointers on the stack unless it perceives an inner nesting level. The compiler can be made to "see" the presence of inner nesting levels for externally defined procedures through the use of a "NULLBODY" compiler directive, rather than actually declaring the procedure as EXTERNAL (See Example of Separate Compilations (Main), below). A separate compiling of an inner procedure must inform the compiler about the nesting environment (See Example of Separate Compilations (Secondary), following).

EXAMPLE: SEPARATE COMPILATIONS (MAIN).

```
          SYSTEM TEST;                              "Main Compilation"

            PROCEDURE CODE_GEN_FOOLER; FORWARD;

            PROGRAM SCOPE(P1,P2: SOME_TYPE);
            VAR X,Y,Z: SOME_TYPE
```

```
        PROCEDURE  OUTER(E,F,G: SOME_TYPE);
        VAR P,Q,R: SOME_TYPE;

          PROCEDURE INNER(A,B: SOME_TYPE);
            BEGIN                               (* INNER *)
                 (*$NULLBODY *)      "Forces update of the display"
            END;                                (* INNER *)

          BEGIN                                 (* OUTER *)
               (* PROCEDURE OUTER activity *)
          END;                                  (* OUTER *)

        BEGIN                                   (* SCOPE *)
             (* PROGRAM SCOPE acitivity *)
        END                                     (* SCOPE *)

      BEGIN                                     (* TEST *)
           (* SYSTEM TEST activity *)
      END                                       (* TEST *)
```

5.2.4.1.2 The MPP Code Generator: In this case, the register assignment algorithm tries to optimize code through more effecient use of available registers. Variables that would have been stored on the stack can be placed in registers. Thus, variable access via scope becomes a matter of bookeeping for the code generator, and the complete variable environment which is subject to scope access must be known during this activity. To stop this activity of the code generator, a "FORWARD" reference to a nonexistent procedure can be used. Figure 5- showed the main compilation segment containing an INNER procedure which might use any of the variables referenced within the boundaries of the system. The next sample shows the corresponding secondary compilation segment the definition of procedure INNER.

NOTE:

1. The stack for each task <u>must</u> be exactly the same between the main and any of the secondary compilation parts when scope access of variables is employed. Separate compilations done with without observing the preceeding control mechanisms can result in serious run-time error conditions.

2. Separate compilations at the PROGRAM level are immune to the above problem because MPP forbids variables at any system level other than COMMON.

EXAMPLE: SEPARATE COMPILATIONS (SECONDARY).

```
    SYSTEM DUMMY1;
      PROCEDURE CODE_GEN_FOOLER; FORWARD;
      PROGRAM DUMMY2(P1,P2: SOME_TYPE);    "Secondary Compilation"
      VAR X,Y,Z: SOME_TYPE;

        PROCEDURE OUTER(E,F,G: SOME_TYPE);
        VAR P,Q,R: SOME_TYPE
        BEGIN                                   (* INNER *)
        END; (* PROCEDURE INNER activity *)     (* INNER *)

      BEGIN                                     (* OUTER *)
               (*$ NULLBODY *)
      END;                                      (* OUTER *)

    BEGIN                                       (* DUMMY2 *)
               (* NULLBODY *)
    END;                                        (* DUMMY2 *)

  BEGIN                                         (* DUMMY1 *)
               (* NULLBODY *)
  END.                                          (* DUMMY1 *)
```

## 5.2.4.2 Differences Between Native and Pcode Environments

When executing in Pcode, it is possible and acceptable to have two procedures that have the same name but that can be uniquely identified through scope of access. This is not possible in the native code environment because the link edit phase has no concept of scope in the manner that scope is defined in MPP.

The example below will compile, save, and codegen in Pcode, but, although is will also execute using the Host "EXECUTE" command, it will not correctly link in native code, nor can it be run under the Host debugger.

EXAMPLE: SCOPE OF PROCEDURES.

```
SYSTEM EXAMPLE;
   .  .  .
   .  .  .
   .  .  .
   Program A;
       Process X;
       .  .  .
       .  .  .
   Begin B.
       .  .  .
       .  .  .
     Start X;
   END;
```

```
Program B;
  Process X;
  . . .
  . . .
BEGIN
```

## 5.2.5 Saving Segments

The Microprocessor Pascal System provides a utility that takes the interpretive code generated by the compiler and converts it into a standard 990 tagged object file format. (NOTE: only the format is changed; the file still contains MPP Pcodes, not 990 object code.) This object module can be included in a Host debug session or included in an Interpretive RTS target system.

The utility requests the listing file access name and the segment file access name. The input files to this utility are temporary files created by the previous compile; therefore, the only time a segment and be saved is immediately follwing the compilation. The segment file is the output file that will contain the 990 tagged object format Pcode file.

The utility to produce the segment will prompt for the segment number to be assigned to this segment, and whether or not debug information is to be placed into the object module. The prompt file is shown below; the responses are proceeded by "-->":

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

The segment number is needed for Interpretive RTS system construction and 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, beginning on page 5-14, 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 information at the top of the listing indicates the segment number, segment length, and debug information status. In the body of the listing, each module in the system is indicated by the name and number. Each module is also identified as an 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.

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 thoroughly tested, it may be saved without debug information (to conserve space), but still may be used to test other parts of the system.

### 5.2.6 Compiler Error Messages

This subsection describes the error messages generated by the compiler. Paragraph 5.2.6.1 describes the error messages generated by the compiler when it finds errors. Paragraph 5.2.6.2 describes all other error messages generated by the compiler.

5.2.6.1 Syntax Error Number Descriptions. This section describes each error number generated by the compiler.

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 se 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 no 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 uppe 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 semaphor 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 constant 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 cause 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 on 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.

5.2.6.2 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 identifie 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.

## 5.3 THE NATIVE CODE GENERATOR

Native 9900 code is generated by the code generator. A utility program is also provided that will produce assembler source from the object module produced by the code generator. The information that follows describes each of these programs. Listings generated by each program are also presented.

### 5.3.1 CODEGEN

As previously described, the intermediate code generated by the compiler can be executed interpretively or converted into 9900 object code. CODEGEN provides the capability to make this conversion.

The input to the code generator is the intermediate code produced directly by the compiler and not the interpretive code saved by the save segment utility.

5.3.1.1 CODEGEN Execution Messages. The code generator produces messages similiar to the compiler as it executes. These messages indicate how much of the system has been processed. These messages are normally output to the user´s display and appear as follows:

```
              Execution begins.
              PRODUCER
              CONSUMER TUTORIAL
              Stack used =   1314  Heap used =   546
              Execution Ends.
              CODGEN EXECUTION COMPLETE.
```

The first and last three message lines are generated by the operating system. Each of the remaining lines appears as code is generated for the module named. Note that only the first eight characters of a module name is displayed.

5.3.1.2 Code Generator Listing. An example of the listing produced by the code generator follows:

```
MODULE - PRODUCER
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0012
   R13 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
* LITERAL CODE LENGTH = 0022,   TOTAL CODE LENGTH = 010C

MODULE - CONSUMER
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0026
   R12 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
* LITERAL CODE LENGTH = 0036,   TOTAL CODE LENGTH = 0106

MODULE - TUTORIAL
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
* LITERAL CODE LENGTH = 0018,   TOTAL CODE LENGTH = 00A0
```

For each module processed, the name of the module is listed first. Next, any local variables assigned to registers are listed followed by the total length of the object module. The literal data length in hexadecimal bytes is also provided.

Registers R4, R5, R6, R12, and R15 are normally available for global allocation. Either the address or value of the most used variables are placed into registers. The value of a variable is placed into a register only if the module has no lexical sons and if the variable contains a single word value. Otherwise, the address of the variable is placed in the register. The code generator only indicates if a value is placed in a register, because in this case, the location in the stack frame is not used. Rather, the value resides in the indicated register.

The code generator also has a statement map option (STATMAP) which allows it to display the hexidecimal displacement in the object module for each Pascal statement. An example of the listing produced when the STATMAP option is specified is:

```
MODULE - PRODUCER
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0012
   R13 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
  1 0044    2 004C    3 0062    4 006A    5 006E    6 0078    7 0082
  8 0088    9 0094   10 00A0   11 00AE   12 00C0   13 00CC   14 00D8
 15 00E4   16 00F2   17 00FC
* LITERAL CODE LENGTH = 0022,   TOTAL CODE LENGTH = 010C

MODULE - CONSUMER
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0026
   R12 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
  1 0058    2 006E    3 0076    4 007A    5 0086    6 0092    7 00A2
  8 00BA    9 00C6   10 00D2   11 00DE   12 00EC   13 00F6
* LITERAL CODE LENGTH = 0036,   TOTAL CODE LENGTH = 0106

MODULE - TUTORIAL
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0000
  1 0034    2 003C    3 0044    4 004E    5 0060    6 0070    7 0082
  8 008A    9 0092
* LITERAL CODE LENGTH = 0018,   TOTAL CODE LENGTH = 00A0
```

The statement map appears immediately before the code length messages. For each statement, the statement number and its appropriate hexadecimal dsplacement are printed.

### 5.3.2 Reverse Assembler

The Reverse Assembler (RASS) provides a TMS9900 family assembly language source program that corresponds to an object module generated by the code generator. The output of reverse assembler may be directly assembled. Submitting an object module to the reverse assembler to obtain the assembly language source code allows the user to perform manual optimization when appropriate. The assembly language source code also allows debugging at the machine language level.

5.3.2.1 Reverse Assembler Execution Messages. The reverse assemble outputs messages similar to the compiler and code generator. Such messages indicate the name of the module being processed.

```
Execution begins
PRODUCER
CONSUMER
TUTORIAL
Stack used =   1906  Heap used =  1352
Execution ends
REVERSE ASSEMBLY IS COMPLETE.:
```

5.3.2.2 Rass Listing File. Below is an example of the assembly listing for the module FACTORIA. The listing begins with an IDT directive containing the module name.

Following this line are the DEF directive and the REF directives. Next comes the PSEG directive. The comment line to the right of the PSEG directive provides a heading for location counter values and the contents of each data work area displayed. These contents are indicated in hexadecimal and ASCII representations. Non-printable characters are displayed as periods. When the data word represents a relocatable address, the hexadecimal value is followed by a plus sign (+).

After the directives, another comment provides a heading for the instructions which follow. For each instruction the listing shows the location counter value and the value of the word or words of the machine language instruction in hexadecimal format. The file that contains the listing (OUTPUT) may be used as a source code file for the assembler. The values that appear at the right end of most lines are treated as comments by the assembler.

EXAMPLE: RASS LISTING.

```
*
       IDT   'PRODUCER'                     7/ 1/81  13:42: 3
*
       CSEG 'MESSAG'
MESSAG BSS   30
*
       DEF   PRODUC
       REF   SIGNAL
       REF   WAIT
       REF   SWAP
       REF   MSG$
       REF   S$PRCS
       REF   E$PRCS
       REF   CALL$
       REF   STAT$
       REF   I$DIVC
       REF   EXIT$P
*                                            LC    HEX   CHAR
       PSEG
PRODUC EQU   $
PR     EQU   7   R7
CODE   EQU   8   R8
LF     EQU   9   R9
SP     EQU   10  R10
L0     EQU   $
       DATA  L0022-L0
       DATA  L00FC-L0
       DATA  >0000                          0004   0000   ..
       DATA  >0000                          0006   0000   ..
D0008  DATA  >0012                          0008   0012   ..
D000A  DATA  >0001                          000A   0001   ..
D000C  DATA  >0014                          000C   0014   ..
D000E  DATA  >0064                          000E   0064   .d
```

```
       DATA >4974                     0010    4974    It
       DATA >656D                     0012    656D    em
       DATA >2070                     0014    2070     p
       DATA >726F                     0016    726F    ro
       DATA >6475                     0018    6475    du
       DATA >6365                     001A    6365    ce
       DATA >643A                     001C    643A    d:
       DATA >2020                     001E    2020
D0020  DATA >0010                     0020    0010    ..
*                                       LC    WORD(S)
L0022  EQU  $
       MOV  @D0008-L0(CODE),*SP+      0022    CEA8    0008
       MOV  @D000A-L0(CODE),*SP+      0026    CEA8    000A
       MOV  @D000C-L0(CODE),*SP+      002A    CEA8    000C
       MOV  @D000E-L0(CODE),*SP+      002E    CEA8    000E
       CLR  *SP+                      0032    04FA
       DATA CALL$,S$PRCS              0034
       LI   R15,STAT$                 0038    020F    0000
       BL   *R15                      003C    069F
       DATA >0000                     003E    0000
       MOV  LF,R12                    0040    C309
       INCT R12                       0042    05CC
       BL   *R15                      0044    069F
       DATA >0001                     0046    0001
       LI   R13,>005A                 0048    020D    005A
       BL   *R15                      004C    069F
       DATA >0002                     004E    0002
       MOV  CODE,R6                   0050    C188
       AI   R6,>0010                  0052    0226    0010
       MOV  R12,R5                    0056    C14C
       LI   R4,>0008                  0058    0204    0008
L005C  EQU  $
       MOV  *R6+,*R5+                 005C    CD76
       DEC  R4                        005E    0604
       JNE  L005C                     0060    16FD
       BL   *R15                      0062    069F
       DATA >0003                     0064    0003
       LI   R14,MESSAG                0066    020E    0000+
       BL   *R15                      006A    069F
       DATA >0004                     006C    0004
L006E  EQU  $
       BL   *R15                      006E    069F
       DATA >0005                     0070    0005
       CI   R13,>005A                 0072    028D    005A
       JNE  L0082                     0076    1605
       BL   *R15                      0078    069F
       DATA >0006                     007A    0006
       LI   R13,>0041                 007C    020D    0041
       JMP  L0088                     0080    1003
L0082  EQU  $
       BL   *R15                      0082    069F
       DATA >0007                     0084    0007
       INC  R13                       0086    058D
L0088  EQU  $
```

```
        BL    *R15                          0088    069F
        DATA  >0008                         008A    0008
        MOV   @>0006(R14),*SP+              008C    CEAE     0006
        DATA  CALL$,WAIT                    0090
        BL    *R15                          0094    069F
        DATA  >0009                         0096    0009
        MOV   @>0008(R14),*SP+              0098    CEAE     0008
        DATA  CALL$,WAIT                    009C
        BL    *R15                          00A0    069F
        DATA  >000A                         00A2    000A
        MOV   *R14,R6                       00A4    C19E
        SLA   R6,1                          00A6    0A16
        A     R14,R6                        00A8    A18E
        MOV   R13,@>0008(R6)                00AA    C98D     0008
        BL    *R15                          00AE    069F
        DATA  >000B                         00B0    000B
        MOV   *R14,*SP+                     00B2    CE9E
        DATA  CALL$,I$DIVC                  00B4
        DATA  >000A                         00B8
        MOV   @>0002(SP),*R14               00BA    C7AA     0002
        INC   *R14                          00BE    059E
        BL    *R15                          00C0    069F
        DATA  >000C                         00C2    000C
        MOV   @>0008(R14),*SP+              00C4    CEAE     0008
        DATA  CALL$,SIGNAL                  00C8
        BL    *R15                          00CC    069F
        DATA  >000D                         00CE    000D
        MOV   @>0004(R14),*SP+              00D0    CEAE     0004
        DATA  CALL$,SIGNAL                  00D4
        BL    *R15                          00D8    069F
        DATA  >000E                         00DA    000E
        STWP  R6                            00DC    02A6
        MOVB  @>001B(R6),@>000F(R12)        00DE    DB26     001B     000F
        BL    *R15                          00E4    069F
        DATA  >000F                         00E6    000F
        MOV   R12,*SP+                      00E8    CE8C
        MOV   @D0020-L0(CODE),*SP+          00EA    CEA8     0020
        DATA  CALL$,MSG$                    00EE
        BL    *R15                          00F2    069F
        DATA  >0010                         00F4    0010
        DATA  CALL$,SWAP                    00F6
        JMP   L006E                         00FA    10B9
L00FC   EQU   $
        BL    *R15                          00FC    069F
        DATA  >0011                         00FE    0011
        MOV   @D000A-L0(CODE),*SP+          0100    CEA8     000A
        DATA  CALL$,E$PRCS                  0104
        B     @EXIT$P                       0108    0460     0000
        END
*
        IDT   'CONSUMER'                       7/ 1/81  13:42: 8
*
        CSEG  'MESSAG'
MESSAG  BSS   30
```

```
*
        DEF   CONSUM
        REF   SIGNAL
        REF   WAIT
        REF   SWAP
        REF   MSG$
        REF   S$PRCS
        REF   E$PRCS
        REF   CALL$
        REF   STAT$
        REF   I$DIVC
        REF   EXIT$P
*                                                   LC      HEX     CHAR
        PSEG
CONSUM  EQU   $
PR      EQU   7   R7
CODE    EQU   8   R8
LF      EQU   9   R9
SP      EQU   10  R10
L0      EQU   $
        DATA  L0036-L0
        DATA  L00F6-L0
        DATA  >0000                                 0004    0000     ..
        DATA  >0000                                 0006    0000     ..
D0008   DATA  >0026                                 0008    0026     .&
D000A   DATA  >0001                                 000A    0001     ..
D000C   DATA  >0014                                 000C    0014     ..
D000E   DATA  >0064                                 000E    0064     .d
        DATA  >2020                                 0010    2020
        DATA  >2020                                 0012    2020
        DATA  >2020                                 0014    2020
        DATA  >2020                                 0016    2020
        DATA  >2020                                 0018    2020
        DATA  >2020                                 001A    2020
        DATA  >2020                                 001C    2020
        DATA  >2020                                 001E    2020
        DATA  >2020                                 0020    2020
        DATA  >2020                                 0022    2020
        DATA  >4974                                 0024    4974     It
        DATA  >656D                                 0026    656D     em
        DATA  >2063                                 0028    2063      c
        DATA  >6F6E                                 002A    6F6E     on
        DATA  >7375                                 002C    7375     su
        DATA  >6D65                                 002E    6D65     me
        DATA  >643A                                 0030    643A     d:
        DATA  >2020                                 0032    2020
D0034   DATA  >0024                                 0034    0024     .$
*                                                   LC      WORD(S)
L0036   EQU   $
        MOV   @D0008-L0(CODE),*SP+                  0036    CEA8     0008
        MOV   @D000A-L0(CODE),*SP+                  003A    CEA8     000A
        MOV   @D000C-L0(CODE),*SP+                  003E    CEA8     000C
        MOV   @D000E-L0(CODE),*SP+                  0042    CEA8     000E
        CLR   *SP+                                  0046    04FA
```

```
       DATA CALL$,S$PRCS               0048
       LI   R15,STAT$                  004C    020F    0000
       BL   *R15                       0050    069F
       DATA >0000                      0052    0000
       MOV  LF,R13                     0054    C349
       INCT R13                        0056    05CD
       BL   *R15                       0058    069F
       DATA >0001                      005A    0001
       MOV  CODE,R6                    005C    C188
       AI   R6,>0010                   005E    0226    0010
       MOV  R13,R5                     0062    C14D
       LI   R4,>0012                   0064    0204    0012
L0068  EQU  $
       MOV  *R6+,*R5+                  0068    CD76
       DEC  R4                         006A    0604
       JNE  L0068                      006C    16FD
       BL   *R15                       006E    069F
       DATA >0002                      0070    0002
       LI   R14,MESSAG                 0072    020E    0000+
       BL   *R15                       0076    069F
       DATA >0003                      0078    0003
L007A  EQU  $
       BL   *R15                       007A    069F
       DATA >0004                      007C    0004
       MOV  @>0004(R14),*SP+           007E    CEAE    0004
       DATA CALL$,WAIT                 0082
       BL   *R15                       0086    069F
       DATA >0005                      0088    0005
       MOV  @>0008(R14),*SP+           008A    CEAE    0008
       DATA CALL$,WAIT                 008E
       BL   *R15                       0092    069F
       DATA >0006                      0094    0006
       MOV  @>0002(R14),R6             0096    C1AE    0002
       SLA  R6,1                       009A    0A16
       A    R14,R6                     009C    A18E
       MOV  @>0008(R6),R12             009E    C326    0008
       BL   *R15                       00A2    069F
       DATA >0007                      00A4    0007
       MOV  @>0002(R14),*SP+           00A6    CEAE    0002
       DATA CALL$,I$DIVC               00AA
       DATA >000A                      00AE
       MOV  @>0002(SP),@>0002(R14)     00B0    CBAA    0002    0002
       INC  @>0002(R14)                00B6    05AE    0002
       BL   *R15                       00BA    069F
       DATA >0008                      00BC    0008
       MOV  @>0008(R14),*SP+           00BE    CEAE    0008
       DATA CALL$,SIGNAL               00C2
       BL   *R15                       00C6    069F
       DATA >0009                      00C8    0009
       MOV  @>0006(R14),*SP+           00CA    CEAE    0006
       DATA CALL$,SIGNAL               00CE
       BL   *R15                       00D2    069F
       DATA >000A                      00D4    000A
       STWP R6                         00D6    02A6
```

```
        MOVB @>0019(R6),@>0023(R13)            00D8     DB66     0019     0023
        BL   *R15                              00DE     069F
        DATA >000B                             00E0     000B
        MOV  R13,*SP+                          00E2     CE8D
        MOV  @D0034-L0(CODE),*SP+              00E4     CEA8     0034
        DATA CALL$,MSG$                        00E8
        BL   *R15                              00EC     069F
        DATA >000C                             00EE     000C
        DATA CALL$,SWAP                        00F0
        JMP  L007A                             00F4     10C2
L00F6   EQU  $
        BL   *R15                              00F6     069F
        DATA >000D                             00F8     000D
        MOV  @D000A-L0(CODE),*SP+              00FA     CEA8     000A
        DATA CALL$,E$PRCS                      00FE
        B    @EXIT$P                           0102     0460     0000
        END
*
        IDT  'TUTORIAL'                          7/ 1/81   13:42:11
*
        CSEG 'MESSAG'
MESSAG  BSS  30
*
        DEF  SYSTM$
        REF  INITSE
        REF  PRODUC
        REF  CONSUM
        REF  S$PRCS
        REF  E$PRCS
        REF  CALL$
        REF  STAT$
        REF  EXIT$P
*                                                LC      HEX     CHAR
        PSEG
SYSTM$  EQU  $
PR      EQU  7   R7
CODE    EQU  8   R8
LF      EQU  9   R9
SP      EQU  10  R10
L0      EQU  $
        DATA L0018-L0
        DATA L0092-L0
        DATA >0000                             0004     0000      ..
        DATA >0000                             0006     0000      ..
D0008   DATA >0000                             0008     0000      ..
D000A   DATA >012C                             000A     012C      .,
D000C   DATA >01F4                             000C     01F4      ..
D000E   DATA >0001                             000E     0001      ..
D0010   DATA >0008                             0010     0008      ..
D0012   DATA >0004                             0012     0004      ..
D0014   DATA >0006                             0014     0006      ..
D0016   DATA >000A                             0016     000A      ..
*                                                LC     WORD(S)
L0018   EQU  $
```

```
         MOV  @D0008-L0(CODE),*SP+          0018    CEA8     0008
         CLR  *SP+                          001C    04FA
         SETO *SP+                          001E    073A
         MOV  @D000A-L0(CODE),*SP+          0020    CEA8     000A
         MOV  @D000C-L0(CODE),*SP+          0024    CEA8     000C
         DATA CALL$,S$PRCS                  0028
         LI   R15,STAT$                     002C    020F     0000
         BL   *R15                          0030    069F
         DATA >0000                         0032    0000
         BL   *R15                          0034    069F
         DATA >0001                         0036    0001
         LI   R14,MESSAG                    0038    020E     0000+
         BL   *R15                          003C    069F
         DATA >0002                         003E    0002
         MOV  @D000E-L0(CODE),*R14          0040    C7A8     000E
         BL   *R15                          0044    069F
         DATA >0003                         0046    0003
         MOV  @D000E-L0(CODE),@>0002(R14)   0048    CBA8     000E     0002
         BL   *R15                          004E    069F
         DATA >0004                         0050    0004
         MOV  R14,*SP                       0052    C68E
         A    @D0010-L0(CODE),*SP+          0054    AEA8     0010
         MOV  @D000E-L0(CODE),*SP+          0058    CEA8     000E
         DATA CALL$,INITSE                  005C
         BL   *R15                          0060    069F
         DATA >0005                         0062    0005
         MOV  R14,*SP                       0064    C68E
         A    @D0012-L0(CODE),*SP+          0066    AEA8     0012
         CLR  *SP+                          006A    04FA
         DATA CALL$,INITSE                  006C
         BL   *R15                          0070    069F
         DATA >0006                         0072    0006
         MOV  R14,*SP                       0074    C68E
         A    @D0014-L0(CODE),*SP+          0076    AEA8     0014
         MOV  @D0016-L0(CODE),*SP+          007A    CEA8     0016
         DATA CALL$,INITSE                  007E
         BL   *R15                          0082    069F
         DATA >0007                         0084    0007
         DATA CALL$,PRODUC                  0086
         BL   *R15                          008A    069F
         DATA >0008                         008C    0008
         DATA CALL$,CONSUM                  008E
L0092    EQU  $
         BL   *R15                          0092    069F
         DATA >0009                         0094    0009
         CLR  *SP+                          0096    04FA
         DATA CALL$,E$PRCS                  0098
         B    @EXIT$P                       009C    0460     0000
         END
```

The reverse assembler is specifically designed to process the object modules produced by the code generator and has limited application for processing other object modules because it cannot recognize which words are data and which are instructions.

### 5.3.3 Code Generator Error Messages

The following error messages are generated by the code generator when an internal error has occurred. With few exceptions, the user should contact the local Regional Training Center to resolve the problem signaled.

BADOP <number> IN STATEMENT <number>

BAD OPERAND <number> IN STATEMENT <number>

BAD STATE <number> IN STATEMENT <number>

STATEMENT <number> TOO COMPLEX -- NO REGISTERS The user may correct this error by simplifing the statement indicated.

TEMPORARIES NOT FREED IN STATEMENT <number>

The following error messages are fatal internal errors; the code generation process will stop if one of these errors occurs.

CANNOT FIND LABEL

END OF FILE ON PCODE

INVALID LABEL

SET LITERAL TOO LONG

STACK OVERFLOW

STRING LITERAL TOO LONG

TOO MANY COMMONS REFERENCED

TOO MANY EXTERNALS REFERENCED

# SECTION 6

# HOST DEBUGGER GUIDE

## 6.1 OVERVIEW

A flaw or error in software is commonly called a bug. The act of correcting/removing these bugs is known as debugging. While the Source Editor helps the user construct a syntactically correct program, the Microprocessor Pascal System language, along with the compiler, helps the user discover and correct semantic errors. Because Pascal itself prohibits the use of semantically inconsistent operations on data, the user is freed from many traditional debugging chores.

The design, implementation, and testing of large, complex systems is a difficult task. The ideal approach to this problem is to break up large pieces of software into small, independent units. The smaller modules can then be examined individually to ensure that they perform as desired. 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.

In addition, the Host Debugger is an interactive debugging tool that is useful in observing the behavior of Microprocessor Pascal Systems as they evolve; and as such is also a useful design tool. It can be used to ensure that modules perform correctly by themselves. The debugger is also useful for monitoring and altering the interfaces between modules and concurrent processes. The Debugger´s user interface is designed to facilitate the debugging effort. Whenever the Debugger expects a response from the user, a prompting message (usually consisting of the characters "<>") is displayed. 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 hard copy if desired. User input commands, debugger responses, trace information, and status displays are sent to a log file, stored on disc at some user determined pathname. At times it is helpful to use this information to track the steps that resulted in a certain state in the program. HELP displays are not echoed on the log file. Note also that the log file does not contain any user input, output, or messages.

Table 6-1 is a summary of the debugger command names. A detailed description of each command is given in Paragraph 6.3.

TABLE 6-1. HOST DEBUG COMMANDS

| 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 |
| Process 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 |

(Continued)

TABLE 6-1. HOST DEBUG COMMANDS. (CONTINUED)

| | |
|---|---|
| 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 Subsection 5.2.2 for the available compiler options). Consider the following example:

```
 X Microprocessor Pascal System Compiler   3.0  02/28/81 10:25:55  PAGE

S  0 {$ DEBUG, MAP }
T  0 PROGRAM EXAMPLE;
A  0 VAR
T  0    N: INTEGER;          (FORMAL PARAMETER)
E  2    M: INTEGER;
M  4
E  0       FUNCTION FACTORIAL(I:INTEGER): INTEGER;
N  1       BEGIN                                          { FACTORIAL }
T  1         IF I <= 1
   2         THEN FACTORIAL := 1
N  3         ELSE FACTORIAL := I * FACTORIAL(I-1)
U  4       END;                                           { FACTORIAL }
M  4
B  1 BEGIN                                                { EXAMPLE}
E  1 {# STACKSIZE=200 }
R  1    N:= 5;
S  2    M:= FACTORIAL(N)
   3 END.                                                 { EXAMPLE }


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 FACTORIAL´s stack frame. The function result is stored at displacement 2. The statement numbers are listed in the body section.

The following is an example of an interactive debugging session. All user input commands are preceded by the characters <>. Output messages from the debugger are displayed following many of the commands, although some commands do not evoke a response from the debugger. Explanatory comments about the example walk-through are enclosed between braces { }.

```
                         { start of debugging session }


 HOST DEBUGGER          02/28/81  10:29:08
  System heap size = 5 (K)bytes
  Most recently compiled system will be loaded.

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

<>GO                   { execute }
     run-time support now initialized

<>GO                   { resume execution }
     *** 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                   { list breakpoints }
     Breakpoints for Process EXAMPL(1)
       EXAMPL          1
       FACTOR          1

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

<>LB                   { list breakpoints }
     Breakpoints for Process EXAMPL(1)
       FACTOR          1

<>AB(EXAMPLE, 3)       { assign breakpoint to statement 3 of EXAMPLE }

<>AB(EXAMPLE, 2)       { assign breakpoint to statement 2 of EXAMPLE }

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

<>LB                   { list breakpoints }
     Breakpoints for Process EXAMPL(1)
       EXAMPL          1
       EXAMPL          2
       EXAMPL          3
       FACTOR          1

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).EXAMPL    Statement 1

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 2  Current = 2

     Call Order           Name                 Statement
          1            EXAMPL                      1
```

```
<>SF(EXAMPLE)          { show frame for EXAMPLE }
     stack frame for EXAMPL(1).EXAMPL
 C100 (0000) 0000 0000                                (....             )
{ frame as exists prior to execution of statement 1 }

<>SF(FACTORIAL)        { show frame for FACTORIAL }
     stack frame not found
{ as of yet, FACTORIAL as not been called }

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).EXAMPL    Statement 2

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 2  Current = 2

     Call Order          Name                  Statement
          1           EXAMPL                      2
{ only EXAMPLE has been called }

<>SF                   { show frame for default routine }
     stack frame for EXAMPL(1).EXAMPL
 C100 (0000) 0005 0000                                (....
{ N = 5; stored at displacement 0000 of EXAMPLE }

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).FACTOR    Statement 1

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 18  Current = 18

     Call Order          Name                  Statement
          1           EXAMPL                      2
          2             FACTOR                    1
{ FACTORIAL called }

<>SF                   { show frame for default routine }
     stack frame for EXAMPL(1).FACTOR
 C122 (0000) 0005 0000                                (....             )
{ FACTORIAL called with I = 5 }

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).FACTOR    Statement 1

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 35  Current = 35

     Call Order           Name               Statement
           1            EXAMPL                   2
           2              FACTOR                 3
           3              FACTOR                 1
{ FACTORIAL called recursively }

<>SF                  { show frame for default routine }
     stack frame for EXAMPL(1).FACTOR
 C144 (0000) 0004 0001                                (....           )
{ FACTORIAL called with I = 4 }

<>GO                  { resume execution }
     *** Breakpoint ***      EXAMPL(1).FACTOR    Statement 1

<>DP                  { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 52  Current = 52

     Call Order           Name               Statement
           1            EXAMPL                   2
           2              FACTOR                 3
           3              FACTOR                 3
           4              FACTOR                 1
{ FACTORIAL called recursively again }

<>SF                  { show frame for default routine }
     stack frame for EXAMPL(1).FACTOR
 C166 (0000) 0003 0001                                (....           )
{ FACTORIAL called with I = 3 }

<>GO                  { resume execution }
     *** Breakpoint ***      EXAMPL(1).FACTOR    Statement 1

<>DP                  { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 69  Current = 69

     Call Order           Name               Statement
           1            EXAMPL                   2
           2              FACTOR                 3
           3              FACTOR                 3
           4              FACTOR                 3
           5              FACTOR                 1
{ FACTORIAL called recursively again }

<>SF                  { show frame for default routine }
     stack frame for EXAMPL(1).FACTOR
 C188 (0000) 0002 0001                                (....           )
```

```
{ FACTORIAL called with I = 2 }

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).FACTOR    Statement 1

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 86  Current = 86

     Call Order          Name                Statement
         1           EXAMPL                      2
         2             FACTOR                    3
         3             FACTOR                    3
         4             FACTOR                    3
         5             FACTOR                    3
         6             FACTOR                    1
  { FACTORIAL called recursively again }

<>SF                   { show frame for the default routine }
     stack frame for EXAMPL(1).FACTOR
 C1AA (0000) 0001 0001                              (....                 )
{ FACTORIAL called with I = 1 }

<>GO                   { resume execution }
     *** Breakpoint ***      EXAMPL(1).EXAMPL    Statement 3

<>DP                   { display process - default process chosen }
Static/Dynamic Calling Order for Process EXAMPL(1)

     Stack Size (words) = 200
     Stack Used (words) Maximum = 86  Current = 2

     Call Order          Name                Statement
         1           EXAMPL                      3
  { function FACTORIAL complete }

<>SF                   { show frame for default routine }
     stack frame for EXAMPL(1).EXAMPL
 C100 (0000) 0005 0078                              (...x                 )
{ value of M is >78 = 5 factorial; stored at displacement 0002 }

<>GO                   { resume execution }
     *** Process Terminated *** EXAMPL(1)
         Stack Used (words) = 86

<>DAP                  { display all processes }
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
{ only IDLE process active }

<>QUIT

Execution Terminated
Memory Used (bytes)  Maximum = 3012  Current = 1246
```

## 6.3 DEBUGGER COMMANDS

A Host Debugger command is similar to a procedure call in Microprocessor Pascal or AMPL. (AMPL "A Microprocessor Prototyping Lab", is a useful tool for debugging at the the target machine level.) The following constraints must be observed:

- A debugger command name is followed by a possibly empty list in parenthesis. (NOTE: if the list is empty, then the parenthesis are not used.)

- Parameters are separated by commas (as in a normal MPP system call).

- Only one debug command may appear on an input line. Continuation of a command across an input line boundary is not allowed.

Command names may be written using either upper-case, or lower-case characters; 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: [ ] are used to indicate that the enclosed symbol is optional. Items within angle brackets < > are defined by the user.

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, so in this case, the ( ) could be empty, i.e., the command becomes SF.

