Advanced Computer Design

AOS Programmer's Manual

AOS PROGRAMMER'S MANUAL

VERSION 1.0

June 1982

Advanced Computer Design

Authors: Rich Gleaves
Barry Demchak

Part #:

TABLE OF CONTENTS

"*" indicates extension of a standard Pascal intrinsic.

I. INTRODUCTION

1.0 Scope of this Manual

This manual describes the areas in which the UCSD Pascal Advanced Operating System (AOS) version 1.0, running on the PDQ-3 Computer system, differs from standard Pascal. Users are assumed to possess a working knowledge of standard Pascal. The following topics are covered:

1) Extensions to standard Pascal.

2) Deviations from standard Pascal.

3) Operating system customization.

4) Implementation-dependent programming practices.

This manual does not describe standard Pascal; it is intended to be used in conjunction with the language description presented in the Pascal User Manual and Report (2nd edition) by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975). Though the User Manual and Report is suggested as an ultimate reference, many other fine books describing Pascal are available. The following books are recommended:

Beginner's Guide for the UCSD Pascal System
Kenneth L. Bowles
Byte Books (McGraw-Hill), Peterborough, New Hampshire, 1979.

UCSD Pascal Handbook
Randall Clark and Stephen Koehler
Prentice-Hall, 1981.

Programming in PASCAL
Peter Grogono
Addison-Wesley, 1978.

Other documents related to the PDQ-3 Computer System include:

PDQ-3 Hardware User's Manual - Describes the physical characteristics of the computer.

AOS System User's Manual - Describes the Advanced Operating System software (including the operation of the Pascal compiler).

AOS Library User's Manual - Describes the library modules available with the Advanced Operating System.

AOS Architecture Guide - Provides details of the system software to experienced programmers. (Available in the indeterminate future).

1.1 Overview

The UCSD Pascal language was designed for the development of interactive Pascal programs for microcomputers; however, it contains sufficient extensions to also serve as a systems implementation language. All software on the PDQ-3 is written in UCSD Pascal.

UCSD Pascal extensions include:

Concurrency - Concurrent processes, interrupt handling, and an asynchronous I/O system allow the development of real-time and multiuser applications.

Separate Compilation - The "unit" construct enables programs to be built from groups of separately compilable modules; in addition, users may create standard library modules containing commonly used routines.

Program Segmentation - Programs may be partitioned on a procedural basis into a number of disk-resident code segments. Thus, large programs can run in limited amounts of memory by controlling the loading and unloading of its segments.

Extended I/O - Pascal's standard file I/O intrinsics are modified to operate in an interactive, single-user environment. Extensions are provided for direct access to the file system, random-access disk files, and device I/O.

Strings - Strings are character sequences whose length may vary dynamically during program execution. A predefined string type is provided, along with a set of intrinsics for common string-processing operations.

Precision Arithmetic - The standard integer data type may be extended to allow arithmetic with "long" integers containing up to 36 digits.

This manual is organized into eight chapters. Introduction presents an overview of UCSD Pascal along with information needed to use the manual. Deviations describes the areas where UCSD Pascal conflicts with standard Pascal. Extensions describes the UCSD Pascal extensions. UCSD Intrinsics provides detailed descriptions of all the UCSD Pascal intrinsic routines; the intrinsics are listed in alphabetic order for easy referencing. Compile Options describes compiler directives which affect either the compiler's operation or the nature of the code produced. Operating System Customization describes the addition and modification of operating system features. Programming Practices provides common programming practices in UCSD Pascal. Appendices includes information on I/O device attributes, implementation size limits, differences from other UCSD Pascal release versions, and available library routines.

1.2 Notation and Terminology

This section describes the notation and terminology used in this manual to describe UCSD Pascal.