```
                        SM ( <address>, [<length>] )
```

This is the syntax for the SM (Show Memory) command. The first parameter must be present; 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 single quotes (a single quote is represented by two single 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

6.3.2.1 Process. Debugger commands definitions make extensive use of two special kinds of parammeters: process parameters and routine parameters.

A process may be represented at the source code level by a SYSTEM, PROGRAM, or PROCESS. The term process is used to refer to a specific entity that "owns" a set of resources and performs some succession of computations. See also Paragraph 2.4.3.

A process parameter specifies a process, either by name or by a unique positive integer assigned by the debugger. A process name, when given as a parameter, must refer 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.

6.3.2.2 Routine. A routine is considered to be any executable sequence of source statements that is delineated by a BEGIN-END pair. The term routine can be used to refer to the body of a SYSTEM, PROGRAM, PROCESS, PROCEDURE, or FUNCTION.

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.

For example, one can 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.

TABLE 6-2. FORMATS FOR ROUTINE PARAMETERS.

(Note: EXAMPLE is process number 4 and has been selected as the default process.)

| | | EXAMPLE |
|---|---|---|
| (1) | process name.routine name | EXAMPLE.DUMMY |
| (2) | process name.routine number | EXAMPLE.3 |
| (3) | process number.routine name | 4.DUMMY |
| (4) | process number.routine number | 4.3 |
| (5) | routine name | DUMMY |
| (6) | routine number | 3 |

Each of the example routine parameters in Table 6-2 refers to the same routine. 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 5 and 6), 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, the routine´s 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:

[ <process>. ] <routine>

where the <process> parameter (along with the period separator) is optional, if the process parameter identifies the default process.

## 6.3.3 Optional Parameters

If a parameter is optional, it can 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)". The latter command results in showing 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 may be followed by a set of 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)".

## 6.4 CONDUCTING A DEBUGGING SESSION

### 6.4.1. Starting A Session

If the Debug option has been placed in the compiler code, the debug session may be initiated after compiler execution is complete by entering the command: DEBUG, followed by a RETURN/NEW LINE. The Debugger then asks the question: "Do you wish to debug the most recently compiled system?". If the response is YES, the 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 (See Paragraph 6.3.4.5). Therefore, it is not necessary to recompile a system each time it is to be debugged, provided the object is saved. (Note: that debugging a saved file uses up less space than debugging a file generated as the last program compiled.)

For relatively large and complex systems it is advantageous to divide the system into segments that can be separately compiled and saved (see Paragraph 5.2.5). 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" references from one code segment to another. The diagram in Figure 6-1 illustrates the alternative debugging strategies.

NOTES ON FIGURE 6-1:

The rectangular shapes represent Microprocessor Pascal Development System commands. Each rhomboid shape represents 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.

The boxes above the dotted line illustrate a single system from the EDIT phase through the DEBUG phase.

The SAVE command box illustrates that interpretive code can be collected into a library of interpretive code segments 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 Figure 6-1, 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, if the program to be debugged is the previously compiled and saved system, the user need not use the LOAD command.

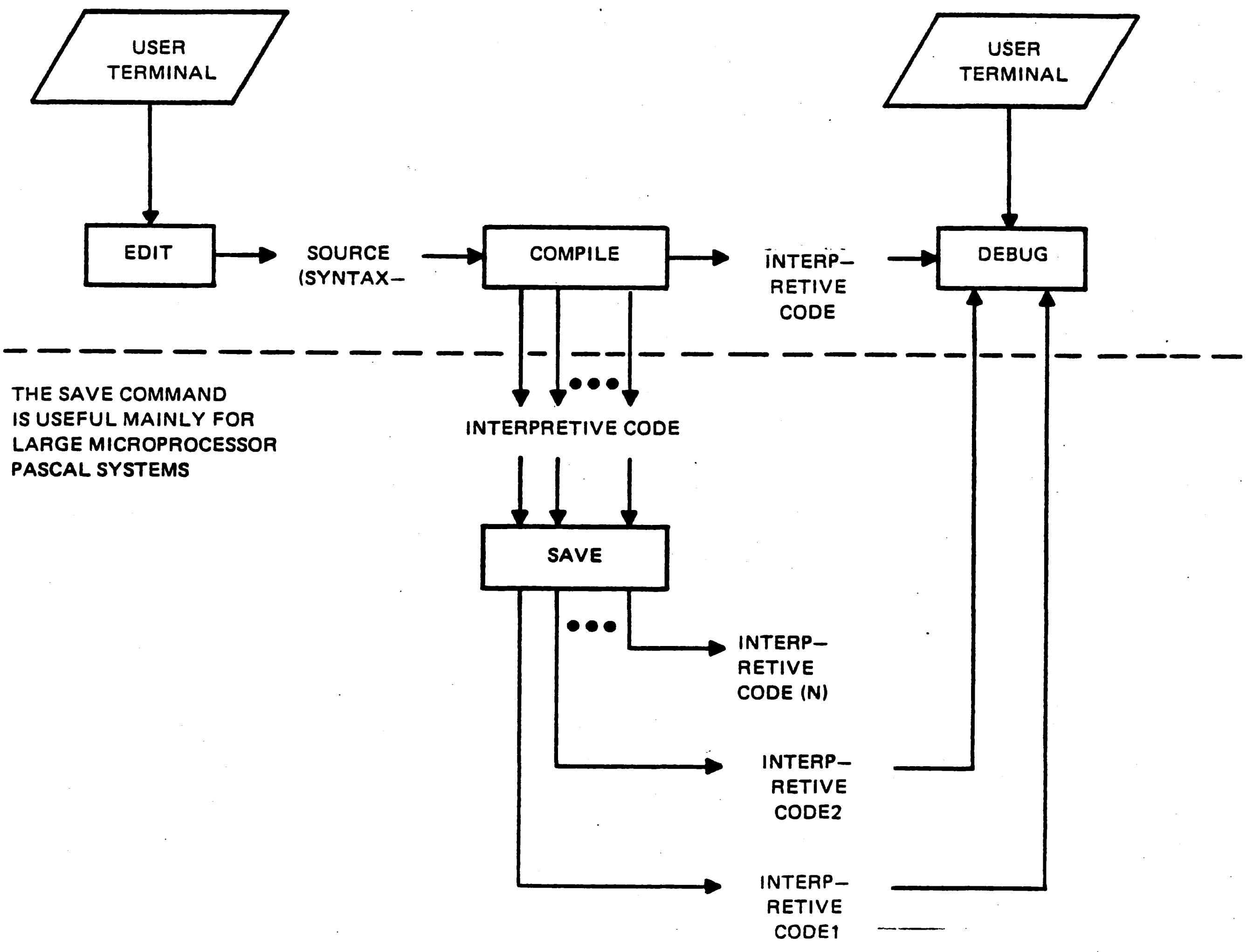


FIGURE 6-1. DEBUGGING STRATEGIES.

## 6.4.2 DEBUG Commands

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

6.4.2.1 GO Command. This command 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.

After the initial GO command has been given, a message should also 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.

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

```
IDLE INSTRUCTION
```

Execution of the IDLE INSTRUCTION places the processor in an idle state, where it remains until an interrupt occurs. When the idle process becomes active unexpectedly, the user may invoke the display commands DP and DAP to attempt to discover why no other processes are active. Processes that are supended pending an interrupt can be relaeased by using the SIMI command, which simulates an interrupt and allows execution to resume.

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

6.4.2.3 HELP Command. This command is used to display information concerning available debugger commands and their parameters. The syntax 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. (Under some host systems, only the latter function is allowed.)

6.4.2.4 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 is:

```
DEBUG ( <process name>,[<flag>] )
```

The process name parameter must be the name of a process; a process number is not allowed, as the process is yet to be createde (and the process number assigned) when the debug command in issued. 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 (or T and F). 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.

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

```
LOAD ( ´<pathname>´ )
```

The single pathname parameter is a string enclosed in single quotes. The pathname is the name of the file in which the code segment was saved. The LOAD command and the Microprocessor Pascal System´s SAVE capability provide a convenient means of separately compiling large and small systems. The LOAD command 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 the most recently compiled one, at least one LOAD command must be given to load a previously compiled system. Note that the first module loaded must contain either the body of the system to be debugged or a dummy system body (nullbody) with the same name as the system to be debugged.

6.4.2.6 SE Command. This command is used to show the names of any unresolved external routines. The syntax 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. The same name may be repeated in the list. Each entry corresponds to a reference to the given external routine.

6.4.2.7 COPY Commmand. This command is used to execute a series of commands from an external file. The syntax is:

```
COPY ( ´<pathname>´ )
```

The COPY command provides a convenient way to perform a sequence of frequently executed commands without having to individually enter the commands from the terminal. The single pathname parameter is a string enclosed in single quotes. The pathname is the name of the file in which the sequence of commands is written. 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 in a 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.

### 6.4.3 Status Display Commands

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. These commands operate on a user specified process or the default process.

6.4.3.1 <u>Display All Processes Command (DAP)</u>. This command lists the status of every process currently known to the system. Consider the following Table:

---

TABLE 6-3. STATUS SUMMARY OF ALL EXISTING PROCESSES.

| PROCESS NAME | | SITE OF EXECUTION | STATUS | PRI | ENABLED TRACES | STMT 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 |

---

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 that require a process parameter.

The next column contains process names. Each of the process names are indented to shows its static lexical nesting level.

The site of execution indicates a routine name and statement number unless the process is currently executing in run-time support code. In the example, the site of execution displayed is "runtime code".

The status column indicates the current status of each process as follows. "Active" identifies the currently active process. "Ready" indicates the process is ready for execution. The status of a waiting process is indicated as "Wait" and identifies the reason for the wait. These reasons include "Wait File" (waiting for file management services), "Wait Mem" (waiting for memory management services), "Wait Sema" (waiting on a semaphore), or "Wait Prcs" (waiting for process management services). The status "Hold" indicates use of the Hold Process command (HP) to temporarily hold the process from normal scheduling. If the process cannot be immediately held (e.g., the process is waiting on a semaphore), the status displayed ("Wait Sema") is followed by "H" indicating the pending hold. The default process is indicated by the character ">" which immediately preceeds the status.

In the PRI column, the priority is listed for 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 or not statement breakpoints are set for each process.

6.4.3.2 <u>Display Process Command (DP)</u>. This command displays detailed status of a single process. The syntax of this command is:

```
                              DP ( [<process>] )
```

If <process> is not specified, the default process is assumed. Consider the following example:

```
        ( STATIC/DYNAMIC CALLING ORDER FOR PROCESS CSXIN(2) )

     Stack size (words) = 1484
     Stack used (words) 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 name and number of the process being displayed.
The second line shows the maximum amount of stack space available for the process. The third line indicates how much of the stack has been used and how much is currently being used. The body of the display names all routines nested within the process and all ancestor processes.
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 number 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 displayed before routine number 1 are ancestor processes. The ancestors of a process must be displayed since their stack frames are accessible. Another DP command can be given to display the status of the desired ancestor.

### 6.4.4 Breakpoints/Single Step

Microprocessor Pascal source statements are numbered by the compiler; the compiled code is supplied with these numbers if the DEBUG compiler 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 a message:

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

"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 can be breakpointed at different statements according to 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 not currently on all host development systems.) When the CMD key is pressed, the debugger displays the following message before interacting with the user for further commands:

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

6.4.4.1 Assign Breakpoint (AB). 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 (routine name or call number obtained via DP command) identifies the routine which contains the breakpoint. The <statement number>, if given, specifies at which statement to breakpoint; the default statement number is 1. (REMEMBER: the breakpoint is exclusive to the specified process/procedure.)

6.4.4.2 Delete Breakpoint (DB). This command is used to delete a statement breakpoint from any routine. The syntax is:

```
DB ( <routine>, [<statement number>] )
```

The <routine> parameter (routine name or call number obtained via DP command) identifies the routine which contains the breakpoint. The <statement number>, if given, indicates which breakpoint to delete; the default statement number is 1.

6.4.4.3 Delete All Breakpoints (DAB). This command is used to delete all breakpoints from any process in the system. The syntax is:

```
DAB ([<process>])
```

The <process> parameter specifies the process in which breakpoints are to be deleted. The default is the "Default Process".

6.4.4.4 List Breakpoints (LB). This command is used to list all breakpoints set in the specified process. The syntax is:

```
LB ( [<process>] )
```

The <process> parameter (identification number obtained via DAP command) is optional. If no process is given, the default process is assumed. The list displays the process name, routine name, and statement number for each breakpoint set in the specified process.

6.4.4.5 Single-step Mode (SS). This command is used to perform single-step execution. The syntax is:

```
SS ( [<process>], [<flag>] )
```

The <process> parameter (process name or identification number obtained via DAP command) is optional. If no process is given, the default process is assumed. The <flag> parameter must be the Boolean value (TRUE or FALSE,or T or F). A TRUE value turns on the 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 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.

## 6.4.5 Showing/Modifying Data Commands

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 an SF (show frame) command. The format of the output in this example is similar to the output format of the other show variable commands. The stack frame in the example is 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 display is split into "words". (A word is sixteen bits and is displayed as four Hex digits.) The first word of every display line is the absolute memory address of the first data word displayed on that 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 periods. Notice that the only printable characters in this example are those for the string "SAMPLE" which starts at displacement 6 ( >5341 on the first line).

6.4.5.1 Show Frame Command (SF). This command is used to display a stack frame or some part of a stack frame. The syntax 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 <routine> parameter specifies the routine for which the stack frame is to be displayed (this parameter can be identification number obtained via the DAP command). 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

entire common area is displayed. If the <displacement> and <length> extend beyond the length of the common, only the bytes in the specified common are displayed.

6.4.5.4 Show Indirect Command (SI). This command is used to display a value arrived at indirectly via a pointer contained by a variable (similar to indirect addressing used by the assembly language programmer). The syntax

```
SI ( <routine>, <displacement>, [<length>] )
```

The <routine> parameter (which can be specified by name or call number) is the routine to which the variable parameter belongs. The <displacement> parameter is the byte displacement of the variable in the given routine stack frame. The <length> parameter is the number of bytes to be displayed (length defaults to 1).

6.4.5.5 Show Memory Command (SM). This command is used to show the contents of an absolute memory location. The syntax is:

```
SM ( <address>, [<length>] )
```

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

6.4.5.6 Modify Frame Command (MF). This command is used to modify a single word value in the specified stack frame. The syntax 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. 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.

6.4.5.7 Modify Heap Command (MH). This command is used to modify a single word value in heap. The syntax 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.

6.4.5.8 Modify Common Command (MC). This command is used to modify a single word in a common. The syntax 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.

6.4.5.9 Modify Indirect Command (MI). This command is used to modify a single word indirect variable (VAR parameter). The syntax

```
   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.

6.4.5.10 Modify Memory Command (MM). 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 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.

### 6.4.6 Tracing Commands

Tracing commands are available for tracing functions, examining the behavior of the scheduling algorithm, observing the dynamic behavior of routine calls and exits, and determining 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 convenience, however, the display of trace data at the terminal can be suppressed using the TOFF (Trace OFF) command and enabled again using TON (Trace ON). Each of the trace commands are described below.

6.4.6.1 Trace Process Command (TP) This command turns process scheduling tracing on or off for the specified process. The syntax is:

```
                     TP ( [<process>], [<flag>] )
```

The <process> parameter is optional and may be either the process name or the identification number obtained via the DAP command. If <process> is omitted, the default process is assumed. The <flag> parameter must be one of the Boolean values TRUE or FALSE (or T or F). A TRUE value enables tracing; a FALSE value disables tracing. If <flag> is omitted, 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. An example follows.

```
            .
            .
            .
    *** 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
            .
            .
            .
```

6.4.6.2 <u>Trace Routine Entry/exit Command (TR)</u>. This command turns routine entry/exit tracing on or off for the specified process. The syntax is:

```
TR ( [<process>], [<flag>] )
```

The <process> parameter is optional. If <process> is specified, the default process is assumed. <Flag> 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 <flag> 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. An example follows.

```
            .
            .
            .
    *** 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
            .
            .
            .
```

6.4.6.3 <u>Trace Statement Flow Command (TS)</u>. This command turns statement execution tracing on or off for the specified process. The syntax is:

```
TS ( [<process>], [<flag>] )
```

The <process> parameter is optional and may be either the process name or identification number obtained via the DAP command. If <process) is omitted, 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, a new line is written to the trace file each time a statement instruction is encountered (statement instructions only exist in routines compiled with the DEBUG option). 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. An example follows.

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

        .
        .
```

6.4.6.4 Trace Echo Off Command (TOFF). This command prevents 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. By using this command, it is possible to obtain an execution trace of a system when it is impractical to examine a large amount of trace data interactively.

6.4.6.5 Trace Echo ON Command (TON). This command enables the display of all trace information at the terminal. Please note that terminal display of trace information is the default. The TON command has no parameters.

## 6.4.7 Monitor Process Scheduling

The commands described below provide the user with limited control over the scheduling of processes. The user can "hold" a process, temporarily blocking it from becoming active until he explicitly releases it, and can assign process breakpoints forcing a breakpoint immediately before a process becomes active (scheduled for execution).

6.4.7.1 Select Default Process Commnand (SDP). This command is used to select the default process. The syntax 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 most recent instance of the process. A number may be given to select a former 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 having process parameters; this provides a convenient shorthand notation for later 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. Note: only a previously created process can be selected as the default process. Also, since many commands allow implicit reference to the default process, a default process to exist at all times. For this reason, whenever the default process terminates, the currently active process is chosen as the new default process by the debugger and the user is so notified. The message output when this is the case is:

```
*** 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.

6.4.7.2 Assign Breakpoint To Process Command (ABP). This command is used to set a process breakpoint. The syntax is:

```
ABP ([<process>])
```

If the process parameter is not specified, the default process is assumed; however, the parenthesis must always be specified. Process breakpoints persist until they are deleted using DBP or until the process terminates.

6.4.7.3 Delete Breakpoint From Process Command (DBP). This command is used to delete a process breakpoint before a specified process. The syntax is:

```
DBP (<process>)
```

where <process> can be specified by name or identification number (obtained via the DAP command), or may be the default process. If the process parameter is not specified, the default process is assumed; however, the parenthesis must always be specified.

6.4.7.4 Hold Process Command (HP). This command is used 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 is:

```
HP ( <process> )
```

where <process> can be specified by name or identification number (obtained via the DAP command), or may be the default process. If the process parameter is not specified, the default process is assumed; however, the parenthesis must always be specified.

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. While, the HP command does not take affect until the semaphore is signaled, this delay remains transparent to the user.

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

```
RP ( <process> )
```

where <process> can be specified by name or identification number (obtained via the DAP command), or may be the default process. If the process parameter is not specified, the default process is assumed; however, the parenthesis must always be specified.

### 6.4.8 Interprocess File Simulation Commands

Executive RTS interprocess files are simulated by the debugger with extensions allowing host files and devices to be connected to RTS channels as producers or consumers of components (See Section 10). Two debug commands are provided with which the user may specify a run-time mapping of internal channel names to external host file names. At debug time, the user can utilize the Connect Input File (CIF) and Connect Output File (COF) commands to specify that a particular host file is to a act as a producer and/or consumer to a particular channel. The functions SETNAME and FILENAMED should not be used for this purpose.

6.4.8.1 Connect Input File command (CIF). This command allows the user to specify a particular host file 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 previously connected reading files become disconnected (by calling RESET, REWRITE, or CLOSE). 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 is:

```
CIF ( ´<INTERNAL CHANNEL NAME>´, ´<input host file name>´ )
```

NOTE: the INTERNAL CHANNEL NAME may not be more than 7 characters.

Both the <internal channel name> and <input host file name> are string parameters and, as such, must be enclosed in eithr single quotes or double quotes. The <internal channel name> may be no more than eight characters long. The <input host file name> identifies the file to be connected to the specified channel 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"; i.e., CIF (´input´, ´me´), then the user is prompted for input whenever host requests are made.

6.4.8.2 Connect Output File Command (COF). This command allows the user to specify a particular host file 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 connects to the channel. If the channel already has files that write connected to it, the host connection takes place when all such connected files 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, as such, must be enclosed in ether single or double quotes. The <internal channel name> may be no more than eight characters long The <output host file name> identifies the file to be connected to the specified channel 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.

### 6.4.9 Interrupt Simulation Command (SIMI)

This command is used to simulate an interrupt at the specified level. The syntax 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).

### 6.4.10 Selection of CRU Mode Command (CRU)

The CRU debugger command is used to control how CRU instructions are handled. The syntax is:

```
CRU ( [<process>], <cru mode>)
```

The <process> parameter (process name or identification number obtained via the DAP command) 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 default mode for CRU instructions is DEBUG.

The CRU instructions, TB, LDCR, SBO, SBZ, and STCR are standard procedures that may be called at any point in the user's program. The interaction that results when each of these instructions is called are described below.

6.4.10.1 Test Cru Bit Command (TB). The following message for input is displayed:

```
"TEST CRU BIT" ADDRESS = nnnn, TRUE OR FALSE?:
```

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

6.4.10.2 Load Cru Value Command (LDCR). The 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.

6.4.10.3 Set Bit to Logic One Command (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.

6.4.10.4 Set Bit to Logic Zero Command (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.

6.4.10.5 Store Cru Value Command (STCR). The following message and prompt for input are displayed:

"STORE CRU VALUE" ADDRESS = nnnn, WIDTH = nn, VALUE?:

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

### 6.4.11 Ending A Session

The user may halt a debug session at any point by issuing a QUIT command.

## 6.5 ERROR MESSAGES

This section lists and explains debugger error messages. The messages are concerned with syntax, breakpoint, show/modify command errors, and miscellaneous errors.

### COMMAND SYNTAX ERRORS

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

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 proper command. This message usually represents 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 of 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 single 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 was not received (either TRUE, FALSE, T, or F). Use the HELP command to check which parameter are Boolean.

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 single quotes.

## 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 and states that no breakpoints were 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.

## 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 misspelled. 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; 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.

## MISCELLANEOUS ERRORS

The following errors are general in nature and as such are considered miscellaneous.

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.

NO INTERPRETIVE CODE FOUND; USE LOAD COMMAND

This is a warning message which indicates the default 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 warning message indicates that the module being loaded does not contain debugging information. If desired, the module should be recompiled and resaved 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 as yet been resolved. 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 contained 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
This error occurs when an attempt was made to service an interrupt and no waiting process exists.

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 INTERRRUPT
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. Contact your Texas Instruments' Regional Technology Center (RTC) if such an error occurs and its cause cannot be determined.

# SECTION 7

# CONVENTIONAL PASCAL PROGRAM EXECUTION

## 7.1 OVERVIEW

The Microprocessor Pascal System provides an executive that will load a conventional Pascal program and execute it without any interactive debugging capability. The program must be a single PROGRAM with no SYSTEM and/or PROCESSes and no Executive Run Time Support calls. Only standard Microprocessor 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

After the user enters the EXECUTE command, (and after responding to prompts for an output file), and memory), the executive brings up the following prompt:

DO YOU WANT TO EXECUTE THE LAST PROGRAM COMPILED?

The default response is yes (the user simply presses the RETURN key to give the default response). In response to a "No" reply, the following prompt is displayed:

ENTER PATHNAME OF MAIN SEGMENT:

To resolve any external references in the program to be executed, the execute program prompts the user for the pathname of each required external segment:

ENTER PATHNAME OF EXTERNAL SEGMENT:

If the external references are to be ignored, no pathname is specified and the RETURN key is pressed.

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 used as needed for either stack or heap. In the DX version, the combined amount of stack and heap is requested automatically in the user program (see Subsection 8.2.13). 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 as indicated by the following message:

```
EXECUTION BEGINS
```

After the user´s program has completed execution normally, control is passed back to the executive and the following message is displayed:

```
NORMAL USER PROGRAM TERMINATION
```

If the user´s program terminated abnormally, the following message preceeds the stack and heap utilization message:

```
ABNORMAL USER PROGRAM TERMINATION
```

## 7.4 I/O SUPPORT

The user´s program may utilize I/O supported by Microprocessor Pascal to access host files. All Microprocessor Pascal files are supported including TEXT, SEQUENTIAL, and RANDOM files (see paragraph 8.4.2.4.).

The destination of the output file is specified during the initial prompt (following entry of the EXECUTE command). All other files required by the program may be connected to the host file via the SETNAME procedure (see Appendix C, paragraph C.2). 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:

```
´filename´ FILE:
```

Note: The pathname may contain synonyms when running under DX/10.

## 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 - NAME= filename
```

where "ee" is either the I/O error type (described in Appendix E) or 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.

Normally, when an error is found by the run time support routines, the user's program is halted and control is passed back to the execute program. When this happens, 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 indicating the type of the error that 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 - an unresolved procedure was called.

DIVIDE BY 0 - a divide by 0 was attempted.

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 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 appears, a trace back message is generated. The first line of the trace back message contains:

- the name and number of the routine in which the error occurred if the program was loaded from interpretive code or from a segment saved with debug information. For example: ´ROUTINE´ name STATEMENT NUMBER = nn

or

- the words ´RUN TIME SUPPORT´ if segment was not saved with debug information.

Under the routine name (or "RUN-TIME SUPPORT" header), information is presented indicating how the routine was called.

# SECTION 8

# MICROPROCESSOR PASCAL SYSTEM LANGUAGE

## 8.1 OVERVIEW

Texas Instruments has built language extensions into the basic Wirth Pascal language, customizing it to support the software development and the run-time executive features of the Microprocessor Pascal System.

This section examines T.I´s Microprocessor Pascal System language. The purpose of the information presented is to define the various elements of this language and the structures comprised by these elements. In order to learn how to program with Pascal (or any other high-level language), the user should consult one of the many books specifically published for that purpose.

## 8.2 LANGUAGE VOCABULARY AND REPRESENTATION

The Microprocessor Pascal System language is composed of symbols. These symbols are made up of various combinations of elements from the language´s character set. In turn, these symbols comprise the language vocabulary. This vocabulary consists of identifiers, numbers, strings, operators, and keywords. The Microprocessor Pascal System character set and vocabulary are defined below.

### 8.2.1 Character Set

The Microprocessor Pascal System character set consists of the letters A - Z and a - z, the digits 0 - 9, and the special characters

```
+ - * / " . , ; : = ´ < > ( ) [ ] { } # ^ @ _
```

### 8.2.2 Special Symbols

The special characters listed above are used to make up special symbols. Special symbols are used for operators (logical and arithmetic) and delimiters. These special symbols are as follows:

```
+   -   *   /   :=   =   <>   <   <=   >=   >   ::  @
(   )   [   ]   {   }   ..   .   ,   ;   :   ´   "   # ^
```

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.

### 8.2.3 Keyword Symbols

Keyword symbols are reserved words with fixed meanings; they may not be declared as identifiers (see Paragraph 8.2.4). Each keyword is composed of letters and is interpreted as a single symbol. These keywords are listed below.

| | | | |
|---|---|---|---|
| ACCESS | ELSE | MOD | REPEAT |
| AND | END | NIL | SEMAPHORE |
| ANYFILE | ESCAPE | NOT | SET |
| ARRAY | FALSE | OF | START |
| ASSERT | FILE | OR | SYSTEM |
| BEGIN | FOR | OTHERWISE | TEXT |
| BOOLEAN | FUNCTION | OUTPUT | THEN |
| CASE | GOTO | PACKED | TO |
| CHAR | IF | PROCEDURE | TRUE |
| COMMON | IN | PROCESS | TYPE |
| CONST | INPUT | PROGRAM | UNTIL |
| DIV | INTEGER | RANDOM | VAR |
| DO | LABEL | REAL | WHILE |
| DOWNTO | LONGINT | RECORD | WITH |

### 8.2.4 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 the $ or _ symbols. Upper and lower-case letters are allowed but a lower-case letter is treated as the corresponding upper-case letter (e.g., the identifier DATA_SIZE is the same as the identifier Data_Size). Identifiers may not cross cardline boundaries and thus may not be more than 72 characters in length. 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 its first six characters. To avoid conflict with any run-time support routines, the names of user-defined routines should not possess 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 - )
```

Please note that some identifiers are standard, i.e., they are predeclared with a given meaning. However, identifiers may be redefined by the user with the result that the standard meaning no longer applies. For example, redefining the name associated with a standard routine prevents the standard routine from being called.

## 8.2.5 Separators

Separators are spaces, ends of lines, comments, or remarks that are used to offset Pascal language elements. For example in

```
WHILE X<10
```

a space separates WHILE and X. It is not equivalent to write:

```
WHILEX<10
```

At least one separator must occur between any two constants, identifiers, keywords or special symbols. However, no separator may occur within any of these elements.

A comment is special case of separator. A comment is any sequence of characters beginning with (* or { and ending with *) or }. However (* 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 line.. However, " appearing within a string does not begin a remark. Examples follow:

```
{ This is a comment }
(* This is also a comment *)
" the rest of the line is a remark
STRING: = '"This is not a remark';
```

## 8.3 DATA

Data refers to the information manipulated by a computer program. In the Microprocesor Pascal System data can be constants or variables.

## 8.3.1 Constants

Constants are named entities that do not change this value within a system. Constants may be integer constants, long integer constants, real constants, string constants, character constants, or Boolean constants.

8.3.1.1 Integer and Long Integer Constants. An integer constant is written as a sequence of decimal digits or a sequence of hexadecimal digits preceded by a # sign. Either form may be followed by the letter "L" to indicate a LONGINT constant (A long integer refers to a larger range of whole numbers than does integer--see Paragraph 8.4.1.2.)

Examples:

```
        Legal INTEGER and LONGINT constants
             133
             #26B
             #AFL
             00022
             252410L
```

8.3.1.2 Real Constants. Real constants can be written as sequences of decimal digits separated by a decimal point, or with the use of exponential notation. In the use of exponential notation, the decimal point must be surrounded 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 )
```

8.3.1.3 String Constants. A string constant is written as a sequence of characters enclosed by apostrophes. The length of the string may from 2 to 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 defined as a:

```
        PACKED ARRAY [1..length] OF CHAR
```

where length is the number of characters in the string. NOTE: length must be 2 or greater.

Examples:

```
        Legal STRING constants
              '  THIS IS A STRING  '
              'UNPRINTABLE CHARACTER #0D'
              'EXAMPLE ##3'
              'CAN''T'
```

8.3.1.4 Character Constants. A character constant is written as one character enclosed by apostrophes. The character may be represented by two hexadecimal digits preceded by a # (its ASCII code). As in a string constant, the character ' is represented by '' and # is represented by ##.

Examples:

```
        Legal CHARACTER constants
              '7'
              'P'
              ''''
              '+'
              '#0F'
```

8.3.1.5 Boolean Constants. A Boolean constant is declared as either TRUE or FALSE.

## 8.3.2 Variables

Variables are data structures that change their values during execution. Every variable has a type associated with it. The type determines the values the variable may assume and the operations that may be performed on the variable itself. Types, their meanings and representations are discussed in Paragraph 8.4.

A variable may be either a simple identifier which represents an entire variable or a qualified variable which represents a component of a variable of a structured type (see Paragraph 8.4.2).

8.3.2.1 Simple Variables. A simple variable is an identifier that references the entire variable. The form of the simple variable is as follows:

```
           <variable identifier>
```

Identifiers are defined in Paragraph 8.2.4.

8.3.2.2 Qualified Varaibles. As previously stated, qualified variables represent the component variables of structured types. These qualified variables are made up of variable identifiers. The way in which these variable identifiers are used as qualified variables depends upon the structure of the type being referenced. These qualified variables include array variables, record variables, and pointer variables. These variables will be discussed in paragraph 8.4 which details data types.

## 8.4 DATA TYPES

Data types are associated with variables. The data type defines the values a variable may assume and the operations that can be performed on the variable. Each variable is associated with one and only one data type.

A variable may be either a simple type or a structured type. The simple types consist of the standard types INTEGER, LONGINT, REAL, BOOLEAN, and CHAR; plus user-defined scalar or subrange types. Structured types consist of variables having more than one component. For structured types, the user must indicate the types of the components and the 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 Paragraph 8.5.3.

### 8.4.1 Simple Types

Simple types consist of the standard types INTEGER, LONGINT, BOOLEAN, CHAR, or REAL, or user-defined scalar or subrange types. In addition, these simple types make up a special class of type called an enumeration type. Enumeration types and the various simple types are described below.

8.4.1.1 Enumeration Types. With the exception of type REAL, the simple types support enumeration. Enumeration means that the variables have a set of 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 the assignment operator (:=) and the relational operators described below:

| | |
|---|---|
| < | less than |
| = | equal |
| > | greater than |
| <= | less than or equal |
| <> | not equal |
| >= | greater than or equal |

Three standard functions can be applied to enumeration types:

| | |
|---|---|
| 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) |

8.4.1.2 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 belongs to a set of whole numbers ranging 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 | integer divide (divide and truncate) |
| MOD | modulus -> A MOD X = A - ((A DIV X) * X)) |

Please note that the definition of the MOD operator (above) supports its use in the Pascal language; it is not the standard math definition.

The standard functions applying to arguments of type INTEGER and LONGINT are:

| Function | Value | Result Type |
|---|---|---|
| ABS(X) | Absolute value of X. | INTEGER (LONGINT) |
| SQR(X) | X * X | 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 an unpacked variable. 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.

8.4.1.3 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.

8.4.1.4 Char Type. A variable of type character has a value which is a printable character. This value 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. |

8.4.1.5 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. The order in which these identifiers are written defines the order of the scalar 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).

8.4.1.6 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 Paragraph 8.4.1.5, is used as the enumeration type for the subrange WORKDAYS.

8.4.1.7 REAL Type. The type REAL may be used to represent REAL numbers. The range of absolute values that can be represented is approximately 1.0E-78 to 1.0E75. However, the user can only expect precision in representations of up to six significant figures (a 24-bit mantissa is involved).

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 |

The standard functions, LTRUNC and LROUND, yield a result of type LONGINT.

## 8.4.2 Structured Types

Structured types are made up of components of other types. Structured types include arrays and records (which can either be PACKED or UNPACKED), sets, files, pointers, and semaphores.

8.4.2.1 Array Type. An array is an ordered collection of variables all of which are of the same type. Arrays are declared as follows:

```
ARRAY [ <index type list> ] OF <component type>
```

The <component type> may be of any simple or structured type except FILE type (see Paragraph 8.4.2.4). 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 of One-Dimensional Arrays:

```
VAR VECTOR, ARRAYROW: ARRAY [1..10]OF REAL;
     SICKDAYS: ARRAY [DAYS] OF BOOLEAN;
```

A component of an array is specified by the name of the array followed by an expression of the <index type> defining the component's relative position within the array. For example, referring to the one-dimensional arrays declared above,

ARRAYROW[3] specifies the third component of the array ARRAYROW. The value of this component can be any real number.

Similarly, referring to the other one-dimensional array SICKDAYS, when:

```
TYPE DAYS = (Mon,Tues,Wed,Thurs,Fri,Sat,Sun)
```

then

SICKDAYS[Tues] refers to the second component of the array SICKDAYS. The value of this component can be either True or False.

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 of Multi-Dimensional Arrays:

```
VAR TABLE:ARRAY [0..10, 10..20] OF INTEGER;
    BOOK:ARRAY [1..MAX, 1..80, 1..66] OF CHAR;
```

The only operator between array operands of compatible types 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 [1..<integer constant>] OF CHAR
```

Packed arrays are discussed in Paragraph 8.4.2.7.

The <integer constant> specifies the length of the string. It must be equal to or greater than 2. The number of characters in a string constant implicitly determines its type (the length of string); it can be up to 70 characters. The limit is 70 because line length is 72, and the string must be enclosed in quotes and not split across lines. 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 type string are assignment (:=) and the relational operators (<, =, >, <=, <>, >=).

Examples:
```
       TYPE STRG = PACKED ARRAY [1..6] OF CHAR;
       VAR WORD1,WORD2:STRG;
       BEGIN
       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] | {starting at element I of array A, move V-U elements to array Z, starting at element I.} |
| UNPACK(Z, A, I) | means: FOR J := U TO V DO A[J-U+I] := Z[J] | {starting at element 1 of array Z, move V-U elements to array A, starting at element I.} |

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
I is an integer where (N-M-I) > = (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, i.e., moves sufficient elements of anunpacked array to fill a packed array.

8.4.2.2 <u>RECORD Type</u>. A record type consists of a number of components of possibly different types. These components are 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 (see paragraph 8.4.2.4). The form of a record type definition is:

RECORD <field list> END

where <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). If present, the <record section> must be in the form:

<field identifier list> : <type>

where <field identifier list> is a list of field identifiers separated by commas. The word "END" must be placed after the last field in a record has been defined.

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;
       VAR VOLTAGE:COMPLEX;
           EXPIRE:DATE
```

A record variable is used to reference a field within a record. A record variable specifies the name of the record followed by the name of the field identifier (separated from each other by a period). For example, using the record declared above:

VOLTAGE.RE specifies the field "RE" in the record "VOLTAGE". The value of this variable must be a REAL number.

and

EXPIRE.YEAR specifies the field "YEAR" in the record "EXPIRE". The value of this variable must be an integer.

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. For example, 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;
```

A record may also have a variant part. The variant section is used to allow individual records to have some differences in their structure.

Based on the value of a field in the record (referred to as 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.

The <variant part> of a record has the following form:

```
        CASE <tagfield>:<tagtype> OF <variant list>
```

where <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). If present, the <variant> must have the form:

```
        <case label list>: ( <field list> )
```

A <case label list> is a list of <case label>s separated by commas. A <case label> is either a constant value or subrange value. The <case label> must be compatible with the <tagtype> for the variant section.

Example:
```
       TYPE CODE = (CREATE,CHANGE,DELETE);
       VAR 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: every field name in a record must be distinct, even those field names appearing in different variant parts of the record. See Paragraph 8.4.2.7 for a discussion on PACKED records.

An alternate way of implementing the case statement is to remove the tag field. In this way, the record length is shortened by one word. In the above example, the case statement would appear as follows:

```
           CASE CODE OF
```

8.4.2.3 Set Type. A set type is used to define variables whose values are sets. A set type specifies a base type; 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.

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 [ ].

A set is always based at zero. 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

NOTE: Based on the above definition, the set of INTEGERs is not valid.

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 Paragraph 8.4.1.5.

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 for these operations must be compatible sets (in the case of the IN operation, 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 . . .
```

8.4.2.4 File Type. A file type is a data structure made up of a sequence of components all of which are of the same predeclared type (e.g., standard, scalar, structured, etc.) The form of a file type declaration is one of the following:

NF;.NJ 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 a file´s components indicates the length of the file. This number is not fixed and may be increased according to the storage medium with which the file is associated.

Files are sequential files unless declared otherwise by the use of the prefix RANDOM. The components of a sequential file are accessed in their order of placement in the file.

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. TEXT is a predeclared type defined by:

```
TYPE
   TEXT:FILE OF PACKED ARRAY [1..80]OF CHAR;
```

INPUT and OUTPUT are standard predeclared TEXT files.

Examples:
```
TYPE REC =
       RECORD
         NAME:PACKED ARRAY [1..15] OF CHAR;
         ID_NUM:INTEGER;
       END;
     IFILE = FILE OF INTEGER;

VAR  EMPLOYEE:RANDOM FILE OF REC;
     TEMP:TEXT;
     F:IFILE;
```

A special file type predeclared to the system is the type ANYFILE. The type ANYFILE is used to pass file parameters to modules. When ANYFILE is used as a formal parameter, the actual parameter may be any file type (i.e., ANYFILE is a generic type).

An example of the use of ANYFILE as a formal parameter is demonstrated below.

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. Paragraph 8.8 contains more information about the various file types and how they are managed.

NOTE: the us of ANYFILE is restricted, in that the READ/WRITE standard procedures are called differently or RANDOM and SEQUENTIAL files; therefore, the code in the body of the procedure with the ANYFILE declaration has to be written explicitly for either RANDOM or SEQUENTIAL file I/O.

8.4.2.5 <u>Pointer Type</u>. Variables can be referenced indirectly by means of a pointer. A pointer can be thought of as an address. Microprocessor Pascal restricts this address to be the location of an object of the type specified in the pointer declaration.

```
     <pointer type>  =  @<type identifier>
 or  <pointer type>  =   <type identifier>
```

where <pointer type> is a pointer to variables of <type identifier> and <type identifier> must not be a file type (the type pointed to is usually a record type). The symbol @ or  identifies <pointer type> as a pointer. <Type identifier> need not be defined before the pointer type is defined, provided it is declared sometime later in the declaration section. (This is a forward type declaration; such declarations are only permitted with pointer types.)

Besides pointing to variables of the type declared, a pointer type can also point to the predefined constant NIL (i.e., the pointer points to nothing at all).

An example of the use of a pointer types follows:

```
        TYPE PTR = @LIST;       {PTR points to a record of type LIST}
             LIST =
               RECORD
                 VALUE:REAL;
                 LOC:0..#FF;
                 NEXT:PTR;
               END;
        VAR P : PTR
```

The variable pointed to by the <pointer type> is a pointer variable. This pointer variable takes the form of:

```
            <variable>
    or      <variable>@
```

The value of the pointer variable is undefined until a value is assigned to it (an address) or until a NEW is performed on it to allocate an area of dynamic storage. (As mentioned above, the constant NIL can also be assigned to it.)

Using the type and variable declarations given above:

```
        NEW(P)
        P@.VALUE := 13.5
        ...
```

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
```

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. In addition, pointer types are used when dynamically allocating data from the heap.

8.4.2.6 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 the counting semaphore. Operations on variables of type SEMAPHORE are performed by various functions and procedures which must be declared EXTERNAL by the programmer (see Paragraph 8.5.10 for EXTERNALs)

Aritmetic 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 (Paragraph 8.5.2.5). Information on how semaphores work is provided in Chapter 9.

8.4.2.7 Packed Types. Packing a data structure results the in storing (where possible) of several of its components in one data word. A structure is declared to be packed by placing the keyword PACKED before the word "ARRAY" or "RECORD" (the concept of packing is meaningless for other structured data types). Packing economizes the storage requirements of a data structure but may cause a loss in efficiency of access of its components, i.e., while data storage requirements are reduced, execution time and code size may be increased by use of the ´PACKED´ type.

An example of a packed type is a string type which is discussed in Paragraph 8.4.2.1. A string type is a packed array of characters. Passing an element of a packed structure to a procedure as a VAR parameter (by reference) is not permitted.

### 8.4.3 Size Algorithm for PACKED Types

If a type occurs in a packed structure, as much storage as specified by the size algorithm will be allocated to it, subject to the following restrictions.

- o Only enumeration types are packed.
- o Every structured type starts on a new word boundary and occupies an integral number of words or a fraction of a word.

The size algorithm specifies the internal representation of the value of a type in terms of bits or words. Thus, either a portion of a word or an integral number of words is allocated. As a result, 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 fraction of another word. Consequently, gaps of unused bits may occur. If a type occurs in an unpacked 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 the number required NR(N) to be the least value I such that N < 2**I (2 to the Ith power). Then the scalar type required R(N) bits.

EXAMPLE:

TYPE COLORS = [RED,ORANGE,GREEN,YELLOW];

In this case, N = 3 and the least value for I is 2 (i.e., 3 <2**2).

SUB-RANGE TYPE: Let L and U be the lower and upper bounds of the 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.

EXAMPLE:
TYPE MY_COLORS = [ORANGE..YELLOW]
In this case, U = 2 and the least value for I is 2 (i.e., 2 <2**2).

REAL: 2 words (24 bit mantissa, 8 bit (excess 64) exponent)

POINTER
TYPE: 1 word (16 bits)

ARRAY: One element, or at least one bit of an element (in the case of an element greater-than the size of a word), always falls to the left-most (Most-Significant) bit of the word.

All elements within an array are the same size.

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.

Examples: (Assume a 5-bit element (E5) and a 16-bit word)

(1) Array size: 1 element (not packed)

```
 0   4|5          15
|-----|-----------|
|  E5 |   Blank   |
|_____|___________|
```

(2) Array size: 3 elements (not packed)

```
0   4|5         15|0   4|5         15|0   4|5          15
|-----|-----------|-----|-----------|-----|-----------|
|  E5 |   Blank   |  E5 |   Blank   |  E5 |   Blank   |
|_____|___________|_____|___________|_____|___________|
|<---1st Word---->|<---2nd Word---->|<---3rd Word---->|
```

If the array is not packed, each element occupies the smallest number of words necessary to contain it, e.g., 5-bit elements requre 1 word each; a 17-bit or 29-bit element would require two words each, etc. Each element is right-justified within the words allocated to it, and is continguous. For example: (Assume a 21-bit element (E21)and a 16-bit word)

(3) Array size: 1 element (not packed)

```
 0           11-15|0                           15
|----------------|-----|--------------------|
|    Blank       |<----------E21----------->|
|________________|_____|____________________|
|<-----1st Word-------->|<----2nd Word----->|
```

If the array is packed, but the element size is greater than 8 bits, only one element may be placed within a single word. Each element will occupy one or more words. In this case, the elements are positioned as in arrays that are not packed. (See example 3, above)

If the array is packed and the element size is equal to or greater than 8 bits, the elements are packed as follows:

(4) Example: 8-bit elements packed (E8)

```
|0      7|8      15
|--------|--------|
|  E8    |   E8   |
|________|________|
|<----1 WORD----->|
```

(5) Example: 7-bit elements packed (E7)

```
|0|1      7|8|9      15
|-|--------|-|-------|
|B|  E7    |B|  E7   |
|_|________|_|_______|
|<-----1 WORD------->|
```

(6)Example: 6-bit elements packed (E6)

```
|0 |2     7|8 |      15
|--|-------|--|-------|
| B|  E6   | B|  E6   |
|__|_______|__|_______|
|<-----1 WORD------>|
```

(7) Example: 5-bit elements packed (E5)

```
|0   4|5   9|10-14|15
|-----|-----|-----|-|
| E5  |  E5 |  E5 |B|
|_____|_____|_____|_|
|<------1 WORD----->|
```

(8) Example: 4-bit elements packed (E4)

```
|0   |5   |9   |13
|----|----|----|----|
| E4 | E4 | E4 | E4 |
|____|____|____|____|
|<-----1 WORD------>|
```

(9)Example: 3-bit elements packed (E3)

```
|0  |3  |6  |9  |12 |15
|---|---|---|---|---|-|
| E3| E3| E3| E3| E3| |
|___|___|___|___|___|_|
|<-------1 WORD------>
```

(10) Example: 2-bit elements packed (E2)

```
|0 |2 |4, ETC..........15
|--|--|--|--|--|--|--|--|
|E2|E2|E2|E2|E2|E2|E2|E2|
|__|__|__|__|__|__|__|__|
|<--------1 WORD------->|
```

(11) Example: 1-bit elements packed (E1)

```
|0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|E1|
|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|
|<----------------1 WORD-------------------->|
```

RECORD: The size of 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, the packing algorithm works as follows:

1) If a record field is sixteen or more bits, it is allocated the next "unused" word(s) in the record and is right-justified. Any unused bits in the current word are left empty; any fields already in the current word are right-justified. For example:

(12) Assume the current word already contains a 6-bit field; The next record field contains 18 bits.

```
|------|----------|
|6 bits| Blank    |
|______|__________|
|<-Current Word-->|
```

Add 18-bit field:

```
 0               |0                |0
|---------|------|----------------|--|--------------|
| Blank   |6 bits|<---18 bit record >| Blank        |
|_________|______|________________|__|______________|
|<-Current Word->|<--Next Word--->|<----3rd Word--->|
```

2) If the record field is fifteen bits or less, and if there are sufficient unused bits in the current word, the field is placed, left-justified, in the unused bits of the current word. If there is insufficient space in the current word, the field is placed, left-justified, in the next word, and the right-most unused bits of the current word are shifted so that the rightmost field in the current word is shifted so that it is right-justified and the unused bits are somewhere in the center of the word. For example:

(13) Assume a record consisting of three fields: one field of 5 bits, one of 7 bits, and one of six bits.

Place 5-bit field:

```
|0                 15
|-----|-------------|
|5bits|  Blank      |
|     |             |
|___________________|
|<-Current Word---->|
```

Add 7-bit field:

```
|0                15
|------|-------|---|
|5 bits| 7 bits| B |
|______|_______|___|
|<--Current Word-->|
```

Add 6-bit field:

```
|0                   |0
|------|----|--------|----------|------|
|5 bits| B  |7 bits  | Blank    |6 bits|
|______|____|________|__________|______|
|<--Current Word---> |<---Next Word--->|
```

The complete record is now placed and occupies two words; the size of the packed record is four bytes.

Field lists within the variant part overlay 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.

### 8.4.4 Type Compatibility

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 explicitly associated with a type (as it is in a TYPE declaration). 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 (implicit conversion takes place when there is no explicit operator or function showing the conversion to be necessary).

## 8.4.5 Overriding the Type Structure

The type of a variable is meant to be an invariant property of that variable. However, it is possible to override the type of an existing variable using type-transfer.

A type transfer is implemented by the following code:

```
<variable>::<type transfer>
```

The <type transfer> is a type identifier and the variable is treated as the type given in the <type transfer>. A type transfer does not perform a value conversion; all that is altered is the apparent type of the variable, i.e., the bit pattern that is the value of the <variable? is treated as though it is the value of a variable of type <type transfer>. Also, the type transfer does not change the way the original variable is accessed and does not apply to any further accessing of that variable. For example:

```
TYPE BYTE = 0..#FF
     RECTYPE =
       PACKED RECORD
          MSBYTE,LSBYTE:BYTE

VAR V:ARRAY[0..9] OF INTEGER;
    R:RECTYPE;
```

Using the above type and variable declarations:

```
    R.MSBYTE := V[0]::BYTE
    V[1]::BYTE := R.LSBYTE
```

There is one restriction applying to type transfer: the size of the variable must be at least as large as the size of the type being transferred to; e.g., a packed byte (8-bit packed component)cannot be transferred to type INTEGER; an INTEGER must not be transferred to type LONGINT.

## 8.5 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 that may be declared are labels, constants, data types, variables, commons, programs, processes, and routines (procedures and functions). Declarations are used to associate unique names to each object. As a general rule, the identifier naming an object must be explicitly declared before it can 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, implicit declarations are used in Microprocessor Pascal System. Examples of implicit declarations are: FOR control variables, ESCAPE labels, and WITH variables (these are defined later in this section).

### 8.5.1 Scope

Each declaration has a scope, which is 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). Thus, once an identifier is declared for some object (e.g., a variable) that object can only be accessed by use of that identifier throughout the module (except when the identifier is redeclared within some inner unit of the scope module). Modules may only be declared to a maximum nesting level of ten.

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).

### 8.5.2 Extent

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 statically declared quantities is the duration of the execution of the unit of scope in which the quantities 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 these variables and the call (if any) to DISPOSE which frees the space allocated to them.

## 8.5.3 System Declaration

A Microprocessor Pascal System is the superstructure containing 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 Paragraph 8.5.12.

The <system data declarations> are described in Paragraph 8.5.3; 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;
```

### 8.5.4 Label Declarations

Label declarations specify all labels that 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.

### 8.5.5 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. These individual declaration parts are described in subsequent sections. The <system data declarations> are the same as the <data declarations> listed above except that <system data declarations> 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. NOTE: 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;              NOTE:   Within the procedure,
    VAR X:T;
    TYPE T = . . . ;
    VAR Y:T;
    . . .
BEGIN       { SAMPLE }
. . .
END;        { SAMPLE }
```

the variables X and Y may not be compatible, since the type T has been redeclared between the two variable declarations.

8.5.5.1 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, integer constant expression, or a previously defined constant identifier.

Example:

```
       CONST MAX = 100;
            ASTERISK = ´*´;
            ONE_HALF = 0.5;
```

The constant declaration part is the only place where integer constant expressions can be used in place of an integer constant.

8.5.5.2 TYPE Declaration Part. A TYPE declaration associates an identifier with a data type. Data types are discussed in Paragraph 8.4. A data type determines the set of values a variable of that type may assume, along with the set of basic operations that may be performed on the variables.

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 Paragraph 8.4.

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;
```

8.5.5.3 <u>VARIABLE Declaration Part</u>. 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;
```

The type VECTOR was defined in the example in Paragraph 8.5.3.2.

8.5.5.4 <u>COMMON Declaration Part</u>. A COMMON declaration is used to declare variables that can be shared with 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 that common identifiers must be unique within their first six characters.

### 8.5.6 ACCESS Declarations

An access declaration serves to identify all common variables that are to be referenced in the body containing the access declaration. Any common variable access 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. Thus, even in a module falling within the scope of an access declaration at a higher level, the common variable is not accessible 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.

### 8.5.7 PROGRAM Declarations

A program declaration specifies an independent process that does not share any global variables (with the possible exception of commons) with any other programs.

A program declaration is of the form:

```
   PROGRAM <identifier> (<program parameter> ;...<program parameter>);
     <program block> ;
```

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 (Paragraph 8.5.11). 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 8.5.1.

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;{# concurrent characteristics }
```

## 8.5.8 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 <program parameters> except <process parameters> may include pointer types (see Paragraph 8.4.2.5 for description of Pointer Types).

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;{# concurrent characteristics }

     BEGIN   " system body
     END;{# concurrent characteristics }
```

## 8.5.9 PROCEDURE Declarations

A procedure declaration specifies a routine that is invoked to perform an action; return is made to the calling routine once the action is completed.

A procedure declaration is of the form:

```
PROCEDURE <identifier> (<procedure parameter> ;...;<procedure parameter>);
  <procedure block> ;
```

where <procedure parameters> are optional.

<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.

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;
```

### 8.5.10 FUNCTION Declarations

A function declaration specifies a routine which is 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 also be supplied. 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 assigning a value to the function identifier or the function will return 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 error that is difficult to discover involves a function that not only returns a value through the function identifier, but also changes the value of non-local variables. An action that changes the value of a non-local variable is called a side-effect. The following rules can 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.

### 8.5.11 Parameter Kinds

There are two kinds of parameter passing:

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 referred to as call by value. This kind of parameter passing 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. This address is used by the called module to access the actual parameter indirectly. This form of parameter passing is known as call by reference. When the parameter is passed by reference, an assignment made to the formal parameter in the called routine changes the actual parameter in the calling module. Variable parameter substitution is specified by placing the reserved word VAR before the formal parameter section.

Value parameter transmission provides security against inadvertant changes to program values as well as an efficient way to pass simple variables as parameters. However, passing structured data such as large arrays by value would prove to be inefficent.

### 8.5.12 EXTERNAL Declarations

A program, process, procedure, or function can be declared to be externally defined. To do this, the identifier EXTERNAL is placed after the routine header, and the routine block is omitted from the listing. EXTERNALS are useful when separately compiled programs or processes 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;
```

### 8.5.13 FORWARD Declarations

A program, process, procedure, or function may be forwardly declared. This is necessary in direct or mutual recursion in which two or more modules call each other.

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 }
```

## 8.5.14 Concurrent Characteristics

The <concurrent characteristics> specify parameters concerned with multiprogramming. Such parameters indicate the memory requirements 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 <concurrent characteristics> as used in the <process body> (defined in Paragraph 8.5.1) must be coded in one of two ways:

```
             {# <concurrent characteristic list> }
                            or
            (*# <concurrent characteristic list> *)
```

where the <concurrent characteristic list> is one or more <concurrent characteristic>s separated by semicolons. Each <concurrent characteristic> is of the form:

```
        <concurrent keyword> = <concurrent value>
```

The <concurrent keyword>s and their associated meanings are given below.

| KEYWORD | MEANING |
|---|---|
| PRIORITY | Relative urgency of the system, program, or process |
| HEAPSIZE | Number of words allocated for the system, program, or process heap |
| 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 an 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: Implementation of stacks and heaps differs slightly between Interpretive and Native execution modes.

INTERPRETIVE MODE: A program or process stack and heap requirements are allocated out of the heap of the parent. The default values for stack heap are both zero; however, a heap size specification of zero implies the sharing of the parent´s heap. A program or process cannot give heap to a process it has started unless enough heap is explicitely allocated to the parent by a HEAPSIZE concurrent parameter. Figure 8-1 illustrates stack and heap allocation from parent to offspring.

NATIVE MODE: All available memory is allocated to the System. Program and process stacks and heaps are allocated directly from this heap, i.e., stacks/heaps are not nested within the parent´s heap. The default stack size parameter is defined in the CONFIG module and can be redefined by the user to whatever meets his requirements. The default for heapsize is to share the global heap with all other processes.

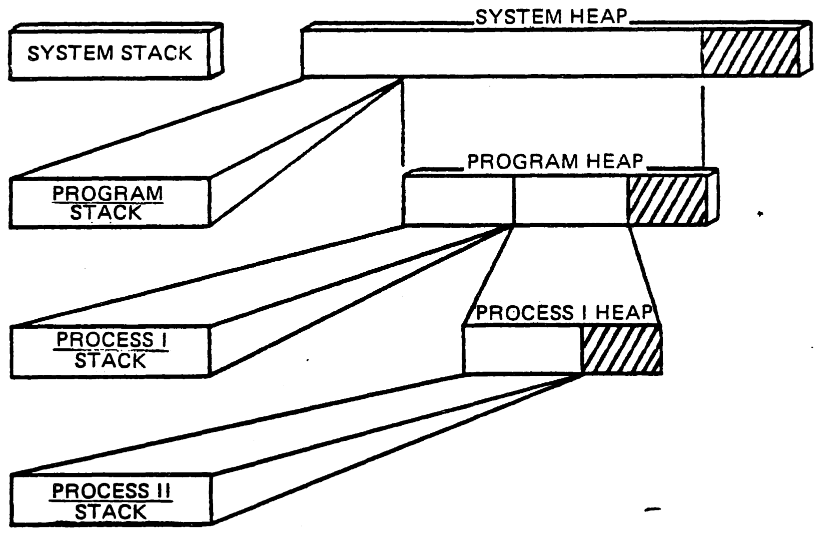


FIGURE 8-1. ALLOCATION OF STACK AND HEAP.

## 8.5.15 Conventional Pascal Program

A system may consist of a single conventional Pascal program. This program must be the only program (or process) in the system; no processes can be declared (nested) within it.

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 (or are even necessary). All parameters are supplied implicitly by the system. Also, when this form is used, no system header or system block need be given and concurrent characteristics are ignored.

## 8.6 EXPRESSIONS

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions are comprised of operands and operators.

### 8.6.1 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>
```

Additional information on set values and function calls is presented below.

8.6.1.1 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]
       [ ]
```

8.6.1.2 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> (<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 or INTEGER and the actual parameter of type INTEGER or LONGINT. Therefore, if a formal value parameter is of type REAL, an actual parameter of type INTEGER is converted to type REAL before it is passed.

## 8.6.2 Operators

An operator specifies an operation to be performed on one or two operands. An operator can only be applied to two operands if their data types are compatible i.e., of the same type. Some operators accept mixed types. In these cases, if one operand is of type REAL and the other is of type INTEGER or LONGINT, the latter is converted to REAL and thus the result is REAL. Also, if one operand is of type LONGINT and the other is of type INTEGER, the INTEGER operand is converted to LONGINT, and thus the result is LONGINT.

The "/" operator always produces a REAL result but may accept INTEGER operands. Also, the DIV and MOD operators 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 character 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 |

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]
        [ ]
```

8.6.1.2 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> (<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 or INTEGER and the actual parameter of type INTEGER or LONGINT. Therefore, if a formal value parameter is of type REAL, an actual parameter of type INTEGER is converted to type REAL before it is passed.

## 8.6.2 Operators

An operator specifies an operation to be performed on one or two operands. An operator can only be applied to two operands if their data types are compatible i.e., of the same type. Some operators accept mixed types. In these cases, if one operand is of type REAL and the other is of type INTEGER or LONGINT, the latter is converted to REAL and thus the result is REAL. Also, if one operand is of type LONGINT and the other is of type INTEGER, the INTEGER operand is converted to LONGINT, and thus the result is LONGINT.

The "/" operator always produces a REAL result but may accept INTEGER operands. Also, the DIV and MOD operators 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 character 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.

### 8.6.3 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 |

## 8.7 STATEMENTS

Statements describe the actions which a system performs on its data. Statements can be simple or structured; structured statements contain components consisting of other 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 (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.

### 8.7.1 Simple Statements

Simple statements contain no other statements. The various kinds of simple statements are defined below.

8.7.1.1 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>
```

This statement reads <variable> becomes <expression>.

Examples:

```
       X := A * NEXT DIV 2;
       FOUND := Z > TOTAL;    {boolean expression}
       COUNT := COUNT + 1;
       VALUE2 := SQR(VALUE);
       WORD := 'PASCAL';
```

Direct assignments may be made to variables of any type except files or semaphores. The type of <expression> must be compatible with that of the <variable> (type compatibility is described in Paragraph 8.4.3). The exception to this rule is in the case of an implicit conversion. An expression of type INTEGER can be implicitly converted to either LONGINT or REAL and an expression of type LONGINT can be implicitly converted to either INTEGER or REAL. For example, after writing

```
       VAR A : REAL;
           B,C : INTEGER;
```

you can write:

```
           A := B+C;
```

when B and C are combined, the integer result will be converted to a REAL number when stored in A.

Another use of the assignment statement is within the body of a FUNCTION where the function result is assigned to a variable that is the same as the function identifier in the calling routine. In this case, the type of the <expression> must be compatible with the result type of the function (see Paragraph 8 5 8)

8.7.1.2 <u>Procedure Statement</u>. 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 <parameter list> is a list of the actual parameters, separated by commas, which are substituted for the formal parameters declared in the procedure heading in the "Declarations" part of a Microprocessor Pascal System (see Paragraph 8.5.7).

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.

8.7.1.3 START Statement. A start statement is similar to a procedure statement except it invokes a program or process to execute concurrently within 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 (Paragraphs 8.5.5 and 8.5.6).

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.

8.7.1.4 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.

8.7.1.5 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, nor is it 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.

NOTE: The use of GOTO statements is not recommended; it is seldom (if ever) necessary to use them. The use of other Pascal control structures has been shown to be more useful in terms of basic software engineering principles.

8.7.1.6 ASSERT Statement. The ASSERT statement allows the programmer to test whether or not a condition is 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 compiler option ASSERTS is turned on, code is generated so that the <expression> in the ASSERT statement is evaluated when encountered at execution time. If the result is TRUE, execution continues; otherwise a run-time error occurs. If the compiler option ASSERTS is turned off, no code is produced for the ASSERT statement. The default value for the ASSERTS option is TRUE (i.e., turned ON).

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.

## 8.7.2 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 and REPEAT, and the WITH statement.

8.7.2.1 Compound Statement. A compound statement is a sequence of statements enclosed by the keywords BEGIN and END. The form of a compound statement is:

```
                BEGIN <statement list> END
```

where the <statement list> consists of zero or more statements, simple or structured, and separated by semicolons. The BEGIN and END act as delimiters around the 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;
```

Semicolons are used to separate statements in the compound statement. No semicolon is 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; i.e., an empty statement which specifies that no action is to be taken. Most empty statements do not alter the flow of statement control but the user should be wary of their use. For example,

```
       BEGIN
         SUM := X + Y + SUM;
         X := X + 5;
         Y := Y - 2;
         WRITELN(SUM);
         { <empty> }
       END;
```

8.7.2.2 IF Statement. The IF statement specifies that a <statement> is to be executed only if a given condition is TRUE; otherwise an alternative <statement> if present is executed. One form of an IF statement is:

```
             IF <expression> THEN <statement>
```

where the <expression> must be of type BOOLEAN and the <statement> may be simple or structured (e.g., compound, IF, etc.). If the <expression> evaluates to false, control passes to the next statement in sequence after the THEN clause. For example,

```
       IF COUNT <= MAX
         THEN READ(X[I]);
       X[I] := MAX + 1
```

Another form of the IF statement is:

```
        IF <expression> THEN <statement> ELSE <statement>
```

If the <expression> evaluates to TRUE, the THEN clause is executed; otherwise the second <statement> alternative, the ELSE clause, is executed. For example,

```
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 can 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;
```

There are no semicolons in an IF statement; it is wrong to put semicolons before the THEN or ELSE statements.

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.

8.7.2.3 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; however, if the value of the <expression? is not one of the <case label>´s, a runtime error will occur.

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;
```

8.7.2.4 <u>FOR Statement</u>. A FOR statement allows 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. Because the control variable is implicitely declared for the scope of the FOR statement, problems may occur if a variable of the same name is also declared in the procedure. For example:

```
VAR I.INTEGER;
 . . .
 . . .
 . . .
  FOR I=1 TO 10
 . . .
 . . .
```

In this case, the variable I declared as VAR is inaccessible within the scope of the FOR loop. In order to access both the control variable and procedure variables within the FOR loop, the control variable must have a different name to the procedure variables. this is also the case with nested FOR loops. For example:

```
FOR I:=1 TO 10 DO BEGIN
  FOR I:= 100 TO 200 DO BEGIN
  END;
```

Within the inner loop, the value of I will be 100 to 200 and the control variable for the outer loop is not accessible.