A variant of Backus-Naur form (BNF) is used as a notation for describing the form of language constructs. Meta-words are words which represent a class of words; they are delimited by angular brackets ("<" and ">"). Thus, the words "trout", "salmon", and "tuna" are acceptable substitutions for the meta-word "<fish>"; here is an expression describing the substitution:

```
<fish> ::= trout | salmon | tuna
```

The symbol "::=" indicates that the meta-word on the left-hand side may be substituted with an item from the right-hand side. The vertical bar "|" separates possible choices for substitution; the example above indicates that "trout", "salmon", or "tuna" may be substituted for <fish>.

An item enclosed in square brackets may be optionally substituted into a textual expression; for instance, "[micro]computer" represents the text strings "computer" and "microcomputer".

An item enclosed in curly brackets may be substituted zero or more times into a textual expression. The following expression represents responses to jokes possessing varying degrees of humor:

```
<joke response> ::= {ha}
```

In many instances, the notation described above is used informally to describe the form required by a language construct. Here are some typical examples:

```
START(<process statement> [,<pid> [,<stacksize> [,<priority>]]])

CONCAT(<string> {,<string>})
```

The syntax for Pascal's IF statement is:

```
IF <Boolean expression> THEN <statement> [ELSE <statement>]
```

Pascal reserved words and predeclared identifiers are printed in CAPITAL LETTERS, while standard terms are underlined.

The following terms are used in the descriptions of UCSD Pascal: file name, block, block number, unit, and unit number. File name refers to the system's file naming convention; file names are described in chapter 2 of the System User's Manual. Block denotes the basic unit of transfer for disk files; a block is defined as 512 bytes of data. Block is also defined in Pascal as the set of declarations and statements comprising a program or procedure. Unit refers either to a separately compilable module or an I/O unit (as described in the System User's Manual). Unit number applies only to I/O units.

II. DEVIATIONS FROM STANDARD PASCAL

This section describes the areas where UCSD Pascal deviates from standard Pascal. Language differences are considered deviations if they meet the following criteria:

A) The differences affect compilation or execution of programs written in standard Pascal. These deviations affect the transportability of standard Pascal programs onto the UCSD Pascal system; they are generally categorized as implementation restrictions.

B) The differences subtly alter the standard Pascal language definition. These deviations affect the transportability of seemingly standard Pascal programs written in UCSD Pascal; they are generally categorized as "features"!

Sections 2.5.4 (WRITE), 2.8 (program headings) and 2.9 (records) describe deviations belonging to both categories. Sections 2.0 (CASE statements), 2.2 (NIL), 2.4 (ODD and CHR), 2.12 (comments), and 2.13 (type compatibility) describe UCSD Pascal "features". The remaining sections describe implementation restrictions.

NOTE - This section describes language deviations only. Implementation-dependent limits are described in Appendix G.

2.0 CASE Statements

In standard Pascal, the result of a CASE statement is undefined if the case selector contains a value which is not matched by any case label listed in the statement.

In UCSD Pascal, CASE statements are defined to have no effect in this situation; case selection "falls through", and execution continues with the statement following the CASE statement.

Example of CASE statement:

```
program fallthrough;
var ch: char;
begin
  ch := 'b';
  case ch of
    'a': writeln('ch = "a"');
    'c': writeln('ch = "c"');
  end;
  writeln('No errors from case...');
end {fallthrough}.
```

2.1 GOTO Statements

In UCSD Pascal, the scope of labels accessible to GOTO statements is restricted to a single block; thus, out-of-block GOTO's are not allowed.

NOTE - A limited form of out-of-block GOTO is provided with the EXIT intrinsic (see section 3.11.7 for details).

Example of out-of-block GOTO:

```
program outside;
label 1;

  procedure jump;
  begin
    goto 1;
  end {jump};

begin
  jump;
1:
end {outside}.
```

2.2 NIL

Standard Pascal defines the symbol NIL as a reserved word. NIL is a predefined identifier in UCSD Pascal.

2.3 FORWARD

Standard Pascal defines the symbol FORWARD as a directive lacking any meaning outside of a procedure declaration. FORWARD is a reserved word in UCSD Pascal.