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;
```

8.7.2.5 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>. For example:

```
I := 1;
WHILE I <= MAX DO
  BEGIN
    VALUE := AMT[I] + TAX[I+2];
    I := I + 1
  END;
```

8.7.2.6 REPEAT Statement. A REPEAT statement allows a sequence of statements to be repeatedly executed as long as the specified condition is false. The 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 until the <expression> evaluates to true. The <expression> is evaluated after each execution of the <statement list>. For 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;
```

8.7.2.7 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 a record variable (see explanation in Paragraph 8.4.2.2). The 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. For 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;
```

Nested WITH statements of this form are also useful and may be written using a shorthand notation be replacing the <record variable> with a list of <record variables> separated by commas (as shown below).

Example of nested WITH:

```
TYPE DATE = RECORD
            MONTH: (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,
            DAY: 1..31;
            YEAR: INTEGER;

          END;

     INFO = RECORD PACKED ARRAY [1..10] OF CHAR;

            NAME:
            NUMBER:1..10;

          END;


VAR
  START_DATE:DATE;
  EMPLOYEE:INFO;

WITH START_DATE, EMPLOYEE DO

          BEGIN

            MONTH :=MAR;
```

```
                    DAY :=30;
                    YEAR :=1980;
                    NAME :=BOB;
                    NUMBER :=5;

                  END;
```

Note: Records must have separate field names.

A more reliable form of the WITH statement defines an <identifier> representing a synonym for the <record variable>. 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. For 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 use of WITH is more reliable because variables in the scope of the WITH statement must be entirely spelled out (eliminating confusion, for example, in a case in which a local variable has the same name as a field of a record).

## 8.8 INPUT AND OUTPUT

Statements are included in the Microprocessor Pascal System for accessing and manipulating files. These I/O statements are examined below for sequential, text and random files.

### 8.8.1 Sequential File Operations

Before values can be read from a sequential file (see Paragraph 8.4.2.4 for definition), a RESET statement must be executed. This statement opens the file for reading and causes I/O to start at the file's beginning. 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 reading a file via the Microprocessor Pascal READ statement, the component of the file being read is assigned to some user-declared variable. The type of the file component being read must be the same as the user declared variable it is being assigned to. This user declared variable cannot be an element of a packed structure. If the file is positioned for I/O at the end-of-file, nothing is read and an error exception occurs.

The general form of READ is as follows:

```
   READ (<file variable>, <variable>,..., <variable>)
```

where components of the file indicated by <file variable> are read into successive user <variables>. The component types of the file being read and the user <variables> must be the same.

A file to be written to must be opened for WRITE by the procedure REWRITE. When applying the Microprocessor Pascal statement WRITE to a file, a user declared variable of a type compatible with the component type of the file is used. The value of this user-declared variable becomes the next component of the file.

The general form of WRITE is as follows:

```
   WRITE (<file variable>, <variable>,..., <variable>)
```

where successive values of user <variables> are written into successive component of the file identified by the <file variable>. The component types of the file being written to and the user <variables> must be compatible.

Sequential file variables may be opened for reading or writing but not both simultaneously.

NOTE: Within the same system (or stand-alone program), when inputing a file from the host debugger, the file cannot be opened for writing then closed, without executing a CIF command. Neither can the file be reopened for reading unless a COF command is issued. (See also paragraph 6.3.10)

Examples of READ and WRITE follow.

```
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.
```

(NOTE: because only 1 record is specified, only 1 record is read, then written. As a result, the records in OLDCOPY are read into the record EMPLOYEE, and EMPLOYEE is written into record the next record of file NEWCOPY.

## 8.8.2 Text File Operations

Input and output data for many devices such as card punches, card readers, line printers, and CRT terminals are 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 text file.

Both input and output to and from text files are two-stage operations.

Writing to a file proceeds line by line. Ther are two statements for for writing to text files: WRITE and WRITELN. There are two forms of WRITELN: either with or without parameters.

The WRITE statement writes the file component values into a line buffer (write does not output directly to the file). The line buffer is output either when the buffer becomes full (80 characters), or when a WRITELN without parameters is executed. A WRITELN with parameters is equivalent to a WRITE with the same parameters, followed by a WRITELN without parameters.

SIMILARILY, when reading from a file, the READ statement takes file components from the line buffer and places them into variables. READLN also has two forms: with and without parameters. A READLN without parameters causes a new line to be fetched from the file and placed into the line buffer. A READLN with parameters is equivalent to a READ with the same parameters, followed by a READLN without parameters.

A RESET command must be used prior to the first READ or READLN statement to open a file for reading. The RESET does an automatic READLN. If this is not anticipated, the input might be shifted by one line from that which is expected.

When the last nonblank character of a line is read, a READLN operation changes the value of the EOLN function from true to false. For more details concerning the text file operations, see Paragraph 10.3.6.

Two standard text files 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; however, they must be declared if they are to be used within that program.

An example of reading and writing to a text file follows.

```
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.
```

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.

8.8.2.1 Text File READ Operation. A READ procedure statement has the form:

```
READ (<file variable>, <variable>,..., <variable>)
```

where <file variable> must indicate a text file and the user <variables> may be of types INTEGER, LONGINT, BOOLEAN, CHAR, REAL, or string (PACKED ARRAY[1..n] OF CHAR). The user <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 follow:

```
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 <variable> is of type CHAR, the value read is the next character in the text file (blanks are significant characters).
>
> If <variable> is of type BOOLEAN, the character T or F (or the standard identifier TRUE or FALSE) is read.
>
> If <variable> is of type string with length L, the next L characters are read from the text file (blanks are significant characters).
>
> If <variable> is of type INTEGER, LONGINT, or REAL, the next series of digits corresponding 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.
>
> 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
>
> ```
>    <variable>:M
> ```

where M is the field width.

8.8.2.2 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. Table 8-1 gives the default values.

TABLE 8-1. 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 field is not real and the value requires more than the specified width, the necessary additional space is allocated and used. If the field is real, a fatal RUNTIME error occurs.

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, 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 identifier HEX, the value is written as a string of hexadecimal 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.

### 8.8.3 RANDOM File Operations

A RANDOM file may be opened simultaneously for reading and writing by executing a REWRITE statement. A RANDOM file may be opened exclusively for reading by executing a RESET statement. The READ and WRITE procedures for a RANDOM file are similar to those for a sequential file, except for the inclusion of an argument that 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 variable is allowed in both the READ and the WRITE statements.

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>.

### 8.8.4 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´);
```

### 8.8.5 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.

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. (Note, when a file is passed by value, the file itself is not passed. Rather, a connection to the file is passed.) Thus, the file must be opened only in the process (RESET or REWRITE) in which it is used.

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 as variables in that program.

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´));
```

It should be noted that in the Debugger, host files cannot be connected implicitly. The CIF and COF commands must be used.

### 8.8.6 Encode and Decode

The procedures ENCODE and DECODE function similar to the text file 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. <Index> specifies an integer indicating where in the string is the value 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:

---

TABLE 8-2. ERROR CODES RETURNED BY STATUS.

| STATUS VALUE RETURNED | MEANING |
|---|---|
| 0 | No error |
| 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 Paragraph 8.8.2.2 for the WRITE procedure statement. The <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,

1) If STRG8 was the string ´12345678´ and STRG4 was the string ´ABCD´

```
    ENCODE ( STRG8,3, STAT, STRG4 : 4)
```

will redefine the STRG8 to the value ´12ABCD78´.

2) If N has the value 2 and CH has the value ´Z´

```
    ENCODE( STRG4,N, STAT, CH )
```

increments N to 3 and redefines the

string STRG4 to be ´AZCD´

3) If another application of the ENCODE statement is applied, STRG4 will have the value ´AZZD´.

4) If NUM=26 and STRG8 = ´R = >     ´

```
    ENCODE (STRG8,6, STAT, NUM:3  HEX)
```

redefines STRG8 to the value ´R = >01A´

5) If STRG8 = ´12345678´

```
    DECODE (STRG8, 3, STAT, NUM:2)
```

identifies NUM as 34.

6) If ST[illegible] 4 = ´ABCD´ and N=1, the first time

```
    DECODE (STRG4, N, STAT, CH )
```

is executed CH will have the value A, the second time the value B and so on.  An error code of 3 is passed back if one decodes (reads) beyond four characters.

# SECTION 9

# PROCESS SYNCHRONIZATION AND PROCESS MANAGEMENT

## 9.1 OVERVIEW

The Executive Run Time Support (RTS) supports multiprogramming (sometimes called multitasking). Multiprogramming refers to the interleaved execution of two or more routines (processes) in the same computer. If the processes executing in a Microprocessor Pascal System are viewed over a time frame of several seconds, each appears to be progressing at a non-zero rate. However, if the processor is followed for several milliseconds, its switching among several processes is apparent. Repetitive switching of processes causes interleaving of the processes´ progression through the algorithms and gives the impression that the processes are executing concurrently.

In a system with one sequential program, the central processor executes the program instructions sequentially. If execution is temporarily halted while the program awaits some system resource, the processor goes idle. The processor becomes active again only when the program is ready to resume execution.

The execution of concurrent processes in the Microprocessor Pascal System is different. Instructions that are successive in code are not necessarily executed by the processor in succession. Rather, the intervention of other instructions resident elsewhere in code occurs when the CPU services interrupts,etc. An interrupt can occur any time, causing the currently active process to be delayed while another process services the interrupt. In all cases, when the active routine becomes suspended, its state is saved and restored later.

## 9.2 SCHEDULING POLICY

Each process is a separate entity with respect to execution on the host processor(s). 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 the Microprocessor Pascal run-time environment is in one of two states:

1) Ready To Execute: This "ready" state includes active processes and the process that is currently executing.

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

The relative priorities of processes determine which of several ready processes becomes active. Each process has a priority represented by a user-assigned, non-negative integer. The value 0 indicates the greatest urgency; 32766 indicates the least urgency. 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 ambiguity between the numeric value of the scheduling parameter "priority" and the relative priority of one process to another; confusion can arise since small numeric values of priority (e.g., interrupt handlers) correspond to great urgency.

A scheduling decision is made by the Executive RTS whenever a blocked process becomes ready or an active process becomes blocked. Currently one processor is managed, and there exists only one active process assigned to that processor. The fundamental characteristic of the Executive RTS scheduling policy is that the active (executing) process is never less urgent than any of the ready processes.

The following scheduling algorithm is used for the one processor:

1) All processes that are ready for execution reside in an ordered structure called a ready 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 ready 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 ready 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 ready queue becomes the active process. Since the ready 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 ready 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 remains active.

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. Figure 9-1 illustrates the application of the Executive RTS scheduling algorithm to a series of process activations and suspensions.

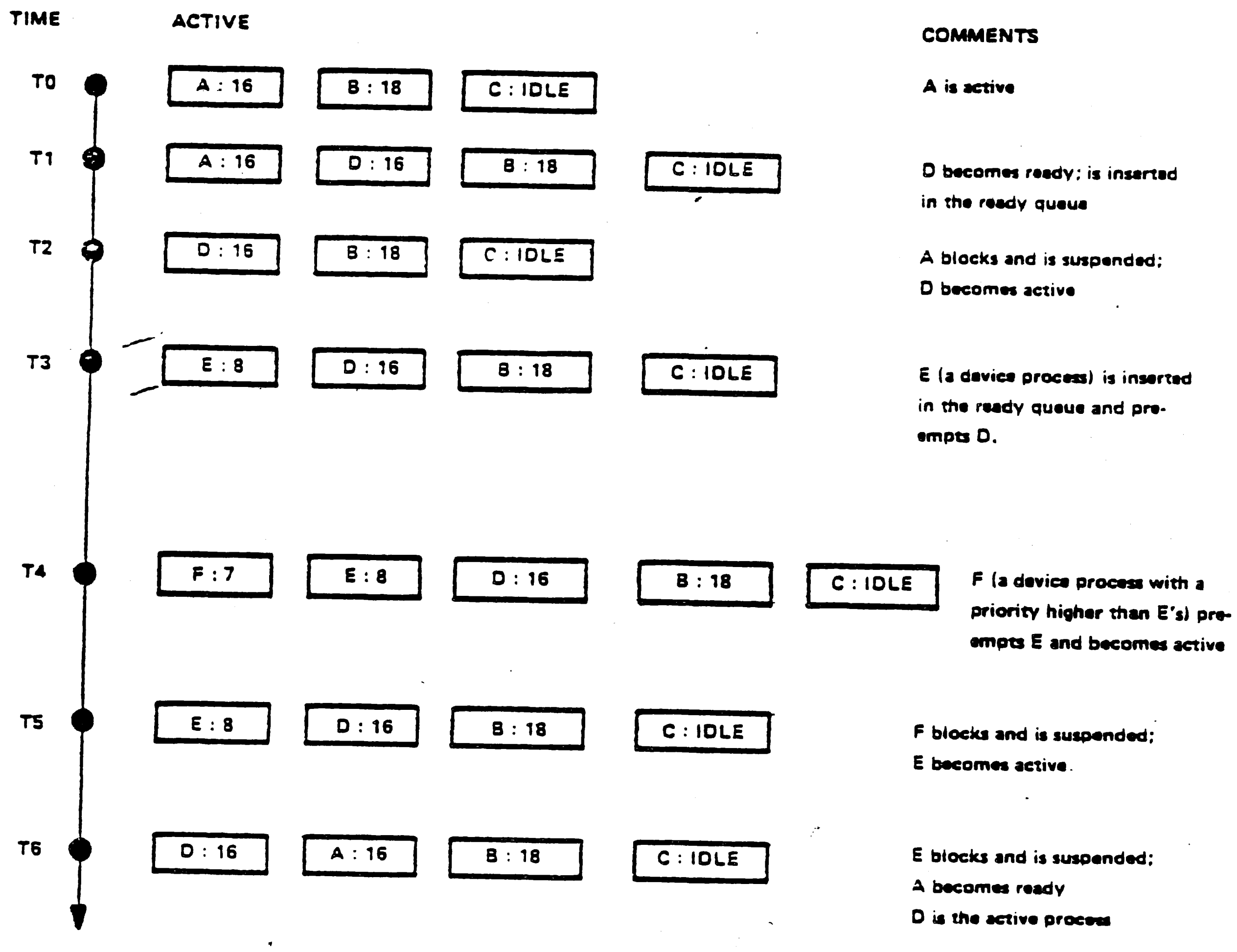

TIME
ACTIVE
COMMENTS
T0
A : 16
B : 18
C : IDLE
A is active
T1
A : 16
D : 16
B : 18
C : IDLE
D becomes ready; is inserted in the ready queue
T2
D : 16
B : 18
C : IDLE
A blocks and is suspended; D becomes active
T3
E : 8
D : 16
B : 18
C : IDLE
E (a device process) is inserted in the ready queue and pre-empts D.
T4
F : 7
E : 8
D : 16
B : 18
C : IDLE
F (a device process with a priority higher than E's) pre-empts E and becomes active
T5
E : 8
D : 16
B : 18
C : IDLE
F blocks and is suspended; E becomes active.
T6
D : 16
A : 16
B : 18
C : IDLE
E blocks and is suspended; A becomes ready D is the active process

FIGURE 9-1. EXAMPLES OF THE EXECUTIVE RTS SCHEDULING POLICY.

In Figure 9-1 the ready queue is represented as a horizontal series of boxes. Each process (box) is labeled with a letter and a priority number. The first box in the ready queue is the active process. Time moves vertically from top to bottom. Comments to the right of each queue describe the action performed.

The execution of the RTS scheduling policy displayed in Figure 9-1 results in process "B" never becoming active. In fact, B will never become active unless all other processes in queue with greater urgency become blocked or terminate execution. A process of higher urgency that becomes ready will always interrupt the currently active process. Once the more urgent process terminates (or becomes blocked) the previously active process will resume execution (unless another higher priority process has become ready). This "preemptive scheduling with resumption" is designed for event-driven systems in which the event is some real-world occurrence that demands the immediate attention of the computer. (See Paragraph 9.3).

It is possible for the processor to alternate among a collection of computation-bound processes by use of the RTS utility:

```
PROCEDURE SWAP; EXTERNAL;
```

This "SWAP" procedure removes the first non-device process from the queue and inserts it behind the last process with the same priority. Figure 9-2 illustrates this.

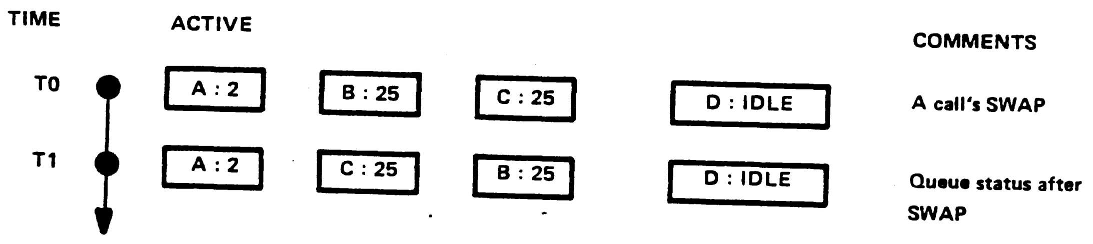


FIGURE 9-2. SWAP PROCEDURE.

A purpose of SWAP is to implement time slicing. SWAP is called to force an active non-device process to relinquish the processor. This swapping of the active process prevents it from running longer than its user-specified execution time (time slice).

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.

## 9.3 EVENTS

Synchronizing the action of a proces with the actions of other processes is achieved by means of an event, which causes the actions to seem to coincide or act together in time.

Real-time programming is concerned with the control of events (input) that do not wait for the computer. The central problem is that the computer must be able to receive data and react to it as fast as it arrives; otherwise the computer falls behind. In the Microprocessor Pascal System, the event mechanism is used to control the actions of concurrent processes within a computer. External events are synchronized via interrupt handlers.

### 9.3.1 External Events

For example, a process may be waiting for an event to occur, so that, when the event happens, the process can either service the event, proceed with the process, or perform some other action. Until the event happens, the waiting process is considered by the Executive RTS to be suspended and thus, it does not compete for the CPU.

### 9.3.2 Internal Events

The event upon which a process waits may be generated by another process. Processees that have related responsibilities often must communicatee among one another, either through shared memory or some other form of message-passing protocol. Successful communication between processes requires synchronization to ensure that they do not interfere with each other. Internal events are synchronized via semaphores. For example, if one process wants to place an item into the next available space in a buffer, the CPU must ensure that the receiving process does not modify pointers to the buffer until transfer is complete. Usually, a system designer cannot determine the rates at which individuals processes progress, so semaphores provide a low-level primitive structure for synchronizing otherwise asynchronous processes.

## 9.4 SEMAPHORES

The fundamental tool in the Executive RTS for the synchronization of processes is the semaphore. A semaphore event is an internally generated event. A semaphore represents some event on which processes synchronize. A process may ensure that an event has occurred by performing a WAIT operation on the associated semaphore before proceeding. If the event has already occurred, the process continues execution; if not 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.

Processes that await the same event are delayed in the same semaphore queue. The Executive RTS orders a semaphore queue by the sequence in which processes are delayed in the 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 has been awaiting the event for the longest period of time is activated.

The semaphores of the Executive RTS are counting (general) semaphores 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).

### 9.4.1 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 abstract algorithms for the operations WAIT and SIGNAL are described below.

```
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 removed from or inserted into the scheduling queue. When a signaled process is made ready, the Executive RTS scheduling algorithm determines whether the SIGNALer or SIGNALee will be the active process.

Since a semaphore acts as an event counter, it must be initialized to the proper count before it can be used. 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. 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 CKSEMAPHORE(SEMA: SEMAPHORE): BOOLEAN;
  EXTERNAL;
```

which returns TRUE for a valid SEMA.

As is discussed in detail in Paragraph 9.4.4, 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. Note: TERMSEMAPHORE must be called before a semaphore is reintialized for reuse.

### 9.4.2 Incorrect Use Of Semaphores

Care must be taken in the usage of semaphores. Figure 9-3 illustrates an incorrect usage. Two processes are considered. One is placing a value into a memory cell. The other process will use this value. The semaphore DATA_AVAILABLE is used to synchronize the 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 9-3. EXAMPLE OF INCORRECT SEMAPHORE USE.

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 into 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. However, 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. Figure 9-4 demonstrates correct implementation of the semaphores.

```
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(DATA_RECEIVED, 1);
    START PRODUCER;
    START CONSUMER;
  END { SYNCHRONIZE };

BEGIN
  START SYNCHRONIZE;
END { TEST }.
```

FIGURE 9-4. EXAMPLE OF CORRECT SEMAPHORE USE.

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.

Another example of incorrect usage of semaphore involves a condition called "deadlock". This takes place when a situation is created in which two or more processes are suspended awaiting a condition that cannot happen because there is no active process to cause the needed event to occur.

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

```
A gets X  ..  A requests Y

B gets Y  ..  B requests X
```

In the above example, neither A nor B will ever resume execution, as A will be waiting for Y (which B has) and B will be waiting for X (which A has). To prevent a situation such as this, one or both processes must check the availability of succeeding resources and, if unavailable, release those already acquired.

## 9.4.3 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 are as follows.

```
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. If a process performs a WAIT operation on a semaphore which has been attached to an interrupt level that is more urgent than the process calling WAIT, an exception occurs.

```
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. SEMA does not receive interrupt events.

```
PROCEDURE TERMSEMAPHORE( VAR SEMA: SEMAPHORE );
  EXTERNAL;
```

This procedure notifies the Executive RTS that SEMA is no longer in use. If SEMA 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; EXTERNAL;
```

This function returns the value in the count field of the semaphore, i.e., 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 no process is waiting for the SEMA event and no unserviced SEMA events have occurred. The result returned by this function must be used with care. It accurately reflects the state of SEMA at the time SEMAVALUE was called. 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. 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. 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. One application of SEMASTATE is to activate all processes waiting on a semaphore:

```
        WHILE SEMASTATE( SEMA ) = AWAITED DO
          SIGNAL( SEMA );
```

For the above 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 the above routine in one indivisible step: a WAIT operation on the WAITFOR semaphore and a SIGNAL on 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. It 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. If a process performs a WAIT operation on a WAITFOR semaphore which has been attached to an interrupt level that is more urgent than the priority of that process, an exception occurs.

```
PROCEDURE CSIGNAL( SEMA: SEMAPHORE;
  VAR WAITER: BOOLEAN ); EXTERNAL;
```

This procedure peforms a SIGNAL operation only if a waiter on the SEMA event exists; i.e.,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;
```

In this example, CSIGNAL is not equivalent to

```
IF SEMASTATE( SEMA ) = AWAITED THEN SIGNAL( SEMA )
```

since this statement is not indivisible: an interrupt can occur after SEMASTATE determines a waiting process exists, and the interrupt handler can activate that process before SIGNAL is 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). 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 the possibility of the calling process being suspended. CWAIT can be used in an interrupt-handling process to accept only a signal that has already been sent. In this case, 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.

### 9.4.4 Implementation of Semaphores

When a variable of type SEMAPHORE is declared in the Microprocessor Pascal System the 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.

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. 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.

## 9.5 INTERRUPT HANDLING

A 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 the 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. The means also exists to implement assembly language interrupt handlers. Several interrupt routines dedicated to this function are described in Paragraph 9.5.2.

### 9.5.1 Interrupts Treated as Events

An interrupt may be viewed as an event which is externally generated. 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 9-5 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 9-5. 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.

### 9.5.2 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;
  REGISTERS = (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,
               R12,R13,R14,R15);
  WP = ARRAY[R0..R15] OF INTEGER;
```

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 (0 through 15) 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 the 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. One interrupt level may be attached to no secondary semaphores, or to the 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 has been designated as the primary receiver of interrupt events for the LEVEL interrupt. If no semaphore has been so designated, this routine does nothing.

```
PROCEDURE NOALTEXTERNALEVENT( LEVEL: INTERRUPT_LEVEL );
  EXTERNAL;
```

This procedures detaches any semaphore which has been designated as the secondary receiver of interrupt events for the LEVEL interrupt. If no semaphore has been so designated, this routine does nothing.

```
PROCEDURE ASSEMBLYEVENT(VAR INTERRUPT_WP: WP;
  INTERRUPT_PC: INTEGER; LEVEL: INTERRUPT_LEVEL); EXTERNAL;
```

This procedure associates the INTERRUPT_WP workspace and the INTERRUPT_PC assembly language program counter with the LEVEL interrupt as the primary receiver of an event. Upon an interrupt occurring at the appropriate level, the machine will execute a BLWP using this workspace pointer and program counter before calling any Pascal handler. Depending upon the actions of the interrupt handler, the interrupt will either be handled entirely in assembly language or in Pascal as well. The workspace to be used can be a Pascal variable or COMMON, and may be on the stack or in heap. The program counter should be passed using the LOCATION function, with the assembly language routine declared EXTERNAL (and the symbol DEFed in assembly language). The same workspace may be used for several levels if desired. See the Microprocessor Pascal Executive User's Manual (MP385) for forms of the handler in native code.

This routine is not supported in the Host Debugger and cannot be called under the Host Debugger.

```
PROCEDURE NOASSEMBLYEVENT(LEVEL: INTERRUPT_LEVEL); EXTERNAL;
```

This procedure disassociates any assembly language interrupt handler which had been designated as the receiver of an interrupt event. After calling this routine, Pascal will handle the next interrupt (if one occurs) for the LEVEL interrupt. If no assembly language interrupt handler had been so designated, this routine does nothing.

This routine is not supported in the Host Debugger and cannot be called under the Host Debugger.

Figure 9-6 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 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 of the particular devices associated with each interrupt, SPURIOUS cannot clear any interrupt that it 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 = 1; STACKSIZE = 200; HEAPSIZE = 0 }
    INITSEMAPHORE( SPURIOUS_INTERRUPT, 0 );
    FOR I := 1 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 { PRIORITY = 1; STACKSIZE =200}
  START SPURIOUS;
  { START OTHER USER PROGRAMS }
END { TEST }.
```

FIGURE 9-6. SPURIOUS INTERRUPT PROGRAM.

### 9.5.3 General Features of Interrupt Handling

9.5.3.1 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.

9.5.3.2 Techniques of Code Style. Figure 9-7 illustrates several stylistic features of interrupt handling and interfacing. (This is an abstraction of the example in Figure 9-5.)

```
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 9-7. 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 a 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. 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.

## 9.6 SCHEDULING OF DEVICE AND NON-DEVICE PROCESSES

The Executive RTS scheduling policy that is presented in Pararaph 9.2 treats the scheduling of device and non-device processes having the same urgency in a different manner. 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 a 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 10

# PROCESS COMMUNICATION

## 10.1 OVERVIEW

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 that communication can be as simple 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 the more sophisticated 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.

## 10.2 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 only be used to implement more flexible communication systems. The mechanisms to be discussed here include:

- o Device communication using CRU
- o Device communication using memory-mapped I/O
- o Interprocess communication using shared variables
- o Interprocess communication using message buffers

### 10.2.1 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)

    PRODEDURE 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.

### 10.2.1.1 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.

### 10.2.1.2 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.

### 10.2.1.3 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.

### 10.2.1.4 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.

### 10.2.1.5 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.

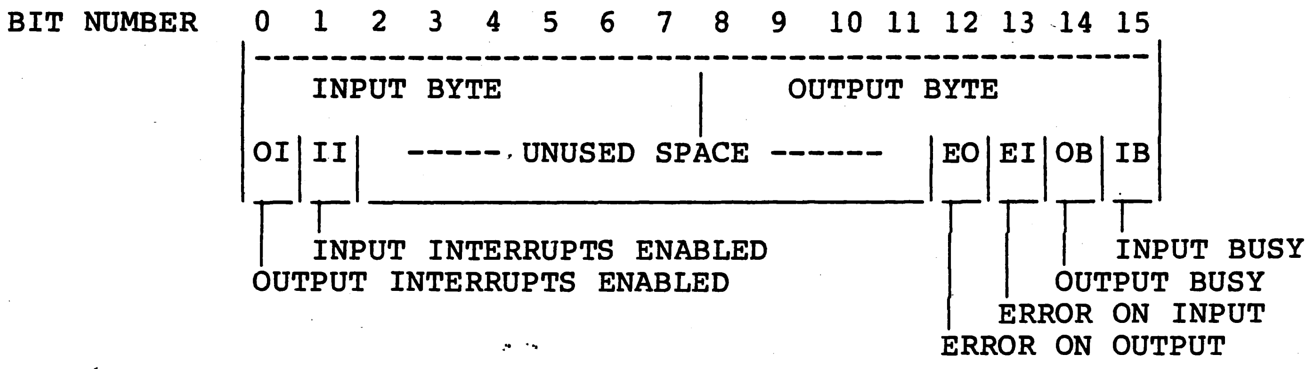
### 10.2.1.6 FUNCTION TB (Test Bit).

```
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.

## 10.2.2 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. Figure 10-1 is an example device control record represented first as a diagram, 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.



```
TYPE
  BYTE     = 0..#00FF;
  TEN_BIT = 0..#03FF;

  DEVICE_INSTANCE =  CONTROL RECORD;
  CONTROL_RECORD = PACKED RECORD
                     INPUT_BYTE                    : BYTE;
                     OUTPUT_BYTE                   :BYTE;
                     OUTPUT_INTERRUPTS_ENABLED     : BOOLEAN;
                     INPUT_INTERRUPTS_ENABLED      : BOOLEAN;
                     UNUSED_SPACE                  : TEN_BIT;
                     ERROR_ON_OUTPUT               : BOOLEAN;
                     ERROR_ON_INPUT                : BOOLEAN;
                     OUTPUT_BUSY                   : BOOLEAN;
                   END:
```

FIGURE 10-1. INTERFACE TO MEMORY MAPPED I/O DEVICE.

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. Figure 10-2 is an example of control space manipulation.

```
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.

NOTE: 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.

### 10.2.3 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.

### 10.2.4 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 IMPLEMENTATION OF MESSAGE BUFFERING DATA.

The declaration of the message buffer is in the form of a record. The 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.

```
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-COMING 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.

NOTE: 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 (Paragraph 10.2) provide a much higher level interface and should be used when possible.

## 10.3 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. 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.

NOTE: The Microprocessor Pascal System (on the Target system) supports logical I/O to only those logical file types for which the relevent como component subsystem is available.

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.

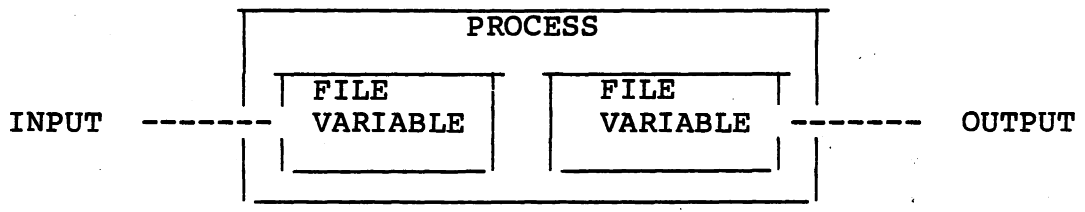


FIGURE 10-6. FILE VARIABLES AS PROCESS-LOCAL PORTS.

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-compatible" test component that injects test data into the system or monitors the data generated by other parts of the system. ("Plug-compatible" means that test component has the same interface structure as the "real" component.)

## 10.3.1 Process-Local File Variables

A Microprocessor Pascal file variable is actually a process-local port which interfaces the process with its external environment as illustrated in Figure 10-6. In referencing this figure, please note: the file should be opened for access (RESET for input and REWRITE for output) only after it has been passed to the process (i.e., the file must be opened only in the process in which it is used).

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". (NOTE;Lower case letters are not significant; MPP converts all characters to uppercase.) 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(in, out: 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");
```

Lower case characters in S are insignifiacnt and trailing blanks are stripped. For example:

```
    SETNAME(F, 'MAGTAPE      ')
```

changes the name of F to "MAGTAPE".

## 10.3.2 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.

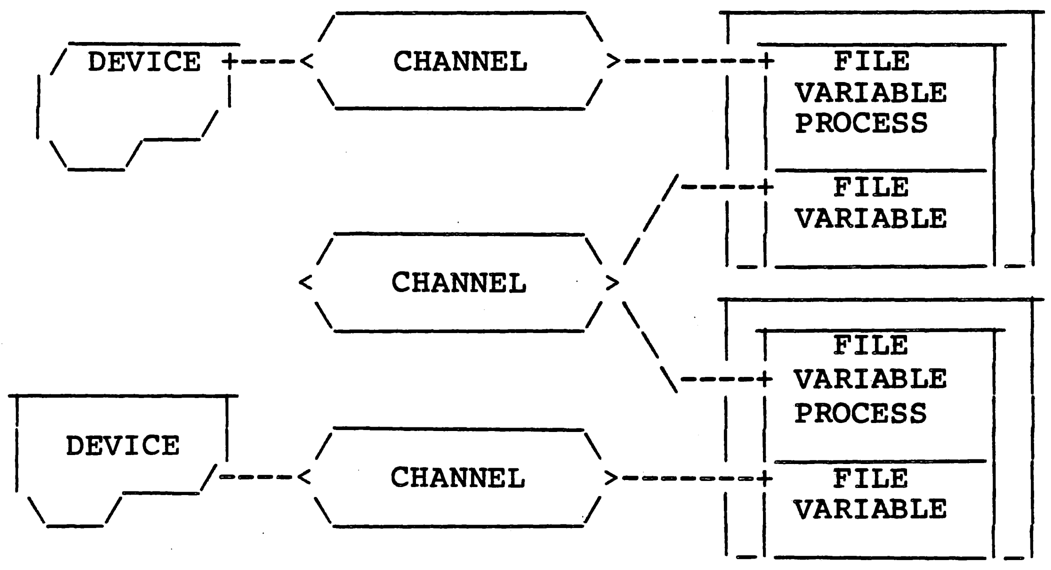


FIGURE 10-7. CHANNEL CONNECTIONS.

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.

## 10.3.3 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).

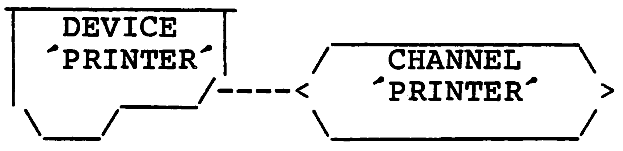


FIGURE 10-8. LOGICAL DEVICE AND ASSOCIATED DEVICE CHANNEL

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.

## 10.3.4 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  : TEXT;
                     OUTPUT : TEXT );
  VAR
    CH          : CHAR;
    PAGE_NUMBER : INTEGER;
    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  : TEXT;
                                    OUTPUT : TEXT);
      VAR
        R     : REAL;
        THETA : REAL;

      FUNCTION  COS ( X : REAL ) : REAL; EXTERNAL;

      FUNCTION  SIN ( X : REAL ) : REAL; EXTERNAL;

      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.

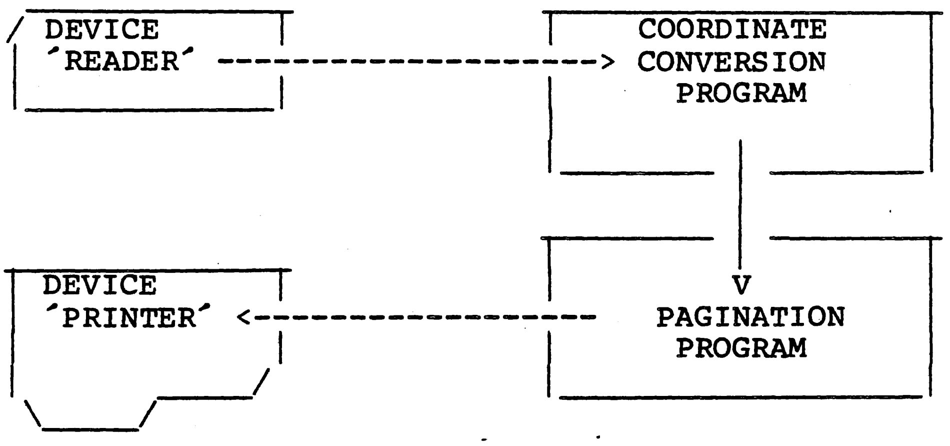


FIGURE 10-11. COMMUNICATION AMONG PROGRAMS AND DEVICES.

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.

## 10.3.5 Sequential (Non-Text) File Operations

Sequential files are used to transmit or receive values to or from their associated channels in "binary" (memory image) 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.

## 10.3.6 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.

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.

```
VAR
  F: TEXT;
  R: REAL;
```

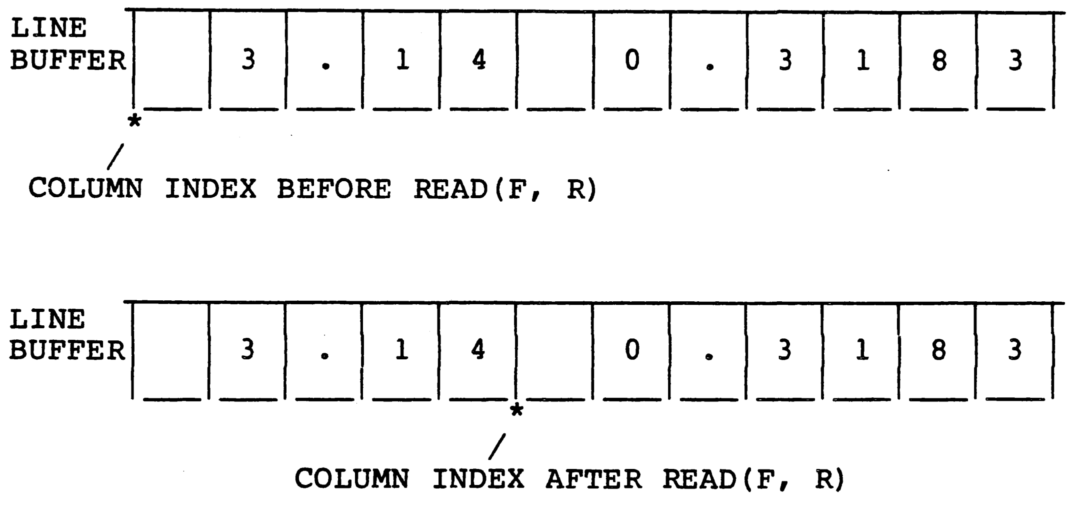


FIGURE 10-12. COLUMN INDEX IS INCREMENTED DURING TEXT READ.

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.

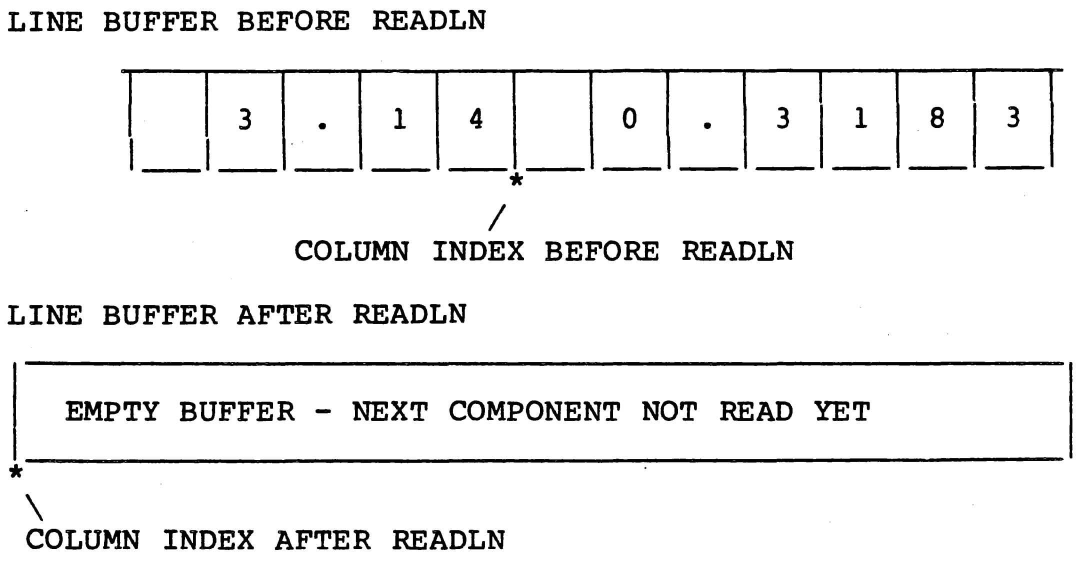

LINE BUFFER BEFORE READLN
3 . 1 4 0 . 3 1 8 3
*
COLUMN INDEX BEFORE READLN
LINE BUFFER AFTER READLN
EMPTY BUFFER - NEXT COMPONENT NOT READ YET
*
COLUMN INDEX AFTER READLN

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.)

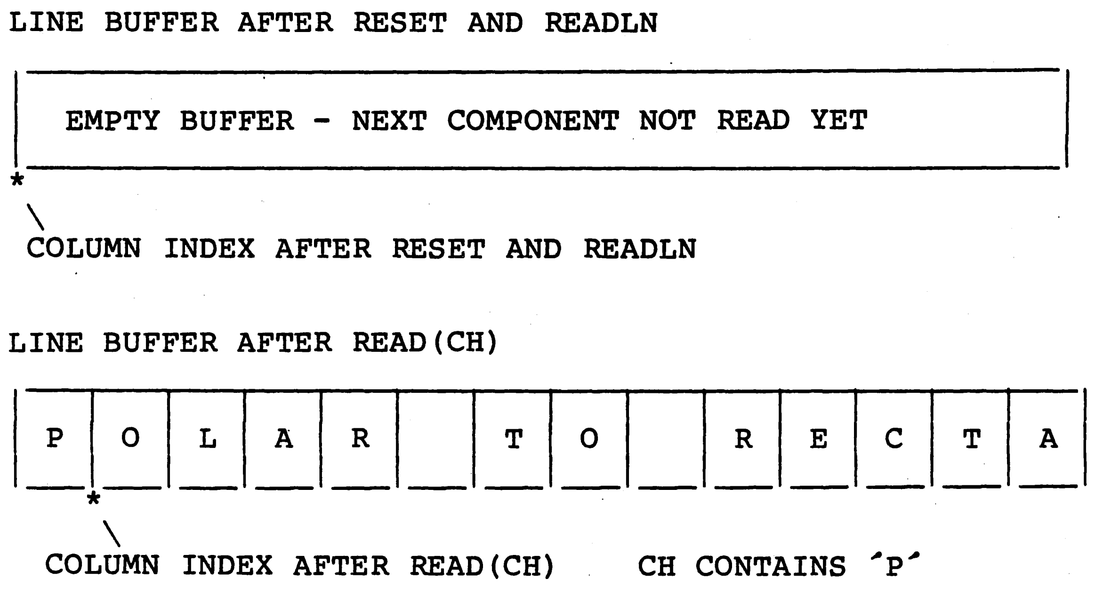

LINE BUFFER AFTER RESET AND READLN
EMPTY BUFFER - NEXT COMPONENT NOT READ YET
COLUMN INDEX AFTER RESET AND READLN
LINE BUFFER AFTER READ(CH)
P O L A R T O R E C T A
COLUMN INDEX AFTER READ(CH)
CH CONTAINS ´P´

FIGURE 10-14. EFFECT OF READING FIRST CHARACTER ON LINE.

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.

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 Section 10.3.8.) 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.

```
LINE BUFFER AFTER RESET AND READLN

|------------------------------------------------------|
|   EMPTY BUFFER - NEXT COMPONENT NOT READ 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.

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.

## 10.3.7 Random File Operations

Random files are aupported in Host "execute" mode, but are not yet implemented in the Host ´debug´ mode. If an attempt is made to "debug" a system contianing random file, the message:

RANDOM FILES NOT IMPLEMENTED

will be displayed.

Random files are supported in the Target "native" mode, but not target interpretive mode (i.e., random files are supported in the Host execute and Target Mpx modes, but not in Host debug or Target MPIX modes.

The operation of random files is very similar to the operation of sequential files, except that a "record number" is included in the parameter list for the standard procedures READ and WRITE, i.e,:

```
READ(f,<record number>,<variable>);
WRITE(f,<record number>,<variable>);
```

The first record in a file is record 0. If record n is written, then every record 0 through n exists, although they may not contain "correct" data.

### 10.3.8 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.

### 10.3.9 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 value 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;
```

### 10.3.10 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.

## 10.3.11 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;
```

### 10.3.12 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$CONDITIONAL can be used to alter the conditional attribute of a sequential file. (Text and random files may not have the conditional attribute.) The calling sequence of F$CONDITIONAL 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 transfer 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.

Figure 10-17 illustrates the use of EOF to detect end of transmission on the channel associated with a conditional file. 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.

```
PROGRAM SERVER(f, FILE OF INTEGER);
  VARIi: 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.

## 10.3.13 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 <u>not</u> 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 11

# PROCESS MANAGEMENT

## 11.1 OVERVIEW

The 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.

Remember, 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 clearly made.

## 11.2 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 statements in a program or process. This 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 that device. Figure 11-1 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 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 execute concurrently.

The priority-concurrent characteristic illustrated in Figure 11-1 sets the priority of the system process to one. The interrupt mask of the system process is zero (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 (than one) executes. When the environment of the system process is created by the Executive RTS before the system process begins, the environment initialization executes an RSET instruction. RSET is a hardware instruction that resets directly connected peripheral devices and those CRU devices that provide for reset in the interface with the CRU. Since all devices may not be reset by the environment initialization, and because servicing interrupts from uninitialized devices can 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 Paragraph 14.2 (Physical Device Interface Systems).

Declarations in the SYSTEM mainly include programs but may also include routines (procedures and functions) which are globally available to the entire system (e.g., provided by the run-time support). Constants declared at the system level can 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 can 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. 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.

A VAR section is not allowed 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.

## 11.3 PROGRAM DECLARATION

The Microprocessor Pascal System allows more than one program to be declared in the SYSTEM construct. Pascal 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, to a great extent, 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). Program parameters 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.

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. Paragraph 10.3 presents information on interprocess files; Section 14 explains the construction of device handlers.

## 11.4 DECLARATION OF A CONVENTIONAL PASCAL PROGRAM

A conventional Pascal program in a non-concurrent environment is shown below:

```
PROGRAM USER;
{ Declarations }
BEGIN
  { Executable Statements }
END.
```

FIGURE 11-3. SIMPLE, CONVENTIONAL PASCAL PROGRAM.

Figure 11-4 presents the Microprocessor Pascal System equivalent to this program.

```
SYSTEM DUMMY;

  PROGRAM USER;
  { Declarations }
  BEGIN
    { Executable Statements }
  END;

BEGIN
  START USER;
END.
```

FIGURE 11-4. EQUIVALENT MICROPROCESSOR PASCAL SYSTEM.

The conventional 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. In the Microprocessor Pascal System, the user's conventional program, 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 (See Section 8). Figure 11-5 shows a program which uses OUTPUT and is compiled as a conventional program in a non-current environment.

```
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.

Figure 11-6 displays its Microprocessor Pascal System equivalent.

```
SYSTEM EXAMPLE;
  PROCEDURE INIT_SYSTEM;
  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 MICROPROCESSOR PASCAL SYSTEM.

If the program in Figure 11-6 is the only user-written code to execute on a processor, no peripheral devices (physical devices) will be initialized or enabled, and no other processes will be 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) must 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, these files must be declared as program parameters. If the program in Figure 11-5 above is compiled within a SYSTEM, then it must declare parameter OUTPUT since it references that file. Also, 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 (see Paragraph 5.2.3) 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 causing 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.

## 11.5 PROCESS DECLARATION

A process is declared within a program or another process starting at lexical level two. Because of the scope rules of Pascal, variables global to a process are addressable by the process. This is the case if these variables are local to a program (or some other process) which is a lexical ancestor of the process in question. Figure 11-7 illustrates the nesting of processes and their variables in scope.

```
SYSTEM EXAMPLE:

  { No VAR Section Allowed At SYSTEM Level }

  PROGRAM PROG;
    VAR GLOBAL: CHAR;<-----------------
                                       |
    PROCESS PROC1;                     |
      VAR A: CHAR;                     |
        B: CHAR;<------------------    |
                                   |   |
      PROCESS PROC2;               |   |
        VAR A: CHAR;<--------      |   |
      BEGIN                  |     |   |
        { The Following      |     |   |
          Variables Are      |     |   |
          Addressable By     |     |   |
          process PROC2:     |     |   |
                             A     |   |
                                   B   |
                                       GLOBAL
        }
      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 has started executing, the variables that are local to the procedure or function are addressable to the process according to scope rules. However, these variables 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 local variables to remain until all lexical descendants of the terminated process finish. Thus the Executive RTS ensures that the extent of local variables of a process is at least as long as the extent of lexically descendent processes which can address these variables (by scope rules). (Since a program has a self-contained data space, no other process must wait on a program to terminate.) As 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 process declarations) or by means of variables in heap (achieved by sharing pointers which reference the heap data). Process synchronization (as explained in Paragraph 9.4) may be required to coordinate the sharing of data among concurrent processes.

## 11.6 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 32767 (where zero indicates the greatest urgency, and the range zero to fifteen is reserved for device processes-- see Section 9). 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. 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 32767 never executes until all other processes have terminated or are blocked. Priority 32767 is reserved for the IDLE process (started by the Executive), which executes when all user processes are suspended or terminated.

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 of the process 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 user PRIORITY >=32767 is considered illegal and causes an exception for the process attempting a START of process with such a 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.

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, no nested heap is created, and the parent's heap is used as the child's heap.

A process references 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 the stacks of nested processes which are started. The default for an unstated HEAPSIZE is zero. This is interpreted to mean use of 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 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 discusses in detail how to select the proper STACKSIZE and HEAPSIZE for a process.

## 11.7 PROCESS INVOCATION

A program or process is initially invoked with the START statement. In the START statement, the program or process identifier is names (the identifier must be in scope) and process parameters are passed (if they exist) much like a procedure call. However, while in a procedure call, the calling routine resumes after the called procedure; 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 copies 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 of this function 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 managing 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 indicating 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.

## 11.8 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. This technique is useful for a routine, which is shared by more than one process; It is more general than an ESCAPE statement since an ESCAPE must specify a lexical parent and this P$ABORT be used in a routine that may be called from an arbitrary process. Please note that P$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. Only then are the local variables no longer addressable and destroyed.

## SECTION 12

## MEMORY MANAGEMENT BY THE RTS

### 12.1 OVERVIEW

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.

### 12.2 MANAGEMENT OF SYSTEM MEMORY

The paragraphs that follow address the topic of RTS memory management as it applies to dynamically allocatable system memory and statically allocatable system memory.

#### 12.2.1 Dynamically Allocated Data Areas

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 described in Section 15 covering the specification of RAM locations. Disjoint contiguous areas of RAM are allowed in the target processor.

12.2.1.1 <u>Heaps:</u> Data space in programmable memory (RAM) that is available to a process is managed by a heap structure. 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 and allocated areas may be freed. In addition, 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 with a non-zero HEAPSIZE characteristic begins, nested heaps are created within the heap. These offspring are returned to the parent when the process which causes the nested heap to be created terminates. Packets allocated out of a nested heap remain allocated even if the process which caused the nested heap to be created terminates. The packets which were in the nested heap are considered to be in the parent's heap and remain addressable.

Since all processes lexically nested within a program either use the program heap or a heap nested within the program heap, a program heap is a logical division of data space for memory mapping. The Microprocessor Pascal System language ensures that no references or pointers may be passed outside of a program. Thus, 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 way to pass data among processes that reside in separate programs and data spaces is through the interprocess file mechanism provided by the Executive RTS.

### 12.2.2 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.

## 12.3 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.

### 12.3.1 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.

### 12.3.2 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.

## 12.4 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;
```

### 12.4.1 Procedure NEW$

```
     PROCEDURE NEW$(VAR PTR: POINTER; LENGTH: BYTE_LENGTH);
       EXTERNAL;
```

This procedure allocates a contiguous area from the current process's heap in bytes of LENGTH or more, and returns a pointer to the area in PTR.

### 12.4.2 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; the packet is returned to the heap from which it was allocated.

### 12.4.3 Procedure HEAP$TERM

```
     PROCEDURE HEAP$TERM(VAR OLDVALUE: BOOLEAN; NEWVALUE:
          BOOLEAN); EXTERNAL;
```

This routine allows user control over heap overflow. Heap overflow 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 new 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.

## 12.5 USE OF COMMONS

The Microprocessor Pascal System language prohibits passing of references or pointers outside of a program. The Executive RTS keeps all data of a program and its lexically nested processes 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. However, 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 disturbed.

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 second 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 example routine suspends the calling process forever.

```
PROCEDURE SUSPEND;
  VAR
    FOREVER: SEMAPHORE;
  BEGIN
    INITSEMAPHORE(FOREVER, 0);
    WAIT(FOREVER);
  END;
```

## 12.6 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 also applies to a program or the system process.

### 12.6.1 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 instances of routines which the process calls. The heap packet holding the process stack is disposed when the process terminates.

Each instance of a routine called within a process requires space in the process stack (called a stack frame) for the parameters passed to it along with its local variables plus a 14 byte administration area. The MAP option of the compiler produces a listing in which the STACKSIZE value is given for each routine. The stack requirements of a process can be determined 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. File operations also require about 250 words of stack.

When a system process, a program, or a 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. However, this may not be enough space for the process to execute the algorithms that the user has coded; a stack overflow, which is a fatal error for the process may result. A larger value of STACKSIZE is the sum of the words allocated for the process´s parameters and global variables, the 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 causing an exception for the process attempting to START a new process with a negative STACKSIZE.

## 12.6.2 Process Heap

The system process and any program are always given a new program heap. This program heap 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 greater than zero is given, then the new program heap is large enough for the user to allocate a call to NEW or NEW$. (Smaller packets may also be allocated up to HEAPSIZE words, but the "checkerboarding" 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. 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 they allocate 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.

## 12.6.3 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 256 words. Notice that the maximum stack usage of 171 words and the current stack usage of 119 words are shown. This process could have been started with a STACKSIZE of 171 words without encountering a stack overflow exception.

```
 <>DP(TEST)
   Static/Dynamic Calling Order for Process TEST(2)

        Stack Size (words) = 256
        Stack Used (words) Maximum = 171  Current = 119

        Call Order          Name                Statement
          1                 TEST                    3
          2                 PASS1                   7
          3                 GETLINE                 7
          4                 GETCHAR                 1
```

FIGURE 12-1. DETERMINING STACK REQUIREMENTS OF A PROCESS.

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 shows an example of this.

```
 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       (................)
C098 (0020) 0000 0000 0000 0000 C062 C058                 (.........b.X     )
```

FIGURE 12-2. DETERMINING HEAP REQUIREMENTS OF A PROCESS.

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 shows an example of the SP command (Show 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 TARGET DEBUGGER TO DETERMINE STACK AND HEAP REQUIREMENTS OF A 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.

## 12.6.4 Allocation of Process Resources

A precise statement of the allocation rules of process resources is given below for several cases.

12.6.4.1 Allocation Of System Process Or New Program. A program heap is created from system memory (all 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 nested processes to be allocated plus 102 bytes of additional managerial area.

12.6.4.2 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´ heap. The stack of the new process is allocated as one heap packet from the parent´s heap.

12.6.4.3 Allocation Of Conventional Pascal Program. A 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.

## 12.7 EXAMPLE

The following program demonstrates 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 = 167; HEAPSIZE = 0}
         PRODUCE_DATA;
      END;

   BEGIN {Copier}
      {PRIORITY = 16; STACKSIZE = 317; HEAPSIZE = 715}
      START READER;
      START WRITER;
   END;

BEGIN  {Example}
   {PRIORITY = 1; STACKSIZE = 250}
   START COPIER;
END.
```

FIGURE 12-4. PROGRAM WITH CONCURRENT CHARACTERISTICS.

Process READER needs 80 bytes of stack space 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 routine called by TRANSFER. For example, TRANSFER itself requires about 6 bytes (for variables I, J, and C) plus the 14 byte administration area in the stack for the execution 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 (40 words) 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 because the Executive RTS makes sure 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 the 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 comprised of 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 about 200 bytes for the calls of PRODUCE_DATA and the routines it calls. So the STACKSIZE of WRITER is 134+200 bytes or 167 words. WRITER needs no heap space; 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 are comprised of (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 200 bytes (167 words STACKSIZE minus 134 bytes for IMAGE) plus 150 bytes (administration area) or 175 words.

Program COPIER uses 134 bytes of stack (the size of variable LINE) plus the requirements to START READER and WRITER; thus 250 words for the STARTs and 67 words for its global variable LINE. Therefore STACKSIZE of COPIER is 317 words. COPIER needs no heap for the algorithm of the program COPIER, but it does need heap space to allocate to its nested processes (READER and WRITER). An instance of READER needs a packet of 83 words for its nested heap and a packet of 375 words for its stack. An instance of WRITER needs a packet of 82 words for its global frame (No nested heap is created), and a packet of 175 words for its stack. Therefore, the HEAPSIZE of COPIER is 83+375+82+175 words or about 715 words. If additional instances 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 size consists of 26 bytes of administration area for the program heap plus the global frame of COPIER. The global frame of COPIER is equal to 30 bytes of administration area and 134 bytes for LINE plus 715 words of HEAPSIZE. HEAPSIZE then is 26+30+134+1430 bytes or 810 words. The stack of COPIER is allocated in one packet from system memory of size equal to 317 words of STACKSIZE minus 134 bytes for LINE plus 150 bytes of administration area. This calulates to 634-134+150 bytes or 375 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 the example in Paragraph 12-7.

| | | |
|---|---|---|
| PROCEDURE TRANSFER: | | |
| Stack Frame: | Administration | 14 bytes |
| | I, J, C | 6 bytes |
| | Routines Called | 580 bytes |
| | | 600 bytes = 300 words |
| PROCESS READER: | | |
| Stacksize: | CARD | 80 bytes |
| | TRANSFER | 600 bytes |
| | | 680 bytes = 340 words |
| Heapsize: | Nested | 30 bytes = 15 words |
| Size of Heap: | Heap Admin | 26 bytes |
| | Global Frame | 30 bytes |
| | Global Var´s | 80 bytes |
| | Avail Space | 30 bytes |
| | | 166 bytes = 83 words |
| PROCESS WRITER: | | |
| Stacksize: | IMAGE | 134 bytes |
| | PRODUCE_DATA | 200 bytes |
| | | 334 bytes = 167 words |
| Heapsize: | (Parent´s Heap) | 0 |
| PROGRAM COPIER: | | |
| Stacksize: | Start processes | 500 bytes |
| | LINE | 134 bytes |
| | | 634 bytes = 317 words |
| Heapsize: | Reader Stack | 600 bytes |
| | Reader Stack Admin | 150 bytes |
| | Reader Nested Heap | 166 bytes |
| | Writer Stack | 200 bytes |
| | Writer Stack Admin | 150 bytes |
| | Writer Global Frame | 30 bytes |
| | Writer Global Var´s | 134 bytes |
| | | 1430 bytes = 715 words |

FIGURE 12-5. MEMORY LAYOUT OF STACKS AND HEAPS FOR PARAGRAPH 12.7.

# SECTION 13

# ERROR RECOVERY AND EXCEPTION HANDLING

## 13.1 OVERVIEW

As a process executes, it may encounter an exception such as division by zero or subscript out of range. The ability of a process to deal with and possibly recover from exceptions is called exception handling. The mechanism by which a process in the interpretive execution mode can 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 |

NOTE: 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 PRIOIRTY
   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 contains a character which is 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 NO SEQUENTIAL1
    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 AFTRER $CREATECHANNEL
F$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$CREATECHANNEL CALLED TWICW
F$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.

207) IMPLICIT HOST FILE CONNECTION NOT ALLOWED
Host connection must be made explicitly using the CIF and COF debugger commands.

### 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 PRODEDURE 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

Concurrent 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 shared data and to keep it consistent. As an example, a heap resource is managed with a semaphore in the heap administration record. If a process was 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 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 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 exception handling procedure stays active throughout reprocessing after an exception. However, if another exception occurs, the execution of the exception handler is lost (by a simulated ESCAPE) and restart 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 to repeatedly 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 Paragrahs 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 only difference is in the reporting of 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 to have no 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> was 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 execution 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. 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 14

# IMPLEMENTATION OF DEVICE HANDLERS

## 14.1 OVERVIEW

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 (see Figure 14-2).

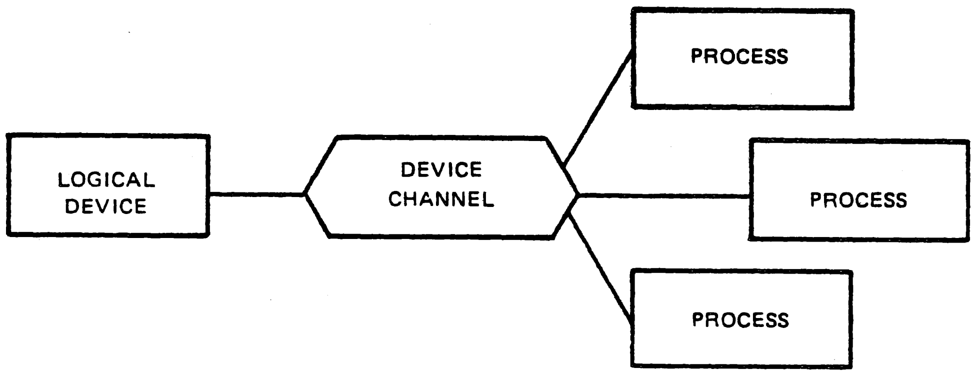


FIGURE 14-1. CONCEPTUAL VIEW OF INTERFACE TO A LOGICAL DEVICE.

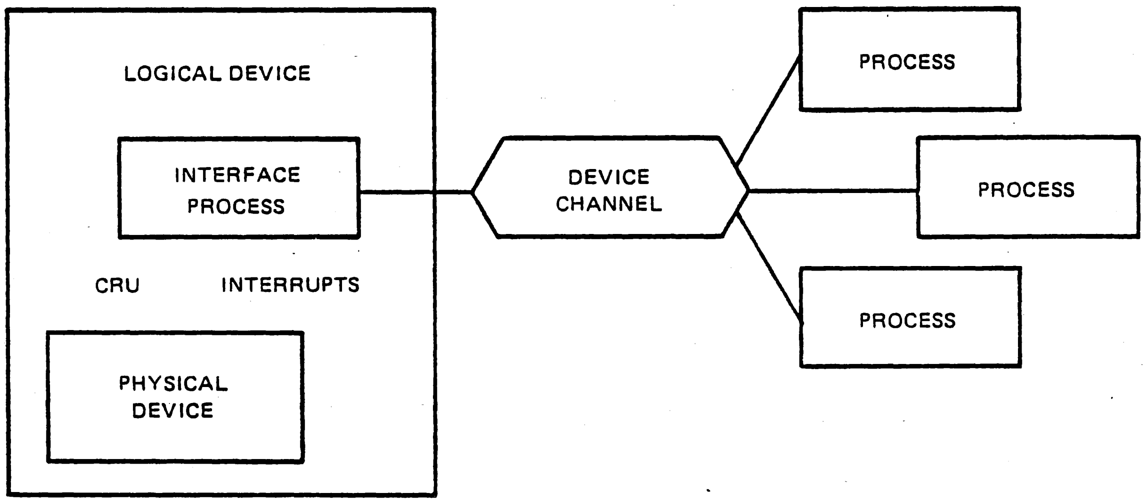


FIGURE 14-2. INTERFACE TO PHYSICAL DEVICE.

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.

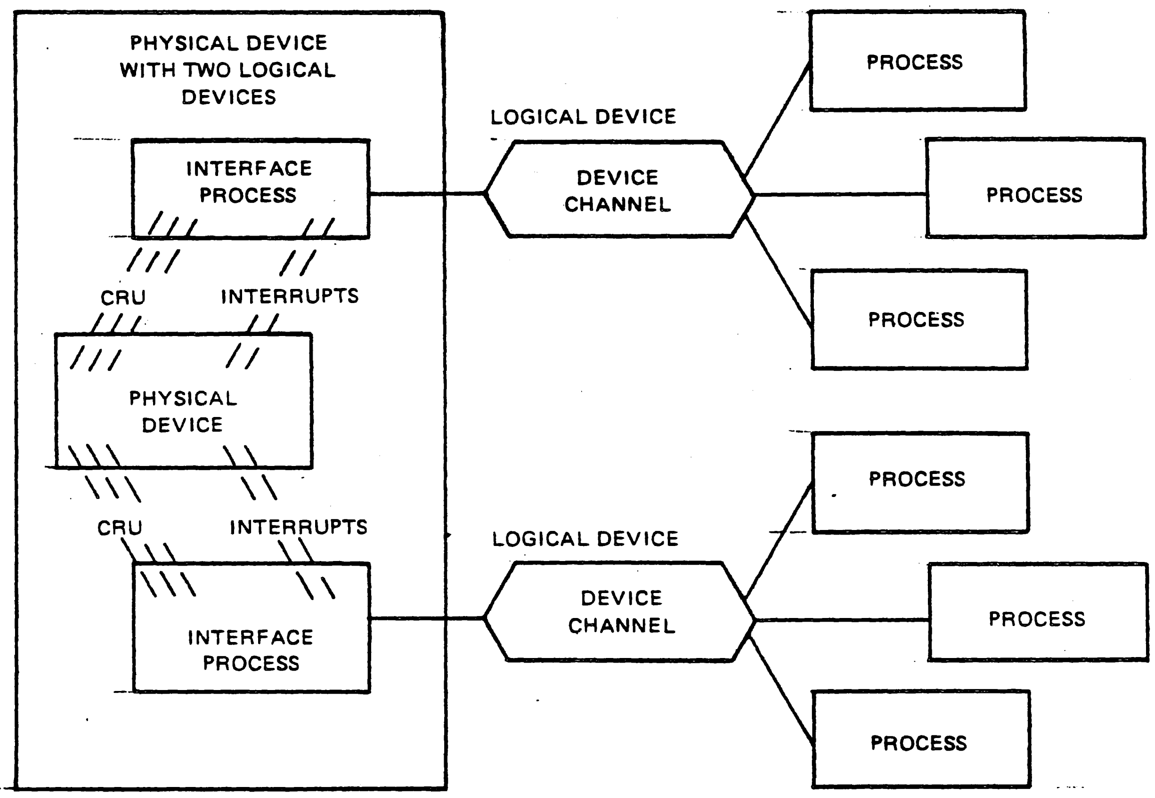


FIGURE 14-4. ILLUSTRATION OF MULTIPLE LOGICAL DEVICES ON A SINGLE PHYSICAL DEVICE.