2.4 ODD, CHR, and NOT

Standard Pascal defines the standard functions ODD, CHR, and NOT to return a result whose ordinal value reflects the result type. Thus, ODD and NOT are defined to return a BOOLEAN result whose ordinal value is in the range 0..1, while CHR is defined to return a result of type CHAR whose ordinal value is in the range 0..255.

In UCSD Pascal, ODD and CHR perform the required type conversion, but the ordinal value of the result is equal to the ordinal value of the argument. For example, ORD(ODD(56)) returns 0 in standard Pascal, but returns 56 in UCSD Pascal. ODD is defined in this manner to allow logical operations on integer types (see section 7.3).

In UCSD Pascal, NOT returns the 16-bit one's complement of its operand rather than complementing only the low-order bit. For example, ORD (NOT FALSE) returns 1 in standard Pascal, but returns -1 in UCSD Pascal. Use of the Boolean Negation compile option (described in section 5.0.13) causes the NOT operator to behave as defined in standard Pascal.

WARNING - This behavior implies that variables of type BOOLEAN and CHAR may contain values outside of their defined ordinal ranges. BOOLEAN and CHAR comparisons do not work correctly when their arguments possess out-of-range ordinal values, as they are implemented with full-word comparison operators. In addition, array indexing using subscripts of types BOOLEAN and CHAR may not behave as expected. Note that conditional statements in UCSD Pascal ignore all but the low-order bit of a BOOLEAN result, and thus are unaffected by this feature.

ODD example:

```
program eccentric;
var I: integer;
begin
  I := 556;
  I := ord(odd(I) and odd(255));
  { The high byte of I has been masked off }
  { I now contains the integer value 44    }
  I := ord(not odd(i));
  { Taking the 1's complement yields -45   }
end {eccentric}.
```

2.5 I/O Intrinsics

Sections 2.5.0 through 2.5.2 describe deviations resulting from UCSD Pascal's file I/O environment (see section 3.3). Sections 2.5.3 and 2.5.4 describe deviations resulting from implementation restrictions.

2.5.0 INPUT

The predeclared file INPUT is defined as an interactive file in UCSD Pascal. Section 3.3.1 describes interactive files.

All data read from the INPUT file is echoed to the predeclared OUTPUT file.

2.5.1 RESET and REWRITE

The standard procedures RESET and REWRITE have been altered to provide direct access to the file system (see section 3.3.0 for details). UCSD Pascal does not allow internal files; all files must be mapped into external files. Internal files may be simulated with temporary external files.

Example of an internal file in standard Pascal:

```
procedure local;
var internal: file of integer;
begin
  rewrite(internal);
  ...
end {local};
```

In UCSD Pascal, EOF is set to false after a file is rewritten; standard Pascal defines EOF to be true after rewriting a file. The reason for this discrepancy is explained in section 2.5.2.

2.5.2 EOF

UCSD Pascal redefines the meaning of the standard function EOF for files that are open for writing. The standard procedure REWRITE initially sets EOF to false; EOF then serves as a physical end-of-file indicator. The standard procedure PUT sets EOF to false after every successfully written record. If PUT attempts to write a record past the end of the space allocated for the disk file (see section 2.1.4.4 of the System User's Manual), and the file space cannot be extended (e.g. because of the presence of another disk file on the volume), EOF becomes true.

2.5.3 READ and READLN

The standard procedure READ may not be applied to files other than

text files. READ and READLN do not accept elements of packed character arrays as arguments. READ and READLN are redefined when used with interactive files (see section 3.3.1 for details).

2.5.4 WRITE and WRITELN

The standard procedure WRITE may not be applied to files other than text files.

Standard Pascal defines an optional control parameter named fraction length for specifying the output format of values of type REAL. The fraction length parameter specifies the number of digits to follow the decimal point in a fixed point representation of the value. If the fraction length specifies more digits than can be represented as significant digits by the underlying floating point implementation, the standard directs the fraction to be padded out with the requisite number of 0's. UCSD Pascal pads out overly long fractional parts with blank characters in place of (nonsignificant) "0" digits.

NOTE - The REALCON library unit (described in the Library User's Manual) provides procedures that allow the translation of real values to the standard format.

2.6 Packed Variables

The standard procedures PACK and UNPACK are not implemented in UCSD Pascal. UCSD Pascal does perform packing of array and record types preceded by the reserved word PACKED (see section 7.0).

2.7 Procedural and Functional Parameters

Procedural and functional parameters are not implemented in UCSD Pascal.

2.8 Program Headings

Parameter lists associated with program headings are ignored in UCSD Pascal. The standard files INPUT and OUTPUT are predeclared and opened to the system console by the operating system. Programs gain access to external files with the intrinsics REWRITE, RESET, and CLOSE (see section 3.3.0 for details).

NOTE - The INPUT and OUTPUT files may be attached to input and output streams other than the system console by using the I/O redirection options described in the System User's Manual.

2.9 Records

UCSD Pascal does not allow records to be declared with empty field lists.

UCSD Pascal does not enforce variant part completeness in record declarations; thus, the case labels need not specify all possible values of the tag field. For example, the following record declaration is legal:

```
type
     devrec = record
                s: string;
                case integer of
                  1: (b: boolean);
                  2: (r: real);
              end;
```

2.10 Files

UCSD Pascal does not allow file variables to be declared as part of an array or record. UCSD Pascal does not allow dynamic allocation of file variables.

2.11 Reserved Words

A number of reserved words have been added to UCSD Pascal. As noted in section 2.3, FORWARD is a reserved word rather than a directive. The following identifiers are reserved words in UCSD Pascal:

```
forward
interface
implementation
process
segment
unit
uses
```

2.12 Comments

Standard Pascal defines the symbols "(*" and "*)" as alternative symbols for the comment delimiters "{" and "}" respectively. Thus, comments may begin with "{" and end with "*)", or begin with "(*" and end with "}". Additionally, comments may not be nested in standard Pascal.

UCSD Pascal treats "(*" and "*)" as separate comment delimiters from the pair "{" and "}". Thus, comments beginning with "{" must end with "}", and comments beginning with "(*" must end with "*)". As a result, comments may be nested by using one pair of delimiters to comment out source code containing one or more comments delimited by the alternative symbols.

Example of comments in UCSD Pascal:

```
program comments;
begin

  (*
  comment out following statements ...

  writeln('don''t writeln');   { a contradiction }

  don'twriteln('writeln');  { a syntactically incorrect }
                            {       contradiction       }

  ... end of comment
  *)

end {comments}.
```

2.13 Type Compatibility

In standard Pascal, the rules for type compatibility are referred to as name compatibility. In general, variables are name-compatible if one of the following conditions is true:

a) The variables are declared with the same type identifier (e.g. var v1: stuff; v2: stuff;).

b) The variables are in the same identifier list of a single variable declaration (e.g. var v1, v2: array[char] of integer;).

In UCSD Pascal, the rules for type compatibility are referred to as structure compatibility. Variables are structure-compatible if the data structures implementing their respective type are structurally equivalent:

Simple types must share the same base type (note that subrange types are compatible with their base types).

Sets must have structure-compatible base types.

Arrays must have structure-compatible base types and index types along with identical array bounds.

Records must have structure-compatible fields declared in the same order.

Example of structure-compatible simple types:

```
type
      length = real;
      weight = real;
```

Example of structure-compatible arrays:

```
type
      t1 = 1..10;
      t2 = 1..10;

      x = array [t1] of integer;
      y = array [t2] of 0..52;
```

Example of structure-compatible records:

```
type
     polar = record
               radius, angle: real;
               int: integer;
             end;

     reall = record
               x: real;
               y: real;
               z: 0...2047;
             end;

     duple = record
               s,t: real;
               u:  10..15;
             end;
```

NOTE - Structure compatibility implies name compatibility; however, the converse is not true. Thus, UCSD Pascal programs utilizing structural type compatibility will not compile on standard Pascal compilers which enforce name compatibility.