## 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:

- o an initialization procedure
- o a supervisor program
- o one or more logical device interface processes
- o one or more logical device channels
- o an optional interrupt demultiplexer process

## 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.

```
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.

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 ASRSUPERVISOR(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 completely initializing 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 only the resources that are required for initialization 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) handling device errors, and (7) restarting 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.

- o name - defaults to name of file
- o component length - defaults to file component length
- o mode (reading or writing)
- o maximum number of users - defaults to 32767
- o 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 can be 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 open 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 opened (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.

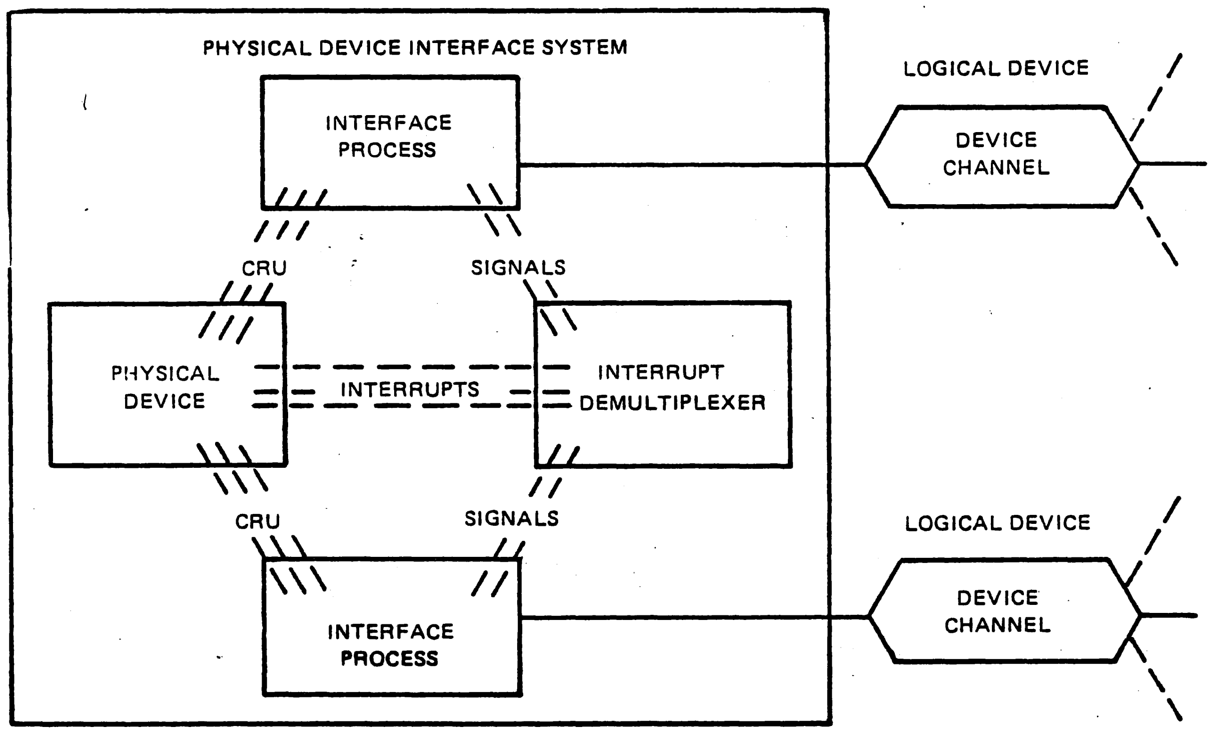


FIGURE 14-8. PHYSICAL DEVICE INTERFACE SYSTEM WITH INTERRUPT DEMULTIPLEXER PROCESS

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(INFIILE,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 LPPUT. 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.

```
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.

This procedure is lexically nested within LP$SUPERVISOR 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.

## 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).

```
PROCESS CASSETTE_DRIVE(F: TEXT;
  INITIALIZATION_COMPLETE: SEMAPHORE);

  VAR
    USERS_MODE: FILE_MODE;
    CH: CHAR;

  BEGIN {CASSETTE_DRIVE}
  F$MASTER(F);
  F$XACESS (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".

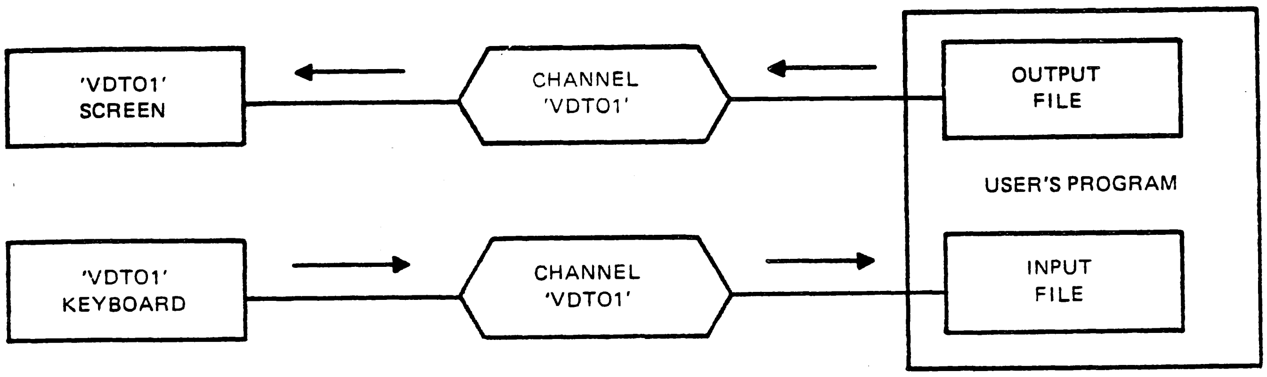


FIGURE 14-14. EXAMPLE OF CONNECTION OF USER FILES TO A VDT.

The VDT physical device interface system is initialized by calling the procedure VDTINIT which has the following calling sequence:

```
    TYPE
      CRU_ADDRESS = 0..#1FFE;
      INTERRUPT_LEVEL = 0..15;
      ALFA = PACKED ARRAY [1..8] OF CHAR;

    PROCEDURE VDTINIT(
      BASE: CRU_ADDRESS;
      LEVEL: INTERRUPT_LEVEL;
      NAME: ALFA;
      FILE_ORIENTED: BOOLEAN );
      EXTERNAL;
```

14.3.3.2 <u>Implementation of Initialization Procedure</u>. Figure 14-15 illustrates the implementation of the procedure VDTINIT which is invoked to initialize the entire system.

```
  PROCEDURE VDTINIT)
    BASE: CRU_ADDRESS;
    LEVEL: INTERRUPT_LEVEL;
    NAME: ALFA;
    FILE_ORIENTED: BOOLEAN );
  VAR
    INITIALIZATION_COMPLETE: SEMAPHORE;
  BEGIN
  INITSEMAPHORE( INITIALIZATION_COMPLETE, 0 );
  START VDTSUPERVISOR( 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.

```
PROGRAM VDTSUPERVISOR(
  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;

  PROCSS VDTKEYBOARD (OUTFILE: TEXT; LEVEL: INTERRUPT_LEVEL);
  ...
  END;

  PROCESS VDTSCREEN(INFILE: TEXT; LEVEL: INTERRUPT_LEVEL);
  ...
  END;

BEGIN {VDT$SUPERVISOR}
{PRIORITY = LEVEL;
 STACKSIZE = VDTSUPERVISOR_STACK;
 HEAPSIZE = VDTSUPERVISOR_HEAP}
CRUBASE(BASE) {SET CRU BASE TO INITIALIZE DEVICE};
SBZ(SELECT_WORD_W0_W1) {SELECT WORD ZERO};
SBO(ENABLE_DISPLAY_W0) {ENABLE DISPLAY};
OUTPUT_LINE := 24 {LAST LINE AVAILABLE FOR OUTPUT};
INITSEMAPHORE(PARTIAL_COMPLETION, 0);
INITSEMAPHORE(EXCLUSIVE_ACCESS_TO_PHYSICAL-DEVICE, 1);
START VDTKEYBOARD(FILENAMED(NAME), LEVEL);
  WAIT(PARTIAL_COMPLETION);
START VDTINIT$SCREEN(FILENAMED(NAME   LEVEL);
  WAIT(PARTIAL_COMPLETION);
SIGNAL(INITIALIZATION_COMPLETE);
TERMSEMAPHORE(PARTIAL_COMPLETION);
END   {VDTINIT$SUPERVISOR};
```

FIGURE 14-16. VDT INTERFACE SYSTEM 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 VDTSUPERVISOR_STACK and VDTSUPERVISOR_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 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 (VDTKEYBOARD and VDTSCREEN) 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 VDTINIT$SCREEN(INFILE: TEXT; LEVEL: INTERRUPT_LEVEL);

  PROCEDURE VDTINIT$SWORK;
  ...
  END;

  PROCEDURE VDTINIT$SEXCEPTION;
  ...
  END;

BEGIN {VDTINIT$SCREEN}
{PRIORITY = LEVEL;
 STACKSIZE = VDTINIT$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(VDTINIT$SEXCEPTION) ) {ESTABLISH EXCEPTION
  HANDLER};
VDTINIT$SWORK {INVOKE SCREEN WORK PROCEDURE};
END   {VDTINIT$SCREEN};
```

FIGURE 14-17. VDT SCREEN LOGICAL DEVICE PROCESS.

The nested procedure VDTSWORK does the majority of the work after the necessary structures are initialized. The nested procedure VDTSEXCEPTION 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 indicate to the supervisor program that initialization of the device channel is complete. The Executive RTS procedure ONEXCEPTION is called to establish VDTSEXCEPTION as the exception handler and the procedure VDTSWORK is called to perform the transfer of information from INFILE to the screen. The implementaion of VDTSWORK is illustrated in FIGURE 14-18.

```
PROCEDURE VDTINIT$SWORK;
VAR
  CH: CHAR;
  OLDCURSOR: INTEGER;
BEGIN {VDTINIT$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};
    VDTINIT$ROLLUP(OUTPUT_LINE);
    WHILE NOT EOLN(INFILE) DO BEGIN
      READ(INFILE, CH);
      SBZ(SELECT_WORD_W0_W1);
      LDCR( 7, ORD(CH) ) {LOAD INTO CHARACTER BUFFER};
      SBO(SET_LOW_INTENSITY_W0) {LOW INTENSITY};
      SBO(WRITE_DATA_STROBE_W0) {STROBE DATA TO SCREEN};
      SBZ(MOVE_CURSOR_W0) {MOVE CURSOR RIGHT};
      END {WHILE NOT EOLN(INFILE)};
  {RESTORE CURSOR ADDRESS TO OLDCURSOR}
    SBO(SELECT_WORD_W0_W1);
    LDCR(11, OLDCURSOR);
    SIGNAL(EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE);
    READLN(INFILE);
    END {WHILE NOT EOF(INFILE)};
  UNTIL ETERNITY;
END   {VDTINIT$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 VDTSCREEN since INFILE is the master of the channel.

Any exception of VDTSCREEN causes an implicit escape from VDTSWORK and an implicit invocation of VDTSEXCEPTION. The implementation of VDTSEXCEPTION 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 VDTSCREEN with the exception still present.

If the exception is a channel abort, the exception is reset (cleared) and VDTSWORK 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 VDTSWORK 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.

```
PROCEDURE VDTINIT$SEXCEPTION;
BEGIN {VDTINIT$SEXCEPTION}
IF ERR$CLASS = F$$CLASS_FILE_ERROR
  AND ERR$REASON = F$$REASON_CHANNEL_ABORT
  THEN BEGIN
    ERR$RSET {CLEAR ERROR};
    VDTINIT$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   {VDTINIT$SEXCEPTION};
```

FIGURE 14-19. IMPLEMENTATION OF VDT SCREEN EXCEPTION HANDLER.

14.3.3.5 <u>Implementation of VDT Keyboard Logical Device Process</u>. 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 VDTKWORK does the majority of the work after the necessary structures are initialized. The nested procedure VDTKEXCEPTION 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 VDTKEXCEPTION is established as the exception handler and the procedure VDTKWORK is invoked to perform the transfer of information from the keyboard to the user through OUTFILE. A simplified version of VDTKWORK is illustrated in Figure 14-21.

```
PROCESS VDTINIT$KEYBOARD(OUTFILE: TEXT; LEVEL: INTERRUPT_LEVEL);
VAR
  INTERRUPT: SEMAPHORE;
BEGIN {VDTINIT$KEYBOARD}
{PRIORITY = LEVEL;
 STACKSIZE = VDTINIT$KEYBOARD_STACK}
INITSEMAPHORE(INTERRUPT, 0);
EXTERNALEVENT(INTERRUPT, LEVEL) {ASSOCIATE SEMAPHORE WITH
  INTERRUPT LEVEL};
F$MASTER(OUTFILE) {ESTABLISH OUTFILE AS CHANNEL MASTER};
IF FILE_ORIENTED THEN F$XACCESS(OUTFILE);
F$STLENGTH(OUTFILE, SCREEN_LINE_LENGTH) {SET MASIMUM LINE
  LENGTH};
F$ULENGTH(OUTFILE) {ALLOW USER'S SEQUENTIAL FILE OF SHORTER
  COMPONENT LENGTH};
F$STMODE(OUTFILE, WRITING) {SET MODE OF KEYBOARD CHANNEL};
F$CREATECHANNEL(OUTFILE) {CREATE CHANNEL};
SIGNAL(PARTIAL_COMPLETION);
ONEXCEPTION( LOCATION(VDTINIT$KEXCEPTION) ) {ESTABLISH EXCEPTION
  HANDLER};
VDTINIT$KWORK {INVOKE WORK PROCEDURE};
END   {VDTINIT$KEYBOARD};
```

FIGURE 14-20. VDT KEYBOARD LOGICAL DEVICE PROCESS.

```
PROCEDURE VDTINIT$KWORK;
VAR
  CH: CHAR; CURSOR: INTEGER;
BEGIN {VDTINIT$KWORK} REWRITE(OUTFILE);
REPEAT {FOREVER COPYING LINES}
  WAIT(EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE);
  OUTPUT_LINE := 23;
  VDTINIT$ROLLUP(24) {RESERVE LINE AT BOTTOM FOR EDITING};
  "RNABLE CURSOR DISPLAY"
  REPEAT {GET CHARACTER FROM KEYBOARD}
    "ENABLE KEYBOARD INTERRUPT"
    SIGNAL(EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE);
    WAIT(INTERRUPT);
    WAIT(EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE);
    "ACKNOWLEDGE AND DISABLE KEYBOARD INTERRUPT"
    "INPUT CHARACTER FROM KEYBOARD INTO CH"
  {EDIT CHARACTER}
    IF CH = BACKSPACE THEN
      IF COLUMN(OUTFILE) > 0 THEN BEGIN "MOVE CURSOR LEFT"
        F$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 {INDLUDE IN NEXT ROLL UP};
  SIGNAL(EXCLUSIVE_ACCESS_TO-PHYSICAL_DEVICE);
  IF CH = DC3 THEN REWRITE(OUTFILE) ELSE WRITELN(OUTFILE);
  UNTIL ETERNITY;
END   {VDTINIT$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 VDTINIT$HANDLER;
  {HIERARCHICAL RELATIONSHIPS AMONG MODULES IN VDTINIT$HANDLER:

      VDTINIT$HANDLER. . . . . . . .NULL-BODIED SYSTEM ENVIRONMENT
        VDTINIT$SUPERVISOR . . . . .PHYSICAL DEVICE SUPERVISOR PROGRAM
          VDTINIT$ROLLUP . . . . . .INTERNALLY USED PROCEDURE
          VDTINIT$KEYBOARD . . . . .KEYBOARD LOGICAL INTERFACE PROCESS
            VDTINIT$KWORK. . . . . .KEYBOARD WORKER PROCEDURE
            VDTINIT$KEXCEPTION . . .KEYBOARD EXCEPTION HANDLER
          VDTINIT$SCREEN . . . . . .SCREEN LOGICAL INTERFACE PROCESS
            VDTINIT$SWORK. . . . . .SCREEN WORKER PROCEDURE
            VDTINIT$SEXCEPTION . . .SCREEN EXCEPTION HANDLER
        VDTINIT$ . . . . . . . . . .PHYSICAL DEVICE INITIALIZATION
                                    PROCEDURE}

    CONST

      VDTINIT$SUPERVISOR_STACK = 200;
      VDTINIT$SUPERVISOR_HEAP = 400;
      VDTINIT$KEYBOARD_STACK = 200;
      VDTINIT$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. (PAGE 1 of 7)

```
 TYPE
   CRU_ADDRESS = 0..#1FFE;
   INTERRUPT_LEVEL = 0..15;
   ALFA = PACKED ARRAY [ 1..8 ] OF CHAR;
   NONEGg = 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 ROUTIINES}
 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$STLENGTH(VAR F: ANYFILE; LENGTH: INTEGERR; EXTERNAL;
 PROCEDURE F$ULENGTH(VAR F: ANYFILE); EXTERNAL;
 PROCEDURE 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 VDTINIT$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. (PAGE 2 of 7)

```
PROCEDURE VDTINIT$ROLLUP(LAST_LINE: INTEGER);
{ ROLL LINES 1 THROUGH LAST_LINE UP ONE ROW LEAVING LINE
  LAST_LINE BLANK}
  VAR CHBYTE: 0..255; CURSOR, CURSOR_AT_LAST_LINE: INTEGER;
BEGIN {VDTINIT$ROLLUP} ASSERT LAST_LINE > 0 AND LAST_LINE <= 24;
CRUBASE(BASE) {ESTABLISH CRU BASE};
SBZ(SELECT_WORK_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   { VDTINIT$ROLLUP };

PROCESS VDTINIT$KEYBOARD( OUTFILE: TEXT; LEVEL: INTERRUPT_LEVEL );
VAR INTERRUPT: SEMAPHORE;
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM. (PAGE 3 of 7)

```
PROCEDURE VDTINIT$KWORK;
VAR
  CH: CHAR; CURSOR: INTEGER;
BEGIN { VDTINIT$KWORK }
CRUBASE( BASE );
REWRITE( OUTFILE );
REPEAT { FOREVER COPYING LINES }
  WAIT( EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE );
  OUTPUT_LINE := 23;
  VDTINIT$ROLLUP( 24 ) { RESERVE LINE AT BOTTOM FOR EDITING };
  SBO( ENABLE_CURSOR_DISPLAY_W1 );
  REPEAT { GET CHARACTER FROM KEYBOARD }
    SBZ( SELECT_WORD_W0_W1 );
    SBO( ENABLE_BLINKING_CURSOR_W0 );
    SBO( ENABLE_KEYBOARD_INTERRUPT_W0 );
    SIGNAL( EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE );
    WAIT( INTERRUPT );
    WAIT( exclusive_access_to_physical_device );
    SBZ( SELECT_WORD_W0_W1 ) { SELECT WORD ZERO };
    SBZ( ENABLE_KEYBOARD_INTERRUPT_W0 );
    STCR( 15, CH::INTEGER );
    CH := CH::WORD.MOST_SIGNIFICANT_CHAR;
    SBZ( ENABLE_BLINKING_CURSOR_W0 );
    SBO( SELECT_WORD_W0_W1 ) { SELECT WORD ONE };
    SBO( ACKNOWLEDGE_KEYBOARD_W1 );
   { EDIT CHARACTER }
    SBZ( SELECT_WORD_W0_W1 ) { SELECT WORD ZERO };
    IF CH = BACKSPACE THEN
      IF COLUMN( OUTFILE ) > 0
        THEN BEGIN
          SBO( MOVE_CURSOR_W0 ) { MOVE CURSOR LEFT };
          F$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. (PAGE 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   { VDTINIT$KWORK };

      PROCEDURE VDTINIT$KEXCEPTION;
      BEGIN { VDTINIT$KEXCEPTION }
      IF ERR$CLASS = F$$CLASS_FILE_ERROR
        AND ERR$REASON = F$$REASON_CHANNEL_ABORT
        THEN BEGIN
          ERR$RSET { CLEAR ERROR };
          VDTINIT$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   { VDTINIT$KEXCEPTION };

    BEGIN { VDTINIT$KEYBOARD }
    {# PRIORITY = LEVEL;
       STACKSIZEE= VDTINIT&KEYBOARD_STACK }
    INITSEMAPHORE( INTERRUPT, 0 );
    EXTERNALEVENT( INTERRUPT, LEVEL ) { ASSOCIATE SEMAPHORE WITH
      INTERRUPT LEVEL };
    F$MASTER( OUTFILE ) { ESTABLISH OUTFILE AS CHANNEL MASTER };
    IF FILE_ORIENTED THEN F$XACESSs( OUTFILE );
    F$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$CREATECHANNEL( OUTFILE ) { CREATE CHANNEL };
    SIGNAL( PARTIAL_COMPLETION );
    ONEXCEPTION( LOCATION( VDTINIT$KEXCEPTION ) ) { ESTABLISH
      EXCEPTION HANDLER };
    VDTINIT$KWORK { INVOKE WORK PROCEDURE };
    END   { VDTINIT$KEYBOARD };
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM. (PAGE 5 of 7)

```
PROCESS VDTINIT$SCREEN( INFILE: TEXT; LEVEL: INTERRUPT_LEVEL );

  PROCEDURE VDTINIT$SWORK;
  VAR
    CH: CHAR;
    OLDCURSOR: INTEGER;
  BEGIN { VDTINIT$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 };
      VDTINIT$ROLLUP( OUTPUT_LINE );
      WHILE NOT EOLN( INFILE ) DO BEGIN
        READ( INFILE, CH );
        SBZ( SELECT_WORD_W0_W1 );
        LDCR( 7, ORD( CH ) ) { LOAD INTO CHARACTER BUFFER };
        SBO( SET_LOW_INTENSITY_W0 ) { LOW INTENSITY };
        SBO( WRITE_DATA_STROBE_W0 ) { STROBE DATA TO SCREEN };
        SBZ( MOVE_CURSOR_W0 ) { MOVE CURSOR RIGHT };
        END { WHILE NOT EOLN( INFILE ) };
    { RESTORE CURSOR ADDRESS TO OLDCURSOR }
      SBO( SELECT_WORD_W0_W1 );
      LDCR( 11, OLDCURSOR );
      SIGNAL( EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE );
      READLN( INFILE );
      END { WHILE NOT EOF( INFILE ) };
    UNTIL ETERNITY;
  END   { VDTINIT$SWORK };

  PROCEDURE VDTINIT$SEXCEPTION;
  BEGIN { VDTINIT$SEXCEPTION }
  IF ERR$CLASS = F$$CLASS_FILE_ERROR
    AND ERR$REASON = F$$REASON_CHANNEL_ABORT
    THEN BEGIN
      ERR$RSET { CLEAR ERROR };
      VDTINIT$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   { VDTINIT$SEXCEPTION };
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM. (PAGE 6 of 7)

```
      BEGIN { VDTINIT$SCREEN }
      {# PRIORITY = LEVEL;
         STACKSIZE = VDTINIT$SCREEN_STACK }
      F$MASTER( INFILE ) { ESTABLISH INFILE AS CHANNEL MASTER };
      IF FILE_ORIENTED THEN F$XACESSs( 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(VDTINIT$SEXCEPTION) ) { ESTABLISH EXCEPTION
        HANDLER };
      VDTINIT$SWORK { INVOKE SCREEN WORK PROCEDURE };
      END   { VDTINIT$SCREEN };

    BEGIN { VDTINIT$SUPERVISOR }
    {# PRIORITY = LEVEL;
       STACKSIZE = VDTINIT$SUPERVISOR_STACK;
       HEAPSIZE = VDTINIT$SUPERVISOR_HEAP }
    CRUBASE( BASE ) { SET CRU BASE TO INITIALIZE DEVICE };
    SBZ( SELECT_WORD_W0_W1 ) { SELECT WORD ZERO };
    SBO( ENABLE_DISPLAY_W0 ) { ENABLE DISPLAY };
    INITSEMAPHORE( PARTIAL_COMPLETION, 0 );
    INITSEMAPHORE( EXCLUSIVE_ACCESS_TO_PHYSICAL_DEVICE, 1 );
    START VDTINIT$KEYBOARD( FILENAMED( NAME ), LEVEL );
      WAIT( PARTIAL_COMPLETION );
    START VDTINIT$SCREEN( FILENAMED( NAME ), LEVEL );
      WAIT( PARTIAL_COMPLETION );
    SIGNAL( INITIALIZATION_COMPLETE );
    TERMSEMAPHORE( PARTIAL_COMPLETION );
    END   { VDTINIT$SUPERVISOR };

    PROCEDURE VDTINIT$( BASE: CRU_ADDRESS;
      LEVEL: INTERRUPT_LEVEL;
      NAME: ALFA;
      FILE_ORIENTED: BOOLEAN );
    { INITIALIZE A PHYSICAL DEVICE INTERFACE SYSTEM FOR A 911 VDT }
      VAR INITIALIZATION_COMPLETE: SEMAPHORE;
    BEGIN { VDTINIT$ }
    INITSEMAPHORE( INITIALIZATION_COMPLETE, 0 );
    START VDTINIT$SUPERVISOR(
      BASE, LEVEL, FILE_ORIENTED, NAME, INITIALIZATION_COMPLETE );
    WAIT( INITIALIZATION_COMPLETE );
    TERMSEMAPHORE( INITIALIZATION_COMPLETE );
    END   { VDTINIT$ };

  BEGIN {$ NULLBODY }
  END.
```

FIGURE 14-22. VDT PHYSICAL DEVICE INTERFACE SYSTEM. (PAGE 7 of 7)

# SECTION 15

# CONFIGURING TARGET SYSTEMS FOR INTERPRETIVE EXECUTION

## 15.1 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 Pascal System into an object load module for a hypothetical target machine.

## 15.2 CONFIGURING THE MICROPROCESSOR PASCAL SYSTEM INTERPRETIVE RTS FOR THE 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 15-1) will not contain the correct specifications for the system being configured and hence will require some of the modifications outlined in Paragraph 15.2.1 and 15.2.2.

```
       IDT  ´CONFIG´
       TITL ´CONFIG:          CONFIGURATION MODULE´
*
*      THIS MODULE DEFINES THE CONFIGURATION OF THE USER´S
*      MICROPROCESSOR PASCAL SYSTEM
*
       DEF  $RAMTB,$RESTA,$LREX
       DEF  INT$WP,BAD$WP
       DEF  I1WP$,I2WP$,I3WP$,I4WP$,I5WP$,I6WP$,I7WP$
       DEF  I8WP$,I9WP$,I10WP$,I11WP$,I12WP$,I13WP$
       DEF  I14WP$,I15WP$
**********************************************************
* THIS MODULE SPECIFIES THE FOLLOWING CONFIGURATION      *
* PARAMETERS:                                            *
*    1) INTERRUPT WORKSPACES (I1WP$-I15WP$)              *
*       THESE ARE THE WORKSPACES 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 15-1. 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
IWP$SZ EQU  >18
       DORG STATIC+IWP$SZ->20
I1WP$  BSS  IWP$SZ
I2WP$  BSS  IWP$SZ
I3WP$  BSS  IWP$SZ
I4WP$  BSS  IWP$SZ
I5WP$  BSS  IWP$SZ
I6WP$  BSS  IWP$SZ
I7WP$  BSS  IWP$SZ
I8WP$  BSS  IWP$SZ
I9WP$  BSS  IWP$SZ
I10WP$ BSS  IWP$SZ
I11WP$ BSS  IWP$SZ
I12WP$ BSS  IWP$SZ
I13WP$ BSS  IWP$SZ
I14WP$ BSS  IWP$SZ
I15WP$ BSS  IWP$SZ
       BSS  >20-IWP$SZ        KEEP FROM OVERLAPPING
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 15-1. CONFIG. (Sheet 2 of 2)

### 15.2.1 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 is shown in Figure 15-2.

Figure 15-3 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 15-2. SIMPLE RAM TABLE.

```
        IDT  ´CONFIG´
        TITL ´CONFIG:          CONFIGURATION MODULE´
*
*       THIS MODULE DEFINES THE CONFIGURATION OF THE USER´S
*       MICROPROCESSOR PASCAL SYSTEM.
*
        DEF  $RAMTB,$RESTA,$LREX
        DEF  INT$WP,BAD$WP
        DEF  I1WP$,I2WP$,I3WP$,I4WP$,I5WP$,I6WP$,I7WP$
        DEF  I8WP$,I9WP$,I10WP$,I11WP$,I12WP$,I13WP$
        DEF  I14WP$,I15WP$
STATIC EQU  >A000
IWP$SZ EQU  >18
       DORG STATIC+IWP$SZ->20
I1WP$  BSS  IWP$SZ
I2WP$  BSS  IWP$SZ
I3WP$  BSS  IWP$SZ
I4WP$  BSS  IWP$SZ
I5WP$  BSS  IWP$SZ
I6WP$  BSS  IWP$SZ
I7WP$  BSS  IWP$SZ
I8WP$  BSS  IWP$SZ
I9WP$  BSS  IWP$SZ
I10WP$ BSS  IWP$SZ
I11WP$ BSS  IWP$SZ
I12WP$ BSS  IWP$SZ
I13WP$ BSS  IWP$SZ
I14WP$ BSS  IWP$SZ
I15WP$ BSS  IWP$SZ
       BSS  >20-IWP$SZ          KEEP FROM OVERLAPPING
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                   RESTART IS LEVEL 0 INTERRUPT
$LREX  EQU  0                   NO LREX BLWP VECTOR
       END
```

FIGURE 15-3. USE OF RAM TABLE IN CONFIG.

Note that the RAM table in Figure 15-3 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 15-3 the first >188 bytes of real RAM, locations >A000 through >A187, are not included in the RAM table. Instead, these locations are used for the allocation of sixteen workspaces, "I1WP$"

to "I15WP$" and "BAD$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.

### 15.2.2 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.

### 15.2.3 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 15-1, defines seventeen workspaces. InWP$ defines the interrupt workspace for interrupts at level "n". These are the workspaces used for interrupt handling in the RTS. BAD$WP is the bad interrupt and XOP workspace. This is the workspace which is usually specified in the transfer vectors of all unimplemented interrupts and XOPS (See Section 17.4). INT$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 I1WP$-I15WP$ and BAD$WP) should be allocated in RAM space which has been excluded from the RAM table.

The workspace for an unimplemented interrupt at level n may be omitted to save space by simply changing the line

```
     InWP$    BSS  IWP$SZ
```

to the following and moving the line after the declaration of BAD$WP.

```
     InWP$    EQU  BAD$WP
```

This specifies the BAD$WP workspace as the workspace for the level n interrupt. Unimplemented interrupts for level n would then behave like XOPs, should one occur.

### 15.2.4 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 15-4 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
*       MICROPROCESSOR PASCAL SYSTEM.
*
        DEF  $RAMTB,$RESTA,$LREX
        DEF  INT$WP,BAD$WP
        DEF  I1WP$,I2WP$,I3WP$,I4WP$,I5WP$,I6WP$,I7WP$
        DEF  I8WP$,I9WP$,I10WP$,I11WP$,I12WP$,I13WP$
        DEF  I14WP$,I15WP$
        REF  BGN$PC                   <--- ADDED
STATIC EQU  >B000                   <--- CHANGED
IWP$SZ EQU  >18
        DORG STATIC+IWP$SZ->20
I1WP$   BSS  IWP$SZ
I2WP$   BSS  IWP$SZ
I3WP$   BSS  IWP$SZ
I4WP$   BSS  IWP$SZ
I5WP$   BSS  IWP$SZ
I6WP$   BSS  IWP$SZ
I7WP$   BSS  IWP$SZ
I8WP$   BSS  IWP$SZ
I9WP$   BSS  IWP$SZ
I10WP$  BSS  IWP$SZ
I11WP$  BSS  IWP$SZ
I12WP$  BSS  IWP$SZ
I13WP$  BSS  IWP$SZ
I14WP$  BSS  IWP$SZ
I15WP$  BSS  IWP$SZ
        BSS  >20-IWP$SZ        KEEP FROM OVERLAPPING
BAD$WP  BSS  >20
BGN$WP  BSS  >20                      <--- ADDED
INT$WP  EQU  $                 THIS WORKSPACE MUST BE LAST!
*****************************************************************
        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 15-4. CONFIG WITH USER MODIFICATIONS.