WARNING - Structural equivalence of records can lead to somewhat strange notions of type compatibility. For instance, assume that the value 1.0 is assigned to the field x in a record of type reall. If the record is assigned to a record of type duple, the value 1.0 is contained in the t field of duple instead of the s field! This is a result of the compiler's scheme for allocating storage space for record fields (see section 7.1.1 for details).

In conclusion, UCSD Pascal programmers are urged to adhere to the "standard" practice of refraining from the use of structural type compatibility.

III. EXTENSIONS TO STANDARD PASCAL

This chapter describes UCSD extensions to standard Pascal. The extensions may be divided into three classes with respect to syntax:

Reserved Words - A handful of reserved words have been added to support segment procedures, units, and processes. Reserved words are listed in Appendix E.

Predeclared Types and Routines - These extensions may be used in any UCSD Pascal program; unlike reserved word extensions, predeclared identifiers may be redefined in the program. Two examples of predeclared types are STRING and SEMAPHORE. Predeclared procedures and functions are usually called intrinsics; they comprise the majority of language extensions in UCSD Pascal. See chapter 4 for detailed descriptions of the UCSD intrinsics.

Syntax Extensions - Standard Pascal syntax has been modified to accommodate some extensions. The SCAN intrinsic requires a parameter known as a "partial Boolean expression". A partial Boolean expression consists of an "=" or "<>" operator followed by a character expression (e.g. = 's' is a valid partial Boolean expression). The declaration of a block file appears as a file type declaration lacking a base type specification (e.g. TYPE BLOCKFILE = FILE; is a valid type declaration). Type declarations for strings and extended precision integers contain subtype specifications which define the type's size attribute (e.g. TYPE LONGINT = INTEGER[20];). Variable addresses are obtained in the PMACHINE intrinsic by preceding a variable reference with the "up-arrow" symbol (e.g. ^PERSON.NAME[I]). Finally, many intrinsics accept optional parameters or parameter sequences.

NOTE - All extensions described in this chapter are recognized by the compiler and are hence part of the UCSD PASCAL base language. Another class of extensions is available through the use of the library modules described in the Library User's Manual. Routines that allow program chaining, extended directory management, screen control, and other system-oriented functions are documented there, but are listed in Appendix K for convenience.

Sections 3.0 through 3.2 describe the major extensions to standard Pascal: concurrency, program segmentation, and separate compilation. Sections 3.3 through 3.6 describe other commonly used extensions: files, strings, dynamic variable management, and precision arithmetic. Sections 3.7 through 3.10 present low-level extensions which possess minimal type-checking and are intended primarily for systems use; these should be used only when necessary. Section 3.11 describes the remaining extensions.

3.0 Concurrency

Concurrency is defined as the simultaneous execution of a number of activities. Most computer systems simulate concurrency by implementing "virtual machines" on the physical machine; a virtual machine is assigned its own processor and memory. Each of the concurrent activities executes on its own virtual machine, leaving the physical machine responsible for simulating concurrent execution of the virtual machines. Virtual machines are generally referred to as tasks.

Concurrency in UCSD Pascal is restricted to concurrent execution of routines declared in a single program. A program may initiate any number of tasks, but the tasks must finish executing before the program is allowed to terminate.

Because the PDQ-3 is a single processor system, concurrency is simulated by sharing the processor among tasks. Processor sharing is accomplished by allocating each task a period of time during which it can execute, and then switching the processor to another task; the latter action is known as a task switch. The task executing on the processor is called the current task, while tasks waiting for processor time are called ready-to-run tasks. Execution states (i.e. processor register values) describing ready-to-run tasks are stored in a system structure known as the ready queue.

Semaphores are special variables used for task synchronization. Semaphores are used both in preventing tasks from executing until an event occurs, and in signalling occurrences of events. Tasks waiting for an event to occur are called suspended tasks. Execution states describing suspended tasks are stored in a semaphore's wait queue.