The RAM table of Figure 15-4 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.

## 15.3 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 15-5.

```
        IDT  'USERINIT'
* ROUTINE LIST:
*       RSET$, SYS$CR
* COPY MODULES:
*       NONE
* MACRO DEFINITIONS:
*       NONE
* EXTERNAL ROUTINES:
*       SEG0-SEG63
* EXTERNAL DATA:
*       $EXEC, $LREX, $RAMTB, $RESTA, BAD$WP, INT$WP,
*       I1WP$, I2WP$, I3WP$, I4WP$, I5WP$, I6WP$,
*       I7WP$, I8WP$, I9WP$, I10WP$, I11WP$, I12WP$,
*       I13WP$, I14WP$, I15WP$
* MODULE CONSTANTS:
R0      EQU  0
R1      EQU  1
R2      EQU  2
R12     EQU  12
        DEF  SYS$CR,IN$PC,$SEGTB,RSET$
        REF  INT$WP,BAD$WP
        REF  $RAMTB,$EXEC,$RESTA,$LREX
        REF  I1WP$,I2WP$,I3WP$,I4WP$,I5WP$,I6WP$,I7WP$,I8WP$
        REF  I9WP$,I10WP$,I11WP$,I12WP$,I13WP$,I14WP$,I15WP$
        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 15-5. USERINIT. (Sheet 1 of 6)

```
************************************************************
*   IF THIS MODULE IS LOADED AT ADDRESS ZERO, THEN         *
*   THE TRAP VECTORS (WHICH FOLLOW) OF THE MACHINE         *
*   ARE ALREADY INITIALIZED.  OTHERWISE, MICROPROCESSOR    *
*   PASCAL INITIALIZATION MOVES THE FOLLOWING DATA TO      *
*   ABSOLUTE ADDRESS ZERO.                                 *
************************************************************
TRAPS   EQU  $                  CONFIGURATION OF TRAP VECTORS
        DATA INT$WP,RSET$       LEVEL 0
        DATA I1WP$,IN$PC        LEVEL 1
        DATA I2WP$,IN$PC        LEVEL 2
        DATA I3WP$,IN$PC        LEVEL 3
        DATA I4WP$,IN$PC        LEVEL 4
        DATA I5WP$,IN$PC        LEVEL 5
        DATA I6WP$,IN$PC        LEVEL 6
        DATA I7WP$,IN$PC        LEVEL 7
        DATA I8WP$,IN$PC        LEVEL 8
        DATA I9WP$,IN$PC        LEVEL 9
        DATA I10WP$,IN$PC       LEVEL 10
        DATA I11WP$,IN$PC       LEVEL 11
        DATA I12WP$,IN$PC       LEVEL 12
        DATA I13WP$,IN$PC       LEVEL 13
        DATA I14WP$,IN$PC       LEVEL 14
        DATA I15WP$,IN$PC       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'
        15-
```

FIGURE 15-5. USERINIT. (Sheet 2 of 6)

```
* 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 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:
*       LIMI 0
*       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 15-5. USERINIT. (Sheet 3 of 6)

```
***********************************************************
* 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
       CLR   R5                  INSTRUCTION JUST TO CLEAR BUS
       C     @ALTZER,*R0
       JNE   EOB
       MOV   @ALTONE,*R0
       CLR   R5                  INSTRUCTION JUST TO CLEAR BUS
       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 15-5. USERINIT. (Sheet 4 of 6)

```
************************************************************
*  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.
*
***********************************************************
```

FIGURE 15-5. USERINIT. (Sheet 5 of 6)

```
IN$PC  EQU  $                    SERVICE INTERRUPT
       LIMI 0
       MOV  R7,R7
       JEQ  NO$ASM
       BLWP R7                   WP NOT ZERO, DO ASM LANG
NO$ASM EQU  $
*
* PASS INTERPRETER´S WORKSPACE IN R9
* PASS ADDRESS OF SEG63, PROCEDURE 1 IN R10
*
       LI   R9,INT$WP
       LI   R10,SEG63
       MOV  @2(R10),R10
       AI   R10,SEG63
       B    *R10                 BRANCH TO RTS
*                                  (SEG63, ENTRY 1)
**********************************************************
* CODE FOR STANDARD TM990 SYSTEM
**********************************************************
BAD$PC EQU  $
       LIMI 0
       JMP  SYS$CR
*
SYS$CR EQU  $
       LIMI 0
       IDLE
       JMP  SYS$CR
*
       END  RSET$               ENTRY POINT OF STANDARD SYSTEM
```

FIGURE 15-5. USERINIT. (Sheet 6 of 6)

## 15.3.1 Assembly Language Interrupt Handlers

There are two different ways that the user may handle interrupts using assembly language for interpretive execution:

1) Pure assembly language handling of interrupts outside the Microprocessor Pascal environment. All communication with the Microprocessor Pascal environment should be through common memory areas.

2) An interrupt handling capability in which the user has the option of handling an interrupt with both assembly language and pascal routines. This system uses a procedure (ASSEMBLYEVENT) that is declared as an external procedure in thee user's Pascal code. NOTE: ASSEMBLYEVENT is not supported in debug mode either the host or target system.

For a complete discussion of Interrupt Handling in Pascal using EXTERNALEVENT and ALTEXTERNALEVENT see Paragraph 9.5 of this manual.

15.3.1.1 Pure Assembly Language Interrupt Handlers. The module "USERINIT" contains the interrupt transfer vectors. Each interrupt level that is to convey valid interrupts must have its own dedicated workspace. All interrupt levels that are to be considered as conveying spurious interrupts share a common workspace as described in Paragraph 15.2.3 above. The program counter to be provided in the transfer vector should be the entry point of the assembly language code of the handler for interrupts at that level.

Thus the "USERINIT" module should be changed to provide the symbols for the dedicated workspace(s) and program counter(s). The "CONFIG" module should also be modified so that each workspace is allocated in RAM outside the RAM area available for MPIX.

The very first instruction of the assembly language code must be a "LIMI 0". Once the interrupt mask is set it should not be changed while the interrupt handler is executing. The mask is reset by the processor when the RTWP, the last instruction in the interrupt handler, is executed. This interrupt handler is executing outside the Pascal environment and communication with Pascal code is possible only via common memory areas.

15.3.1.2 Declaration and Calling Conventions of ASSEMBLYEVENT. The ASSEMBLYEVENT routine is declared by the user to be external in his Microprocessor Pascal code prior to the receipt of any interrupts. The format of the declaration is

```
PROCEDURE ASSEMBLYEVENT(VAR WPX: WORKSPACE; PCX: INTEGER;
                        LEVELX: INTERRUPT_LEVEL); EXTERNAL;
```

For further details see Paragraph 9.5, Section 9.

Unlike the pure assembly language handling of interrupts the modules "USERINIT" and "CONFIG" should provide a single workspace for all interrupt levels. In addition to this, as indicated in the procedure declaration above, the Pascal code must supply the user´s assembly language code with its own workspace and the entry point of that code. The Pascal Type ´workspace´ is declared as below.

```
  TYPE   WORKSPACE = ARRAY[1..16] OF INTEGER;
  {* and for example *}
  VAR    WP : WORKSPACE;
  {* and *}
  PROCEDURE ASM3; EXTERNAL;
```

The allocation of an assembly language interrupt handler (for example called "ASM3") to interrupt level three then looks as follows:

```
        ASSEMBLYEVENT(WP,LOCATION(ASM3),3);
```

Or if the workspace structure is dynamically allocated from the heap by,

```
        NEW(POINTER);
```

where POINTER is declared as

```
        POINTER: @WORKSPACE;
```

the allocation of the assembly language interrupt handler would then look like the following:

```
        ASSEMBLYEVENT(POINTER@,LOCATION(ASM3),3);
```

Another external routine is provided for the user to sever the connection of an assembly language interrupt handler with any particular interrupt level. The procedure is called NOASSEMBLYEVENT and should be declared as follows in the user´s Microprocessor Pascal code:

```
        PROCEDURE NOASSEMBLYEVENT(LEVEL: INTERRUPT_LEVEL); EXTERNAL;
```

Thus, a call to NOASSEMBLYEVENT, to deallocate an assembly language interrupt handler from level 3, looks as follows:

```
        NOASSEMBLYEVENT(3);
```

For further details see Paragraph 9.5.

15.3.1.3 Use of ASSEMBLYEVENT. In MPIX, interrupts that are to be handled via an ASSEMBLYEVENT, are initially dealt with by the System interrupt Handler in the same workspace, IWP$, with the transfer vector in the "USERINIT" module set to a different program counter, InPC (where ´n´ is the interrupt level). When an interrupt occurs a branch through the appropriate vector is executed. The system interrupt handler first looks to see if any user written interrupt handling procedures have been attached to the interrupt level at which the interrupt occured, by the execution of an ASSEMBLYEVENT. If this is the case then the interrupt mask is set to zero (ie all interrupts are masked) and control is then passed to the entry point of this assembly language procedure.

NOTE: The first two parameters of the call to ASSEMBLYEVENT are a transfer vector. The interrupt mask should not be altered inside the interrupt handler. When the assembly language handling of interrupts is completed, the user has two choices of action:

1) If he wishes to perform processing of this interrupt in Microprocessor Pascal in addition to the his assembly language handler, the user simply executes an ´RTWP´ instruction. In this case the return is into the context of the System Interrupt Handler. The System Interrupt Handler then proceeds to look for any Pascal semaphore that is allocated to this interrupt level by EXTERNALEVENT. If one is not found or the EXTERNALEVENT semaphore has no processes waiting on it, then the System interrupt handler looks for an ALTEXTERNALEVENT semaphore. If this is not found or if it also has no processes waiting on it then the system interrupt handler will assume that the interrupt was spurious and a system crash will occur. If a process is found waiting on either semaphore the semaphore will be signaled, the user´s interrupted workspace will be restored and the Pascal interrupt handler become active.

2) If no further processing of this interrupt is required then the user may return directly to the interrupted workspace without invoking any Pascal interrupt handlers. There are many different ways of actually doing this in assembly language. The following is just one example, using 3 words of code.

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

This code causes two ´RTWP´ instructions to be executed in a row.

The MPIX ASSEMBLYEVENT method of handling interrupts is intended to be used in situations where it is essential to respond to interrupts very quickly and yet desirable to do the interrupt initialization at the Pascal level. Relative to handling interrupts entirely in Pascal, use of ASSEMBLYEVENT trades a faster interrupt response time for an Assembly coded interrupt service routine.

An example interrupt service routine to be connected by ASSEMBLYEVENT is presented below. Note: In this revision, the entry point for an assembly language interrupt handler must be an odd number for MPIX.

```
       IDT  'TIC
***********************************************************
*                                                         *
*                                                         *
*                                                         *
*     INTERRUPT SERVICE ROUTINE "INSTALLED" BY            *
*               ASSEMBLYEVENT                             *
*                                                         *
***********************************************************
       DEF  TIC
       EVEN                       forces word alignment
TIC    EQU  $+1                   defines entry point
***********************************************************
       LI   R12,>100              point to TMS9901
       SBO  3                     reset the interrupt
       INC  R0                    R0 := R0 + 1 MOD 1000;
       CI   R0,1000
       JL   BYEBYE
       CLR  R0
BYEBYE LI   R14,R
R      RTWP                       return to the caller
       END
```

The Pascal mainline code which does the interrupt initialization is presented below.

```
SYSTEM  ASMBYEV;

TYPE

  INTLVL = 0..15;

  WP =  ARRAY[0..15] OF INTEGER;

PROCEDURE TIC  ; EXTERNAL;
PROCEDURE INITSEMAPHORE ( VAR S : SEMAPHORE ; VALUE : INTEGER);
          EXTERNAL;
PROCEDURE ASSEMBLYEVENT ( VAR X: WP; PC: INTEGER; LEVEL : INTLVL );
          EXTERNAL;
PROCEDURE WAIT ( S : SEMAPHORE ); EXTERNAL;
```

```
PROGRAM CLKINT;

(* PURPOSE:
         PROVIDE REAL TIME CLOCK FOR MPIX :
         INITIALIZE TMS 9901 FOR 100.02 MS CLOCK INTERRUPTS     *)

CONST

  CLKCRU  = #0100;
  CLKLVL  =     3;
  INTBIT  =     0;
  CLKBIT  =     3;
  COUNTS  =  4689;

VAR
  INTERRUPT, SOME_OTHER: SEMAPHORE;
  INTERRUPT_PC: INTEGER;
  INTERRUPT_WP: WP;

BEGIN                                                   (* PROGRAM CLKINT *)
(*#  STACKSIZE = 256; HEAPSIZE = 64; PRIORITY = CLKLVL       *)
  INITSEMAPHORE ( INTERRUPT, 0 );
  INTERRUPT_PC := LOCATION( TIC );
  ASSEMBLYEVENT( INTERRUPT_WP, INTERRUPT_PC, CLKLVL );
  INTERRUPT_WP[0] := 0;
  CRUBASE ( CLKCRU  );
  LDCR( 15, COUNTS * 2 + 1 );
  SBZ ( INTBIT );                  (* SET INTERRUPT ON *)
  SBO ( CLKBIT );               (* ENABLE CLOCK INTERRUPT *)
  "TIC:     INTERRUPT_WP[0] := INTERRUPT_WP[0] + 1 MOD 1000;
END;                                                      (* CLKINT *)

BEGIN                  (*# STACKSIZE = 256   *)           (* ASMBYEV *)
  START CLKINT;
END.                                                      (* ASMBYEV *)
```

If necessary, the execution of a Pascal segment of code can follow the Assembly language segment if EXTERNALEVENT also couples the same semaphore used in the ASSEMBLYEVENT routine. In this case the Pascal code must execute a WAIT on this semaphore before the first interrupt is allowed to occur. In addition, the second RTWP must be eliminated from the Assembly coded module (delete: LI R14,R ).

For example, after the comment in CLKINT, the user could add:

```
REPEAT
  WAIT(INTERRUPT);
  SIGNAL(SOME_OTHER);
UNTIL FALSE
```

provided EXTERNALEVENT was called prior to the loop with the same semaphore.

## 15.3.2 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 15-6) 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 15-6. STANDARD CRASH CODE.

More elaborate crash routines can be developed. Figure 15-7 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 15-7. ELABORATE CRASH ROUTINE.

## 15.4 ASSEMBLY LANGUAGE CODING CONVENTIONS

It is possible by following the conventions outlined in this section to write assembly language routines which are callable from Microprocessor Pascal. A segment may consist of assembly language routines which may be declared and called as any other external procedures or functions.

### 15.4.1 General Format and Example of Assembly Language Segment.

Figure 15-8 gives a general format for an assembly language segment. Figure 15-9 gives an example of an assembly language segment written according to this format.

```
        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)
routx   EQU   ((x-1)*256 + SEGCOD)

        DEF   SEGn                   segment dictionary
SEGn    DATA  entry1-SEGn            displacement to first entry point
        DATA  entry2-SEGn            displacement to second entry point
        DATA  entryx-SEGn            displacement to ´x´th entry point

LIT     DATA  value                  local literal


entry1  DATA  0                      assembly language flag
        DATA  parms*2                parameter size (bytes)
        ...
*    code for routine
        ...
*    use of 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


entryx  DATA  0                     assembly language flag
        DATA  parms*2               parameter size ( bytes )
        ...
*    code for routine
        ...
        B    *R11                   return
        END
```

FIGURE 15-8. ASSEMBLY LANGUAGE SEGMENT.

This assembly language segment example contains just one external procedure which performs XOP 15 with one parameter passed by address.

```
       IDT  'SEG2'
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 POINTER
MKSIZ  EQU  -14            ADMINISTRATION AREA SIZE
SEGCOD EQU  5              (SEGMENT NUMBER*2)+1
       DEF  SVC
SVC    EQU  SEGCOD
       DEF  SEG2
SEG2   DATA ENTRY1-SEG2
ENTRY1 DATA 0
       DATA 2
       MOV  *SP,R0        GET ADDRESS OF SVC CALL BLOCK
       XOP  *R0,15        INITIATE THE I/O
       B    *R11
       END
```

FIGURE 15-9. ASSEMBLY LANGUAGE SEGMENT.

### 15.4.2 Details of Assembly Language Segment Conventions.

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 and must remain in R13 - R15 of the callee´s workspace.

# 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 allowed 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 an 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

| | |
|---|---|
| BATCH | - BACKROUND COMPILE |
| CODEGEN | - GENERATE NATIVE CODE |
| COLLECT | - COLLECT RUN TIME SUPPORT |
| COMPILE | - COMPILE SYSTEM |
| COPYSRC | - SOURCE PREPROCESSOR |
| DEBUG | - DEBUG SYSTEM |
| DELETE | - DELETE TEMPORARY FILES |
| DXSC | - DISKETTE CONVERSION |
| EDIT | - EDIT MODULE |
| EXECUTE | - EXECUTE PROGRAM |
| GENMAP | - GENERATE ROUTINE MAP |
| PRINT | - PRINT FILE |
| PURGE | - PURGE SYNONYMS |
| RASS | - REVERSE ASSEMBLE OBJECT |
| QUIT | - QUIT SESSION |
| SAVE | - SAVE SEGMENT |
| SCI | - EXECUTE "SCI" COMMAND |
| SHOW | - SHOW FILE |
| SPLIT | - SPLIT OBJECT |
| WAIT | - WAIT ON BACKGROUND |

### AMPLUS COMMAND SUMMARY

* * INTERPRETIVE-CODE SOFTWARE DEVELOPMENT DISKETTE * *
* * NON FLOATING-POINT SYSTEM BOOTSTRAP * *

| | | | |
|---|---|---|---|
| COMPILE | - COMPILE A SYSTEM | EXECUTE | - EXECUTE PROGRAM |
| DEBUG | - SOURCE DEBUGGER | SAVE | - SAVE SEGMENT |
| EDIT | - TEXT EDITOR | | |
| COPY | - COPY "TEXT" FILE | SHOW | - SHOW "TEXT" FILE |
| AMPL | - INTERPRETIVE-CODE TARGET DEBUGGER AMPL PROCEDURES | | |

```
* * NATIVE-CODE SOFTWARE DEVELOPMENT DISKETTE          * *
* *      FLOATING-POINT SYSTEM BOOTSTRAP               * *

     COMPILE - COMPILE A SYSTEM     GENMAP  -  GENERATE AMPL MAP
     CODEGEN - CODEGEN A SYSTEM     RASS    -  REVERSE ASSEMBLER
     EDIT    - TEXT EDITOR          SPLIT   -  SPLIT OBJECT MODULES

     COPY    - COPY "TEXT" FILE     SHOW    -  SHOW "TEXT" FILE

     AMPL    - NATIVE-CODE TARGET DEBUGGER AMPL PROCEDURES
```

## 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 Indirect |
| 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 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 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

## INTERPRETIVE RUN TIME SUPPORT REFERENCE CARD

This appendix lists the user-callable RTS routines. This listing is presented as a quick reference for the programmer. The enumeration is categorized; within each category routines are alphabetized. Unless otherwise stated, the routines listed are applicable to MPIX.

## 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;
procedure assemblyevent( var interrupt_wp: wp;
  interrupt_pc: integer; level: interrupt_level); external;
proedure noassemblyevent( level: interrupt_level); exteral;
```

## 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 (MPIX ONLY)

```
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 HOST RUN-TIME ERROR MESSAGES

The following list consists of all of the errors that can be generated by the host executive while using the Microprocessor Pascal System. If one of these errors is generated under Amplus a message of the following form will be displayed:

```
      INTERPRETER ERROR : ee
```

where the "ee" represents the specific error that occurred. If one of these errors is generated under DX, a message of the following form will be displayed:

```
      Class/Reason = 00ee
```

the meanings associated with these digits (hexadecimal) 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 Amplus 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 Amplus Software System User's Manual (MP375), 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 Amplus 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 or semaphores 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 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 AMPLUS/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 | (Amplus) |
| | Luno Previously Assigned | (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 | |

## E.6 Code Generator Error Messages

The following error messages are generated by the code generator when an internal error has occurred. With few exceptions, the user should contact the TI MOS Hot Line to resolve the problem signaled.

BADOP <number> IN STATEMENT <number>

BAD OPERAND <number> IN STATEMENT <number>

BAD STATE <number> IN STATEMENT <number>

STATEMENT <number> TOO COMPLEX -- NO REGISTERS
  The user may correct this error by simplifing the statement indicated.

TEMPORARIES NOT FREED IN STATEMENT <number>

The following error messages are fatal internal errors; the code generation process will stop if one of these errors occurs.

CANNOT FIND LABEL

END OF FILE ON PCODE

INVALID LABEL

SET LITERAL TOO LONG

STACK OVERFLOW

STRING LITERAL TOO LONG

TOO MANY COMMONS REFERENCED

TOO MANY EXTERNALS REFERENCED

The following is a list of the task code error messages passed back by the DX10 operaton system. In most cases they point to a hardware problem in the system. These should be reported directly to your local systems engineer. If the fault is believed to be caused by the microprocessor Pascal System, the user should have his source and any other pertinent data collected and sent to the HOTLINE for evaluation.

| Error Code | Meaning |
|---|---|
| 01 | A nonrecoverable memory parity error occurred. |
| 02 | The task tried to execute an undefined instruction. |
| 03 | The task accessed an illegal TILINE address; the illegal address could be an address of a memory location that is not provided for the system installed. |
| 04 | The task tried a supervisor call with an illegal supervisor call code. |
| 05 | The task tried to access a memory address outside of its memory area. |
| 06 | The task tried to execute a privileged instruction. |
| 07 | The task was terminated with a kill task SVC. |
| 08 | The installed memory configuration is not big enough to allow the task to be loaded. |
| 09 | The accessed map segment was not present in memory. |
| OA | An execute protection violation occurred. |
| OB | The task performed a write to a write protected segment. |
| OC | The task caused a condition where the stack parameters were exceeded (stack overflow). |
| OD | A hardware breakpoint address error occurred. |

|  |  |
|---|---|
| OE | Time out error (the 12 ms. clock expired). |
| OF | An overflow protection violation occurred. |
| 10 | Task aborted by terminal (Reset, CMD key sequence) |

NOTE

In both the device and task error messages, a station ID of FF(HEX) means no station.

# APPENDIX F

# MICROPROCESSOR PASCAL SYSTEM VS WIRTH´S PASCAL

## F.1 SPECIAL SYMBOLS

The 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 supported 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

# Microprocessor 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.

## G .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 <set> 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 | G- 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.

The software base address is used rather than the hardware base address. For example on the 101 board, the CRU base of 9901 is 100 in MPP and 80 in TMX.

## 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.

CAUTION: It is important to realize that the procedure INITSEMAPHORE allocates a new semaphore every time it is called in the Microprocessor Pascal System. In TIPMX, it doesn´t.

## H.5 MEMORY MANAGEMENT

Heap is allocated differently in TIPMX and MPP. In MPP, a nested heap is allocated out of another heap, called the parent, so that a hierarchy of heaps may be created. TIPMX maintains a free pool of memory from which all heap requests are allocated. Heap "packets" are allocated on a first fit basis for stack regions, process records, and general user work area.

## H.6 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                                                          { CLOCK }
  WHILE TRUE DO
    BEGIN
      FOR I := 1 TO INTERRUPTS DO
        BEGIN
          CKON$;
          WAITINTERRUPT;
          CKOF$
        END;
      SIGNAL(TICK)
    END;
END;                                                           { CLOCK }
```

EXECUTIVE RTS:

```
PROGRAM CLOCK(INTERRUPTS:INTEGER);
VAR
  INTERRUPT: SEMAPHORE;
BEGIN                                                          { CLOCK }
{# 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;
      SIGNAL(TICK)
    END;
END;                                                           { CLOCK }
```

## H.7 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:

```
                 active
                 process

               +--------+   +--------+   +--------+   +--------+
               | A:  3  |--->| C:  7  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+
       ready
        B: 5
               +--------+   +--------+   +--------+   +--------+   +--------+
               | A:  3  |--->| C:  7  |--->| B:  5  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+   +--------+
```

(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:

```
                 active
                 process

               +--------+   +--------+   +--------+   +--------+
               | A:  3  |--->| C:  7  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+
       ready
        B: 5
               +--------+   +--------+   +--------+   +--------+   +--------+
               | A:  3  |--->| B:  5  |--->| C:  7  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+   +--------+
```

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:

```
                 active
                 process

               +--------+   +--------+   +--------+   +--------+
               | A:  3  |--->| C:  7  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+
       ready
        B: 3
               +--------+   +--------+   +--------+   +--------+   +--------+
               | B:  3  |--->| A:  3  |--->| C:  7  |--->| D: 16  |--->| idle   |
               +--------+   +--------+   +--------+   +--------+   +--------+
```

## H.8 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>
```

## .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).

```
#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.

```
            ---------------
             configuration       Address of RAM configuration table
  -#08     ---------------
              address of         Entry point address of interpreter
             interp. entry       (REF and DATA of the symbol $EXEC)
  -#06     ---------------
             crash address       Address in user-written code to service
                                 a system crash.  Crash code is in R0.
  -#04     ---------------
              address of         Address of BLWP vector used by RE$START
             re$start BLWP       routine.
  -#02     ---------------
              address of         Address of BLWP vector which is copied
               LREX BLWP         to >FFFC.
   #00     --------------- <-- Address of segment table
             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.1.4 Interrupt Workspace Record

The following record is a typical workspace used for fielding interrupts. Each workspace pointer in low memory points to an interrupt transfer workspace which is initialized upon system startup. The workspaces for the different interrupt levels are overlaid so that registers R0-R3 should never be used.

```
#00  -----------------  <-R0
             *
             *
             *            Unused
             *
#08  -----------------
         scratch          Scratch registers
        registers
             *
             *
             *
#0E  -----------------
     Int ASSEMBLYEVENT    Interrupt workspace specified
     UP  (or Level 0)     in ASSEMBLYEVENT (or zero)
#10  -----------------
     int ASSEMBLYEVENT    Interrupt PC specified in
     PC (or Level 0)      ASSEMBLYEVENT (or zero)
#12  -----------------
     interpreter WP       Interpreter Workspace

#14  -----------------
         standard         Address of standard interrupt
           Code           interrupt code
#16  -----------------
         linkage          Linkage register

#18  -----------------
        interrupt         Interrupt Level of this
          Level           workspace
#1B  -----------------
         return           Addresses of previous routine's
         context          PC,WP, and Status register.
            *
#1F  -----------------
```

## 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 fundamentail 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  ---------------
```

```
#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    * Index in dictionary of segment where
     dictionary entry     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       * Number of times currently nested within
    |    count      |     RTS code.
#45  ---------------
```

| | | |
|---|---|---|
| #45 | packed data | Unused (4 bits)<br><br>* 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 verify 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  ---------------
       next pathname      Address of next pathname
#10  ---------------
     minimum buffers      Minimum channel buffers
#11  ---------------
        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)
#12  ---------------
        packed data       Component length (13 bits)

                          File state - closed=0,
                          open-writing=1, eoc-writing=2,
                          open-reading=3, eof-reading=4 (3 bits)
#14  ---------------
       verification       Address of this file descriptor
#16  ---------------
```

```
#16  ---------------
         master           Address of master record
#18  ---------------
     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.
#1A  ---------------
         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, eof-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 is 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
        ---------------
   #06       level             Level associated with semaphore
        --------------
```

## 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 for process variables
  stack frame
---------------


--------------- <-- stack base
  first mark         Administration area for first routine
---------------
     first           Stack frame for first routine
  stack frame
---------------
       *
       *             Stack frame of intermediate routines
       *
---------------
 current mark        Administration area for current routine
--------------- <-- local base
     local           Stack frame for currently active routine
  stack frame
---------------
   next mark         Administration area for next routine
---------------
   evaluation        Area where expressions are evaluated and
     stack           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         Structured value parameters are copied
          parameters         into this area.
        ---------------
            local            Local variables
          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         AMPL breakpoint table
         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 appropriate 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 generated, 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. $MACRO pcode

| op | name | before - stack - after | |
|---|---|---|---|
| $END | | | |
| 00 | EQ | OPERAND2<br>OPERAND1 | OPERAND1 = OPERAND2 |
| 01 | GE | OPERAND2<br>OPERAND1 | OPERAND1 >= OPERAND2 |
| 02 | GT | OPERAND2<br>OPERAND1 | OPERAND1 > OPERAND2 |
| 03 | LE | OPERAND2<br>OPERAND1 | OPERAND1 <= OPERAND2 |
| 04 | LT | OPERAND2<br>OPERAND1 | OPERAND1 < OPERAND2 |
| 05 | NE | OPERAND2<br>OPERAND1 | OPERAND1 <> OPERAND2 |
| 06 | ADD | OPERAND2<br>OPERAND1 | OPERAND1 + OPERAND2 |
| 07 | DIV | OPERAND2<br>OPERAND1 | OPERAND1 / OPERAND2 |
| 08 | MOD | OPERAND2<br>OPERAND1 | OPERAND1 MOD OPERAND2 |
| 09 | MUL | OPERAND2<br>OPERAND1 | OPERAND1 * OPERAND2 |
| 0A | SUB | OPERAND2<br>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 | |

| | | | |
|---|---|---|---|
| 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 |
| 26 | IDX,V | LOWER INDEX ADDRESS | ADDRESS := ADDRESS + (INDEX - LOWER) * V * 2; |
| 27 | IDX0,V | INDEX ADDRESS | ADDRESS := ADDRESS + INDEX * V * 2; |

| | | | | |
|---|---|---|---|---|
| 28 | IDX1,V | INDEX<br>ADDRESS | ADDRESS := ADDRESS + (INDEX - 1) * V * 2; | |
| 29 | CIDX,W | LOWER<br>INDEX | LOWER<br>INDEX | check if INDEX is between<br>LOWER and W (upper bound) |
| 2A | CIDX0,W | INDEX | INDEX | check if INDEX is between<br>0 and W (upper bound) |
| 2B | CIDX1,W | INDEX | INDEX | check if INDEX is between<br>1 and W (upper bound |
| 2C | CSUB,W1,W2 | VALUE | VALUE | check if VALUE is between<br>W1 (lower) and W2 (upper) |
| 2D | JMPG,UB,V | jump to label UB if value at local frame + V words<br>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<br>greater than value at local frame + V + 1 words | | |
| 2F | JMPL,UB,V | jump to label UB if value at local frame + V words<br>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<br>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 | |
| 39 | DIF | SET2<br>SET1 | SET1 - SET2 | |
| 3A | INN | SET<br>VALUE | VALUE IN SET | |
| 3B | INT | SET2<br>SET1 | SET1 * SET2 | |
| 3C | SEL | VALUE | [ VALUE ] | |

| | | | |
|---|---|---|---|
| 3D | SRG | UPPER<br>LOWER | [ LOWER .. UPPER ] |
| 3E | UNI | SET2<br>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<br>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<br>DEST | move V words from SOURCE address to DEST address |
| 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 | load field in word at ADDRESS starting |

| | | | |
|---|---|---|---|
| | | ADDRESS | at bit BITDISP of length UB bits |
| 57 | LDPS,UB | BITDISP<br>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<br>BITDISP<br>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<br>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 |
| 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<br>*<br>*<br>6F | unused opcodes | | |
| 70<br>* | 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 |
| 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

# MPP DX/SC DISKETTE I/O UTILITY PROGRAM

## K.1 INTRODUCTION

This utility program is being provided so that a DX10 user may be able to read and write files to a SC formatted diskette. This format is used by the Rx File Manager as well as the AMPLUS Software Development System. Only single-density, single-sided diskettes are supported.

The utility program is designed to process one diskette per invocation. Therefore all commands are performed on one diskette. The utility program tries to optimize access to the diskette until another directory is to be processed. Also the system level directory is not written out until the end of the session.

## K.2 DXSC PROC

The DXSC proc invokes the utility program. The SCI prompt is shown below:

```
DXSC DISKETTE I/O UTILITY PROGRAM
        COMMAND FILE: input command file
        LISTING FILE: listing file
       DISKETTE NAME: diskette drive name
```

The COMMAND FILE parameter specifies the file from which commands are read. The command syntax will be described below. The default for this file is ME which allows commands to be interactively entered at the VDT. The LISTING FILE parameter specifies the file to which a log of the session is sent. All error messages are also sent to this file. The default for this file is ME which displays the messages on the VDT as the commands are processed. The DISKETTE NAME parameter specifies th name of the drive which contains the diskette to be processed. The default for this name is DK01 which is the standard name for the drive 1 diskette.

## K.3 COMMANDS

The commands to the utility program are described in this section. Each command must be entered one per line without any embedded blanks. A command is identified by a command character which must be entered in column one. If the command requires parameters, they must be entered on the same line with commas separating them.

All names are assumed to be 8 characters or less. File names may either be a single name or a directory name and member name with a period separating the, such as "directory.member". DX10 file names may be any length as long as they fit on the command line. One level of synonym mapping is performed on DX10 file names.

All numbers must be integers with an optional leading "#" symbol indicating a hexadecimal number.

Each command will be described by indicating its parameters and describing any output messages or displays generated by the command.

## K.3.1 Display Menu

This command displays a menu of commands which are supported by the diskette utility program. This menu is displayed each time a syntax error is detected in a command or when an unrecognized command is found. The menu is shown below.

```
A(du dump),number
C(reate dir),name,entries
D(elete file),name
I(nit disk),volume name,entries,volume id
M(ap disk)
P(atch adu),number,disp,list
Q(uit)
R(ead file),diskette name,DX pathname
V(erify file),diskette name,DX pathname
W(rite file),diskette name,DX pathname
```

### K.3.2 Display Location Unit - A

This command allows one allocation unit (AU) to be displayed on the listing file. The format of the command is as follows:

```
        A,number
```

The unit number must be given and must range from 0 to 2001. An example of the display is shown below.

```
- DUMP OF AU - 0 >0000
(00) 0000 10FF 4D50 4958 2020 2020 07D2 0000 (....MPIX        ....)
(10) 464D 0015 0004 0080 0000 0000 0000 0001 (FM..................)
(20) 0080 0001 0000 0054 0003 E8FF 0001 004D (.........T........M)
(30) 0000 554E 5553 4544 0000 0000 0000 0000 (..UNUSED...........)
(40) 0000 0000 0000 0000 0000 0000 0000 0000 (....................)
(50) 0000 0000 0000 0020 494E 5445 5250 5245 (...........INTERPRE)
(60) 5449 5645 2052 554E 2D54 494D 4520 5355 (  TIVE RUN-TIME SU )
(70) 5050 4F52 5420 2020 2020 2020 2020 2020 (PPORT               )
```

### K.3.3 Create Directory - C

This command creates a directory. The format of the command is as follows:

```
     C,name,entries
```

The name of the directory must be given as well as the maximum number of directory entries. The following message is displayed on the listing file to indicate that the directory has been created.

```
     CREATE DIRECTORY "name" ENTRIES = number
```

### K.3.4 Delete File/Directory - D

This command deletes a file or directory. The format of the command is as follows:

```
     D,file name
```

The name of the file or directory must be given. If a directory name is given, the directory and all of its files will be deleted. The following message is displayed on the listing file to indicate that the file or directory has been deleted.

```
     DELETE FILE "file name"
```

K.3.5 Initialize Volume - I This command initializes the volume information for a diskette. If the diskette has not been initialized before, it is formatted and must be initialized before any data can be written to it. The format of the command is as follows:

```
     I,name,entries,volume id
```

The name of the volume must be given first, followed by the number of directory entries for the volume level directory, and finally the volume identification must be given. The volume identification may be up to 40 characters long. The following message is displayed on te listing file to indicate that the volume has been initialized.

```
     INITIALIZE VOLUME "name" ENTRIES = number
```

This command may also be used to reinitialize an existing volume by simply using the command "I" without any additional parameters. This will delete all files and directories from the diskette, thus producing a clean volume. The following message is displayed on the listing file to indicate that the volume has been reinitialized.

```
     REINITIALIZE VOLUME "name"
```

## K.3.6 Map Diskette - M

This command produces a display on the listing file which describes the contents of the diskette. An example of the display is shown below.

```
      MAP OF - MPIX          ID = INTERPRETIVE RUN-TIME SUPPORT
        TOTAL ADUS = 2002     FREE ADUS = 355

        NAME       TYPE   PROT   RECL  ADU  LEN      DATE
       ******    MAX ENTRIES = 25 CUR ENTRIES   =    14
        HELP     BS        YES    80   25     2     8/22/80
        COLLECT  REL       YES   256   27   136     8/22/80
        COPY     REL       YES   256   95    48     8/22/80
        KERNEL   BS        YES    80  119     3     8/22/80
        MPIX     BS        YES    80  122     4     8/22/80
        CONFIG   BS        YES    80  306    27     8/22/80
        USERINIT BS        YES    80  333    70     8/22/80
        MATH$    DIR       YES    84 1375     6     8/22/80
          *****  MAX ENTRIES = 5 CUR ENTRIES    =     4
         PCODE   NBS       YES   128 1381    16     8/22/80
         PROCFIL BS        YES    80 1397     9     8/22/80
         SEGMENT BS        YES    80 1406    23     8/22/80
         XREF    BS        YES    80 1429     5     8/22/80
          *****
       ******
```

The volume name and volume identification are given on the first line. The total number of allocation units and number of free allocation units are given next. Then for each file or directory one line of information is given. The name of the file or directory is given in the first column. The type of file is in the second column, BS indicates a blank suppressed sequential file, NBS indicates a non-blank suppressed sequential file, REL indicates a relative record file, and DIR indicates a directory. The next column indicates if te file is protected. The fourth column gives the logical record length in bytes of the file. The next column gives the allocation unit where the file begins. The sixth column gives the length in allocation units of the file. Finally the last column gives the date of the last update made to the file. For directories for the maximum number of entries and current number of entries is also given.

## K.3.7 Patch Allocation Unit - P

This command is used to patch one or more words in an allocation unit. It should be used with extreme caution because there is no verification performed. The format of the command is as follows:

```
P,number,disp,list
```

The allocation unit number is given first and must range from 0 to 2001. The byte displacement of the first word to patch is given next. The displacement must be greater than or equal to 0 and less than 128. A dump of the patched allocation unit is displayed on the listing file.

## K.3.8 Read File - R

This command reads a file from the diskette to a DX10 file. The format of the command is as follows:

```
R,file name,source pathname
```

The diskette file name is given first followed by the DX10 file pathname. The following message is displayed on the listing file to indicate that the file is being read.

```
READ FILE "file name"
```

## K.3.9 Verify File - V

This command reads a file from the diskette and verifies it with a DX10 file. The format of the command is as follows:

```
V,file name,source pathname
```

The diskette file name is given first followed the DX10 file pathname. The following message is displayed on the listing file to indicate that the file is being verified.

```
     VERIFY FILE "file name"
```

## K.3.10 Write File - W

This command writes a file to the diskette from a DX10 file. The format of the command is as folows:

```
      W,file name,source pathname
```

The diskette file name is given first followed by the DX10 file pathname. The following message is displayed on the listing file to indicate that the file is being read.

```
      WRITE FILE "file name"
```

## K.3.11 Quit Session - Q

This command terminates the utility program session.

## K.4 ERROR MESSAGES

Each error message generated by the utility program is described in this section.

* * CANNOT ALLOCATE AU * - The diskette is too full to create the directory or the file being written is too large to fit on the diskette.

* * DIRECTORY FULL * - The directory does not have enough room for another entry.

* * DIRECTORY NAME ALREADY EXISTS - The directory name given already exists.

* * DISK I/O ERROR - OP = opcode - STATUS = error - An error was found while performing I/O to the diskette. The "opcode" indicates th I/O operation being performed and the "error" indicates teh error status returned by the service call.

* * DISKETTE NOT INITIALIZED - COMMAND IGNORED * - The diskette must be initialized before any other command processing is allowed.

* * END OF FILE ON HOST FILE - An end of file was found on the DX10 host file before the end of the diskette file was found.

* * ERROR IN COMMAND SYNTAX - The previous command had a syntax error in it.

* * FILE IS A DIRECTORY * - The file being read or verified is a

directory but not a file.

* FILE NOT FOUND * - The file name given is not present on the diskette.

* FILES DO NOT VERIFY * - The diskette file does not verify with the DX10 host file.

* HOST FILE I/O ERROR - OP = opcode - STATUS = error - An error was found while processing a DX10 file. The "opcode" indicates the I/O operation being performed and the "error" indicates tH error status returned by the service call.

* INCOMPATIBLE FILE TYPES * - The diskette file being read or verified is not compatible with the DX10 hot file.

* INCONSISTENT AU NUMBER * - An allocation unit is being freed which ahs already been marked as being free. This error indicates that some information on the diskette may have been desroyed.

* INVALID AU NUMBER * - The allocation unit number must be greater than or equal to 0 and less than 2002.

* INVALID DISPLACEMENT * - The byte displacement given in the patch command is less than 0 or greater than 127.

* INVALID DISK NAME * - The disk name given in the SCI prompt is not a disk device name.

* INVALID HOST FILE TYPE * - The DX10 host file being written to the diskette is not a relative record or sequential file.

* NAME IS NOT A DIRECTORY * - The directory named is a file but not a directory.

* NOT A SINGLE DENSITY DISKETTE * - The disk name given in the SCI prompt is not a single density diskette device name.

* TOO MANY ENTRIES * - The number of directory entries is limited to 400.

* TOO MANY SECONDARY EXTENTS - The diskette is too fragmented.

# APPENDIX L

# RTS CLOCK INTERRUPT HANDLER

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;
```

AD 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.

# APPENDIX M

# ASSEMBLY LANGUAGE INTERFACE: MPX

## M.1 GENERAL

When writing assembly language programs to be used with MPX, certain conventions must be followed to allow the assembly language routines to interface with MPX and the parts of the application written in Microprocessor Pascal. These conventions apply to the way in which the assembly language code is structured, how routines are called, and which registers may be used. This section details these conventions.

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

Suggestion: To become familiar with the required prologue and epilogue of your source modules, try the following: write a Pascal routine with the appropriate calling sequence; compile it; run it through CODEGEN; then reverse assemble it using RASS.

## M.2 LINKAGE CONVENTIONS

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

### M.2.1 Standard Procedure/Function Linkage

The standard procedure/function linkage supports parameter passing, local storage, reentrancy, and recursion. It acheives these by using the stack data structure illustrated in Figure M-1. In this stack, stack frames grow from the bottom toward high memory while 9900 workspaces grow from the top toward low memory. The stack region is allocated when the process is created. A stack overflow error occurs when there is not enough stack for another procedure call (the stackframes and workspaces overlap). The calling routine´s stack pointer (R10) and local frame pointer (R9) are shown on the left. These pointers, other system pointers, and the routine´s general registers are contained in the workspace pointed to by the workspace pointer shown on the left. The called routine gets a new workspace allocated for it which is pointed to by the workspace pointer shown on the right. Registers R9 and R10 in this workspace point to its local frame and stack shown on the right. This type of routine nesting repeats for as many routines as are called. As routines return to their caller, their stack frame and workspace are returned to the unused portion of the stack.

```
          Calling Routine                      Called Routine
                          +----------------+<--process stack base
                          (                )
                          (    previous    )
                          (     stack      )
                          (     frames     )
                          (                )
                 R9(LF)-->(----------------)
                          (                )
                          (    parent      )
                          (    stack       )
                          (    frame       )
                          (                )
                R10(SP)-->(----------------)<--R9(LF)
                          (    passed      )
                          (  parameters    )
                          (                )
                          (- - - - - - - - )
                          (    local       )
                          (   storage      )
                          (----------------)<--R10(SP)
                          (                )
                          (                )
                          (    unused      )
                          (    stack       )
                          (                )
                          (                )
                          (----------------)<--Workspace Pointer
                          (                )
                          (   workspace    )
                          (                )
     Workspace Pointer--> (----------------)
                          (    parent      )
                          (   workspace    )
                          (                )
                          (----------------)
                          (                )
                          (    previous    )
                          (   workspaces   )
                          (                )
                          (                )
                          +----------------+<--process stack limit
```

FIGURE M-1. STANDARD STACK NESTING

The stack frame for a standard procedure/function is determined by the routine prologue described in Sections M.3.1 and M.3.2. Basically, the routine prologue specifies how many parameters the called routine expects the calling routine to have pushed and how much local storage the called routine needs. Both the parameters and local storage are referenced using R9(LF) as the base register.

The standard procedure/function has three data areas that it may use to accomplish its purpose. It may use any of the general registers (described in Subsection M.4), local storage, and the stack. The general registers should be used for frequently accessed data or if only a few words of storage are needed. If the general registers do not provide enough data space, then either local storage or the stack must be used. Local storage is an area reserved in the stack immediately above the passed parameters at the time the procedure/function is called. This storage remains through any calls this routine makes until the time that it returns. Therefore, this space should be used for data which must remain during nested routine calls. The stack can be used as scratchpad storage between nested calls as long as the stack pointer is returned to its proper value before any routines are called. This use of the stack for scratchpad data reduces the stack requirements by reusing the same memory locations for data and passed parameters.

When a routine is called, there must be enough unused stack to allow for the standard linkage memory requirements. These requirements include the new workspace, the passed parameters, and any local storage. When enough stack does not exist, a stack overflow error occurs.

To make use of the standard procedure/function linkage, the routine must be called in the proper manner. An example of a call using the standard routine linkage is as follows:

```
        MOV  @PARM0,*R10+      PUSH PARAMETER 0
        MOV  @PARM1,*R10+      PUSH PARAMETER 1
              .
              .
        MOV  @PARMn,*R10+      PUSH PARAMETER n
        DATA CALL$             CALL ROUTINE
        DATA routine name
```

This code pushes ´n´ parameters and then calls the routine. All linkage functions are acheived by the standard MPX routine CALL$$. Errors will result if the actual number of parameters passed is different than the number of parameters expected by the called routine. The called routine references the passed parameters at displacements 0 to 2*n off of R9 as follows:

```
        MOV  *R9,@PARM0        GET PARM0
        MOV  @2(R9),@PARM1     GET PARM1
              .
              .
        MOV  @2*n(R9),@PARMn   GET PARMn
```

f any local storage is specified, this storage starts at displacement (2*n)+2 off of R9 and extends for as many bytes as specified.

When the called routine has completed, it returns via a branch or branch and link to the appropriate procedure or function exit routine. This routine deallocates the stack regions allocated to the called routine (parameters, local storage, and workspace) and resumes execution of the calling routine.

## M.2.2 Optimized Linkage

An alternative to the standard procedure/function linkage is the optimized linkage mechanism. This linkage executes much faster but does not perform many of the functions of the standard linkage. The optimized linkage allocates a new workspace for the called routine, but it does not initialize any registers (i.e., SP or LF) in this new workspace. The optimized linkage does not allocate any local storage for the called process. Any routine which was called with the ptimized linkage cannot call other routines.

Figure M-2 illustrates the stack after an optimized procedure/function linkage. The calling routine´s workspace pointer and registers are shown on the left, while the called routine´s workspace pointer is shown on the right. Register 13 in the new routine´s workspace points to the previous workspace. This is the link through which passed parameters are accessed.

```
     Calling Routine                     Called Routine

                     +-----------------+<--process stack base
                     (                 )
                     (     previous    )
                     (      stack      )
                     (      frames     )
                     (                 )
            R9(LF)-->(-----------------)
                     (                 )
                     (     parent      )
                     (     stack       )
                     (     frame       )
                     (                 )
           R10(SP)-->(-----------------)
                     (                 )
                     (                 )
                     (      unused     )
                     (      stack      )
                     (                 )
                     (                 )
                     (-----------------)<--Workspace Pointer
                     (                 )
                     (    workspace    )
                     (                 )
 Workspace Pointer-->(-----------------)<--R13
                     (     parent      )
                     (    workspace    )
                     (                 )
                     (-----------------)
                     (                 )
                     (     previous    )
                     (    workspaces   )
                     (                 )
                     (                 )
                     +-----------------+<--process stack limit
```

FIGURE M-2. Optimized Stack Nesting

The only data area directly available to an optimized procedure/function is the general registers in the new workspace. The optimized linkage does not allocate any local storage or update the stack pointer.

The calling sequence to an optimized procedure/function is identical to the calling sequence for a standard procedure/function. This similarity allows the calling routine to call other routines without knowing how the routines are coded (either standard or optimized). When the MPX linkage routine determines that the called routine is using the optimized linkage, it branches directly to the new routine's code.

The called routine must access any parameters using the stack pointer of the previous routine. The called routine is also responsible for updating the callers stack pointer before it returns. The called routine can reference the passed parameters as follows:

```
        MOV  @20(R13),R1       PUT CALLERS SP IN R1
        AI   R1,-(2*n)         DECREMENT CALLERS SP
        MOV  R1,@20(R13)       UPDATE CALLERS SP
        MOV  *R1+,@PARM0       GET PARM0
        MOV  *R1+,@PARM1       GET PARM1
                .
                .
        MOV  *R1+,@PARMn       GET PARMn
```

If a function result is to be returned, it may be done as follows:

```
        MOV  @RESULT,@20(R13) RETURN FUNCTION RESULT
```

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

### M.2.3 Process Linkage

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

When calling a process, the caller must have 458 bytes of available stack. This stack is necessary to support the procedure calls required to start the new process. The calling process must also have enough heap to supply a process record and the stack for the called process.

## M.3 SOURCE MODULE FORMAT

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

```
+----------------+
(     routine     )
(    descriptor   )
(-----------------)
(     routine     )
(     prologue    )
(-----------------)
(                 )
(     routine     )
(      body       )
(                 )
(                 )
(-----------------)
(     routine     )
(     epilogue    )
+----------------+
```

The routine desriptor contains constants needed by the linkage routine upon routine entry. The routine prologue contains any code necessary to start the routine. The routine body is the code which actually performs the purpose of the routine. The routine epilogue is the code required to exit the routine.

## M.3.1 Standard Procedure

A standard procedure requires a desriptor, body, and epilogue. The desriptor contains the following information:

```
+-----------------+<--procedure address
(      start      )       Offset to beginning of
(      offset     )       procedure body (in bytes)
(-----------------)
(       end       )       Offset to procedure
(      offset     )       epilogue (in bytes)
(-----------------)
(   local size    )       Size of local storage needed
(                 )       (in bytes)
(-----------------)
(   frame size    )       Size of total stack frame
(                 )       (in bytes)
(-----------------)
(      IR$L0      )       System defined constant
(                 )
+-----------------+
```

The start offset defines the offset to be added to the procedure address for the initial procedure program counter. The end offset defines the offset to be added to the procedure address in case the procedure is aborted. The local size specifies how many bytes should be allocated from the stack when the procedure is called for use as local storage. The frame size specifies the total stack frame size including passed parameters and local storage. Both the local size and frame size should be even values. IR$L0 is a system defined constant required by the linkage routine.

The procedure body consists of the assembly language staements required to acheive the procedure's desired effect. This will vary from procedure to procedure.

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

## 0.3.2 Standard Function

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

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

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

## 0.3.3 Process

The standard process format contains a descriptor, prologue, body, and epilogue. The desrciptor contains the following information:

```
+------------------+<--process address
(     start        )        Offset to beginning of
(     offset       )        process body (in bytes)
(------------------)
(      end         )        Offset to process
(     offset       )        epilogue (in bytes)
(------------------)
(       0          )        Zero constant
(                  )
(------------------)
(   parameter      )        Size of passed parameters
(     size         )        (in bytes)
(------------------)
(     IR$LO        )        System defined constant
(                  )
(------------------)
(   frame size     )        Size of stack frame needed for
(                  )        process body (in bytes)
(------------------)
(    lexical       )        Lexical nesting level
(     level        )
(------------------)
(    process       )        Process priority
(    priority      )
(------------------)
(    process       )        Size of stack region to be
(   stack size     )        allocated for process (in words)
(------------------)
(    process       )        Size of heap required for
(   heap size      )        process (in words)
+------------------+
```

The start offset, end offset, IR$LO, and frame size have the same meaning as for a standard procedure or function. The parameter size specifies how many bytes of parameters that the calling routine has pushed onto the stack. The lexical level specifies the number of levels that this process is nested within other processes. The lexical level of a system is 0, a process called from the system has a lexical level of 1, a process called from that process has a lexical level of 2, etc. The process priority specifies the relative urgency of this process compared to other processes. The lower the numerical prioity, the greater the urgency. The process stack size specifies how many words of stack will be required for the routines within the process. The process heap size specifies how many words of heap memory the process will need. A process requires a heap if it calls another process or allocates heap packets using the heap management routines.

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

```
MOV  @>A(R8),*R10+      PASS FRAME SIZE
MOV  @>C(R8),*R10+      PASS LEX LEVEL
MOV  @>E(R8),*R10+      PASS PRIORITY
MOV  @>10(R8),*R10+     PASS STACK SIZE
MOV  @>12(R8),*R10+     PASS HEAP SIZE
DATA CALL$              CALL START PROCESS
DATA S$PRCS
```

This code passes the necessary parameters from the process descriptor to the process start procedure S$PRCS.

The epilogue of a process permanently suspends execution of the process so that it does not compete with other processes for execution. The epilogue contains the following termination code:

```
MOV  @>C(R8),*R10+      PASS LEX LEVEL
DATA CALL$              CALL PROCESS TERMINATION ROUTINE
DATA E$PRCS
B    @EXIT$P            COMPLETE TERMINATION PROCESSING
```

This code passes the necessary parameter from the process descriptor to the process termination routine and then branches to the procedure exit routine to finish the termination processing.

## M.3.4 Optimized Procedure/Function

The optimized procedure/function format contains very little except the routine body. The format of an optimized routine is as follows:

```
+-----------------+<--descriptor
(        0        )
(-----------------)<--body
(                 )
(     routine     )
(       body      )
(                 )
(                 )
(-----------------)<--epilogue
(       RTWP      )
+-----------------+
```

The zero constant indicates to the linkage routine CALL$$ that this routine is in an optimized format. The return with workspace pointer (RTWP) instruction returns to the calling routine when this routine is finished.

## M.4 REGISTER USAGE

MPX uses certain registers within procedure, function, and process workspaces to maintain system level pointers. These registers must not be changed by the application software or erroneous results may occur. The following registers may not be changed at any time:

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

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

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

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

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

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

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

The remaining registers R0, R1, R2, R3, R4, R5, R6, R11, R12, and R15 may be used by the application software. Of these, R12 is reserved as the CRU base if any CRU operations are to be performed. R11 is the subroutine link register and may be used as such for subroutine linkage outside of the MPX environment.

## M.5 EXAMPLE PROGRAM

The MPX demonstration program is included at the end of this section as an example of an an ssembly language program interfacing with the MPX environment. The purpose of the application is to blink a light emitting diode on and off continually.

The demonstration program contains four routines. Of these, three are processes and one is a procedure. The system process, SYSTM$, calls the level one process, PRCS1. Because PRCS1 is of lesser urgency than SYSTM$, it does not gain control of the processor at this time. SYSTM$ continues execution in its process termination code. Once SYSTM$ has been permanently suspended, PRCS1 gains control. It initializes a semaphore in its local storage and then calls the process, TIME, passing a time constant and the semaphore as parameters. Because TIME is of lesser urgency than PRCS1, it does not gain control of the processor at this time. PRCS1 continues execution by initializing its CRU base (R12) to zero. It then WAITs for the semaphore to be signalled. When PRCS1 becomes suspended due to its WAIT, TIME gains control. It calls the DELAY procedure passing the time constant which was passed to it. DELAY enters a loop which repeats as many times as the time constant and then returns. At this point, TIME SIGNALs the semaphore passed to it. This SIGNAL causes PRCS1 (which is more urgent than TIME) to be reactivated. PRCS1 turns the LED off via a Set Bit to Zero instruction and then WAITs on the semaphore again. This causes it to be suspended again and TIME resumes execution. TIME loops back to call DELAY again. Once DELAY returns, it again SIGNALs the semaphore. This SIGNAL reactivates PRCS1 which then turns the light back on via a Set Bit to One instruction. PRCS1 then loops back to WAIT to turn the light off in an infinite loop.

This demonstration program illustrates how a problem can be divided into simpler tasks which execute independently. In this example, the TIME process could be replaced by another process which used an interval timer connected to an interrupt to perform the delays. This major system modification would not effect the other processes within the system.

The following is the source to the demonstration program.

```
       TITL 'MPX DEMO PROGRAM'
       IDT  'DEMOPGM '
*************************************************************************
*                                                                       *
*      THIS SYSTEM CONTAINS THREE PROCESSES AND ONE PROCEDURE          *
*      WHICH BLINK AN L.E.D. ON AND OFF CONTINUALLY                     *
*                                                                       *
*      THIS PROGRAM IS DESIGNED TO RUN ON A TM990/101 CPU               *
*      BOARD WITH EXPANSION MEMORY.  IF IT IS DESIRED FOR THIS          *
*      PROGRAM TO EXECUTE ON A TM990/100M BOARD, THEN A LIGHT EMITTING  *
*      DIODE SHOULD BE CONNECTED TO THE CRU LINE AT BASE ZERO IN        *
*      SUCH A WAY THAT WHEN THE LINE IS LOW, THE LIGHT IS ON.           *
*                                                                       *
*************************************************************************
       DEF  SYSTM$                * MODULE DEFINITION
       REF  IR$L0                 * <-+
       REF  CALL$                 *   |
       REF  S$PRCS                *   |
       REF  E$PRCS                *   |   EXTERNAL REFERENCES
       REF  EXIT$P                *   |   TO MPX MODULES
       REF  WAIT                  *   |
       REF  INITSE                *   |
       REF  SIGNAL                * <-+
       PSEG
*************************************************************************
*
*      SYSTEM PROCESS
*
*************************************************************************
*      DESCRIPTOR
*****
SYSTM$ EQU  $
       DATA TOP1-SYSTM$          * OFFSET TO THE BEGINNING OF CODE 'BODY'
       DATA BOT1-SYSTM$          * OFFSET TO PROCESSS TERMINATION CODE
       DATA 0                    * ZERO CONSTANT
       DATA 0                    * PARAMETER SIZE = 0
       DATA IR$L0                * SYSTEM REQUIRED DATA WORD
       DATA 0                    * FRAME SIZE = 0
       DATA 0                    * LEXICAL LEVEL = 0
       DATA 1                    * PROCESS PRIORITY = 1
       DATA 400                  * STACKSIZE = 400
       DATA 0                    * HEAPSIZE = 0
*****                            *
*      PROLOGUE                  *
*****                            *
TOP1   MOV  @>000A(R8),*R10+     * FRAME SIZE
       MOV  @>000C(R8),*R10+     * LEXICAL LEVEL
       MOV  @>000E(R8),*R10+     * PROCESS PRIORITY
       MOV  @>0010(R8),*R10+     * STACKSIZE
       MOV  @>0012(R8),*R10+     * HEAPSIZE
       DATA CALL$                * START THIS PROCESS (WHICH IS THE SYSTEM)
       DATA S$PRCS               *
*****                            *
*      BODY                      *
*****                            *
       DATA CALL$                * BEGIN THE NEXT LEVEL PROCESS
       DATA PRCS1                *
*****                            *
*      EPILOGUE                  *
*****                            *
BOT1   MOV  @>000C(R8),*R10+     * PUT THE LEXICAL LEVEL ON THE STACK FOR
       DATA CALL$                * THE PROCESS TERMINATION CODE
       DATA E$PRCS               *
       B    @EXIT$P              *
```

```
***********************************************************************
*
*       PRCS1 PROCESS
*
***********************************************************************
*       DESCRIPTOR
*****
PRCS1   EQU  $
        DATA TOP2-PRCS1           * OFFSET TO BEGINNING OF CODE 'BODY'
        DATA BOT2-PRCS1           * OFFSET TO PROCESS TERMINATION CODE
        DATA 0                    * ZERO CONSTANT
        DATA 0                    * PARAMETER SIZE = 0
        DATA IR$L0                * SYSTEM REQUIRED DATA WORD
        DATA 2                    * FRAME SIZE = 2
        DATA 1                    * LEXICAL LEVEL = 1
        DATA 30                   * PROCESS PRIORITY = 30
        DATA 500                  * STACKSIZE = 500
        DATA 2000                 * HEAPSIZE = 2000
        DATA 32000                * TIMING CONSTANT = 32000
*****                             *
*       PROLOGUE                  *
*****                             *
TOP2    MOV  @>000A(R8),*R10+     * TOTAL STACKFRAME SIZE
        MOV  @>000C(R8),*R10+     * LEXICAL LEVEL
        MOV  @>000E(R8),*R10+     * PROCESS PRIORITY
        MOV  @>0010(R8),*R10+     * STACKSIZE
        MOV  @>0012(R8),*R10+     * HEAPSIZE
        DATA CALL$                * START THIS PROCESS.
        DATA S$PRCS               *
*****                             *
*       BODY                      *
*****                             *
        MOV  R9,*R10+             * PUT THE SEMAPHORE ADDRESS ON THE STACK
        CLR  *R10+                * SEMAPHORE INITIAL VALUE = 0
        DATA CALL$                * INITIALIZE THE SEMAPHORE.
        DATA INITSE               *
        MOV  @>14(R8),*R10+       * PASS TIME DELAY CONSTANT
        MOV  *R9,*R10+            * PASS SEMAPHORE
        DATA CALL$                * START PROCESS 'TIME'.
        DATA TIME                 *
        CLR  R12                  * INITIALIZE CRU BASE
LOOP1   MOV  *R9,*R10+            * PUT THE SEMAPHORE ADDRESS ON THE STACK.
        DATA CALL$                * WAIT FOR THE SEMAPHORE TO BE SIGNALED.
        DATA WAIT                 *
        SBZ  0                    * TURN LIGHT OFF
        MOV  *R9,*R10+            * PUT SEMAPHORE ADDRESS ON STACK
        DATA CALL$                * WAIT FOR SEMAPHORE TO BE SIGNALED
        DATA WAIT                 *
        SBO  0                    * TURN LIGHT ON
        JMP  LOOP1                * REPEAT FOREVER
*****                             *
*       EPILOGUE                  *
*****                             *
BOT2    MOV  @>C(R8),*R10+        * LEXICAL LEVEL IS NEEDED FOR PROCESS TERM
```

```
DATA CALL$              *
DATA E$PRCS             *
B    @EXIT$P            *
```

```
************************************************************************
*
*       TIME PROCESS
*
************************************************************************
*       DESCRIPTOR
*****
TIME    EQU  $                  *
        DATA TOP3-TIME          * OFFSET TO BEGINNING OF CODE 'BODY'
        DATA BOT3-TIME          * OFFSET TO PROCESS TERMINATION CODE
        DATA 0                  * ZERO CONSTANT
        DATA 4                  * PARAMETER SIZE = 4
        DATA IR$L0              * REQUIRED DATA WORD
        DATA 4                  * FRAME SIZE = 4
        DATA 2                  * LEXICAL LEVEL = 2
        DATA 40                 * PROCESS PRIORITY = 40
        DATA 50                 * STACKSIZE = 50
        DATA 0                  * HEAPSIZE = 0
*****                           *
*       PROLOGUE                *
*****                           *
TOP3    EQU  $                  *
        MOV  @>A(R8),*R10+      * TOTAL FRAME SIZE
        MOV  @>C(R8),*R10+      * LEXICAL LEVEL
        MOV  @>E(R8),*R10+      * PROCESS PRIORITY
        MOV  @>10(R8),*R10+     * STACKSIZE
        MOV  @>12(R8),*R10+     * HEAPSIZE
        DATA CALL$              *
        DATA S$PRCS             *
*****                           *
*       BODY                    *
*****                           *
LOOP2   MOV  *R9,*R10+          * PASS DELAY CONSTANT
        DATA CALL$              * CALL DELAY PROCEDURE
        DATA DELAY              *
        MOV  @2(R9),*R10+       * PASS SEMAPHORE
        DATA CALL$              * SIGNAL THE SEMAPHORE
        DATA SIGNAL             *
        JMP  LOOP2              * REPEAT FOREVER
*****                           *
*       EPILOGUE                *
*****                           *
BOT3    MOV  @>C(R8),*R10+      * PUT LEXICAL LEVEL ONTO THE STACK
        DATA CALL$              * PROCESS TERMINATION CODE
        DATA E$PRCS             *
        B    @EXIT$P            *
```

```
***********************************************************************
*
*       DELAY PROCEDURE
*
***********************************************************************
*       DESCRIPTOR
*****
DELAY   EQU  $                    *
        DATA TOP4-DELAY           * OFFSET TO THE BEGINNING OF CODE ´BODY´
        DATA BOT4-DELAY           * OFFSET TO THE PROCEDURE TERMINATION CODI
        DATA 0                    * LOCAL VARIABLE SIZE = 0
        DATA 2                    * FRAME SIZE = 2
        DATA IR$L0                * REQUIRED SYSTEM DATA WORD
*
*****                             * THIS PROCEDURE DELAYS
*       BODY                      * FOR THE AMOUNT OF TIME
*****                             * IT TAKES TO EXECUTE
TOP4    EQU  $                    * N LOOPS, IN WHICH
        CLR  R2                   * REGISTER R2 IS INCREMENTED
LOOP3   EQU  $                    * BY ONE AS A TIME WASTING
        C    R2,*R9               * OPERATION.  N IS THE
        JGT  BOT4                 * PARAMETER PASSED TO IT
        INC  R2                   * BY PROCESS "TIME".
        JMP  LOOP3                * N IS FOUND BY *R9
*****                             * BECAUSE R9 IS A POINTER
*       EPILOGUE                  * TO THE PROCEDURE´S LOCAL
*****                             * FRAME, WHERE PARAMETERS
BOT4    EQU  $                    * ARE PUT BY THE CALLER.
        B    @EXIT$P              *
        END
```

# INDEX