This section describes concurrency in UCSD Pascal. Tasks are described in section 3.0.0. Semaphores and applications of task synchronization are described in section 3.0.1. Section 3.0.2 describes interrupt handling, in which semaphores enable tasks to respond to processor interrupts. Section 3.0.3 describes time slicing, which allows simulation of true concurrent processing on the PDQ-3's single processor.

NOTE - See sections 7.6 and 7.7 for applications of concurrency to the development of device drivers and multiterminal programs.

NOTE - See the Architecture Guide for a description of the PDQ-3's concurrency implementation.

3.0.0 Tasks

A task is defined by four attributes: process, task identifier, stack size, and priority.

The primary attribute of a task is the code it executes; tasks may only execute routines declared as processes. Processes are described in section 3.0.0.0. Each task is assigned a unique identifying value when it is created; this value may be retained in a task identifier variable to distinguish the started task from other tasks in the system. Task identifiers are described in section 3.0.0.1. The amount of memory allocated for a new task is determined by the task's stack size. Task stacks are described in section 3.0.0.2. The last (but not least) attribute is priority; a task's priority value determines its ability to obtain processor time given the existence of other tasks. Task priorities are described in section 3.0.0.3.

The system assigns task attributes when a task is initiated. Tasks are created with the UCSD intrinsic START, which has the following form:

```
START ( <process call> [, <processid variable>
                          [, <stacksize expression>
                            [, <priority expression> ]]] );
```

The main parameter to START is a process call; it resembles a procedure call, and may contain parameters passed to the task (e.g. START(Zip) or START(Laughter(30))). Note that starting a single process several times in a program creates a number of tasks executing the same copy of a process's code. The remaining parameters are optional. Task identifiers must be declared as variables of type PROCESSID. The stack size parameter consists of an integer-valued expression, and represents the number of words allocated for a stack space; the default stack size is 200 words. The priority parameter consists of an integer-valued expression; if it is not in the range 0..255, an execution error occurs. The default priority is 128.

Tasks terminate execution when they reach the end of their process code; however, their stack spaces occupy memory until the parent program finishes execution. The system prevents a program from terminating until all of its tasks have terminated.

NOTE - The description of tasks presented here is sufficient for describing the execution of programs containing processes. From the system's point of view, the entire system (of which a user program is merely a part) is called the main task; the other tasks (including system device drivers and user-defined processes) are known as subsidiary tasks. From the processor's point of view, there is no distinction between the main task and subsidiary tasks; they are functionally equivalent. See the Architecture Guide for more information.

3.0.0.0 Processes

Processes are declared similarly to procedures; however, the reserved word PROCESS replaces the reserved word PROCEDURE. The syntax description presented below is derived from the formal syntax description for procedures in Appendix D of the Pascal User Manual and Report:

```
<process declaration> ::= <process heading> <block>

<process heading> ::=
    PROCESS <identifier> <formal parameter part> ;
```

RESTRICTIONS - Processes must be declared in the outer (global) block of a program; they may not be declared within a procedure or another process. START may only be called from the main task; thus, subsidiary tasks cannot create new tasks. Violating this restriction causes an execution error.

WARNING - Tasks are not allocated their own heap space; dynamic variables are always allocated on the system heap. For this reason, using DISPOSE to deallocate dynamic variables within a task is recommended rather than MARK and RELEASE, as MARK and RELEASE may inadvertently remove variables created by other tasks. Sections of the operating system dealing with global resource management (e.g. the file system and heap) are protected from task contention; nevertheless, processes using these resources should do so carefully. Section 3.0.0.2 describes other problems caused by interactions between tasks and the system heap.

NOTE - Variable parameters passed to a process may require an associated semaphore in order to ensure mutually exclusive access to the actual parameter (see section 3.0.1 for more information).

Examples of process declarations:

```
process Zip;
begin
  ...
end;

process BackgroundLaughter (Laughs: integer);
begin
  i := 0;
  if Laughs > 0 then
    repeat
      write('ha');
      i := (i + 1) mod Laughs;
      if i = 0 then writeln;
    until false;
end {BackgroundLaughter};
```

3.0.0.1 Task Identifiers

START assigns each task a unique value distinguishing it from other tasks. These values may be obtained by specifying a task identifier variable as the processid parameter to the START intrinsic; START assigns the value associated with the started task to the variable. Task identifier variables must be declared with the predefined type PROCESSID, and can be used in the same manner as pointer variables (i.e. the only valid operations are assignment and comparison with other task identifier variables).

NOTE - Other uses of task identifier variables are described in the Architecture Guide.

In the following example, two tasks are created with START; the variables PID1 and PID2 are assigned values identifying the tasks. Because these values are unique, this program writes "Truth" when executed:

```
Program a;

Var PID1,PID2: processid;

  process t;
  begin
    ...
  end;

begin
  start(t, PID1);
  start(t, PID2);
  if PID1 <> PID2 then writeln ('Truth');
end.
```

3.0.0.2 Task Stacks

Each task is allocated an area of memory in which it can execute; because UCSD Pascal programs execute on stack-oriented machines, the memory area is called a stack space. Stack space is used to store parameters and variables, procedure call information, and segment procedure code. When a task exhausts its stack space, a "stack overflow" occurs, and the system must be restarted.

NOTE - The main task's stack space is coincident with the system stack, and is limited in size only by the amount of system memory available. Stack spaces for subsidiary tasks are allocated on the system heap by the START intrinsic; hence, they are generally small compared to the main task's stack space. (Note that the main task's stack competes with the system heap for memory, while a subsidiary task's stack space is of fixed size, and is used only by the task stack.)

WARNING - Because task stacks are allocated on the system heap, tasks are susceptible to destruction from careless use of MARK and

RELEASE. Dynamic variables allocated before a started task should be deallocated using DISPOSE; never RELEASE the heap below a started task.

NOTE - Stack sizes must be sufficient for the basic needs of a process; the minimum size is: 32 words plus the number of words used by local variables and parameters. A procedure call uses a minimum of 4 words of space. If the task executes a segment process, stack space may be needed for the process code segment (see section 3.1 for more information). Attempt to avoid calling segments and/or procedures with large local data spaces, as they can quickly consume a task's stack space. (If this is unavoidable, the Libmap utility and compiled listings may be used to reveal the sizes of code and data segments in order to determine the amount of stack space required by a task (see the Architecture Guide for more information)).

Examples of stack size specification:

```
Program a;
Var PID: processid;
    I,J: integer;

  process p1;
  begin
    ...
  end;

begin
  I := 4;  J := 5;
  start(p1,PID);                { stack space = 200 }
  start(p1,PID,10);             { stack space = 10  }
  start(p1,PID,(I + J)*100);    { stack space = 900 }
end.
```

3.0.0.3 Priority

Each task is assigned a task priority value between 0 and 255. A task's priority determines its ability to obtain the processor when other tasks are ready to run. The processor's task scheduling policy is simple: no task may execute when a higher priority task is ready to run. The system enforces this policy by ordering all tasks in the ready queue by their priority, and by performing a task switch when the task at the head of the ready queue has higher priority than the current task.

NOTE - When a task is inserted into the ready queue, it is placed behind all other tasks having priorities greater than or equal to its own.

NOTE - The main task's priority is 128. Priorities above 191 are reserved for the operating system. Starting a process having priority higher than 128 immediately suspends the main task.

Examples of priority specification:

```
Program a;
Var PID: processid;
    I: integer;

  process p1;
  begin
    ...
  end;

begin
  I := 5;
  start(p1,PID,100);          { priority = 128 }
  start(p1,PID,100,90);       { priority = 90  }
  start(p1,PID,100,I*40);     { priority = 200 }
end.
```

3.0.1 Semaphores

Semaphores are variables declared with the predefined type SEMAPHORE. Semaphores are used solely for task synchronization; they are shared by tasks wishing to communicate with each other. Semaphores consist of two parts: a nonnegative integer counter and a queue for storing suspended tasks. Semaphores are never accessed directly; they are accessed with semaphore operators.

The principal semaphore operators are SEMINIT, WAIT, and SIGNAL.

SEMINIT initializes a semaphore variable by assigning it an initial count value and an empty wait queue.

WAIT checks the value of the semaphore count. If it is greater than zero, the count is decremented, and the current task continues to execute. Otherwise, the current task is stopped; it is placed in the semaphore's wait queue, and becomes a suspended task. The task at the head of the ready queue becomes the current task, and resumes its execution. Note that a task executing WAIT either continues as the current task or is stopped and becomes a suspended task.

SIGNAL examines the semaphore's wait queue. If it is empty, the semaphore's count is incremented. Otherwise, a suspended task is removed from the head of the wait queue and placed in the ready queue; it becomes a ready-to-run task. Note that if the signalled task has higher priority than the current task, a task switch occurs; thus, a task executing SIGNAL either continues as the current task or becomes a ready-to-run task.

NOTE - When a task is inserted into a semaphore's wait queue, it is placed behind all other tasks having priorities greater than or equal to its own.

Semaphores may be divided into two classes (with respect to usage): binary semaphores and counting semaphores. Binary semaphores have two states, as their counts only take on the values 0 and 1; they are used for mutual exclusion (section 3.0.1.0). Counting semaphores are so named because their count values can span the range of natural numbers; they are used for resource allocation (section 3.0.1.1).

WARNING - Semaphores must be initialized with SEMINIT before use; otherwise, system crashes may occur. Initializing a semaphore containing suspended tasks causes the suspended tasks to be lost forevermore. A semaphore's count value must not exceed 32767; otherwise, the count value wraps around to a negative value, leaving the semaphore in an undefined state.

Sections 3.0.1.0 and 3.0.1.1 present standard uses of semaphores in concurrent systems. Section 3.0.1.0 describes mutual exclusion, which is used to protect global variables and routines from contention between tasks. Section 3.0.1.1 describes task synchronization, in which semaphores are used to synchronize the execution of a group of tasks.

3.0.1.0 Mutual Exclusion

When processes share a resource (usually a variable or an I/O device), it is often necessary to protect the resource from being accessed by more than one task at a time; this form of resource protection is known as mutual exclusion. Mutual exclusion is ensured by placing all code which accesses the resource within a "critical section".

Critical sections are implemented (using a binary semaphore) by preceding the critical code with a call to WAIT and terminating the critical code with a call to SIGNAL. The binary semaphore is initialized to 1, indicating that the critical section is initially open; when a task executes a critical section (by passing the WAIT), the semaphore count is guaranteed to be zero, ensuring that other tasks may not enter the critical section until it becomes available (by passing the SIGNAL).

Example of mutually exclusive use of a console screen:

```
Program example;
Var Console: semaphore;

  procedure ConWrite(OutMsg: string);
  begin
    wait(Console);      { start critical section }
    writeln('I am ', OutMsg);
    signal(Console);    { end critical section }
  end;

  process MsgWriter(WhoIAm: integer; MyMsg: string);
  begin
    repeat
      ConWrite(MyMsg)
    until false;
  end;

begin
  seminit(Console,1);
  start(MsgWriter(1,'Shakespeare'));
  start(MsgWriter(2,'monkey'));
  start(MsgWriter(3,'typewriter'));
  .
  .
end {example}.
```

3.0.1.1 Synchronization

Semaphores may be used to synchronize the execution of a group of processes so that each process's execution depends on the actions of another process. Processes used in this fashion are known as cooperating processes. Cooperating processes are implemented by assigning a private semaphore to each process. A process considers its own private semaphore to represent an event which must occur before it can resume execution; therefore, the process waits on its private "event" semaphore. Processes wishing to indicate the occurrence of an event do so by signalling the corresponding private semaphore, thus activating the suspended process which owns the private semaphore.

Cooperating processes and private semaphores are illustrated in the example below, which demonstrates buffered data transmission (concurrency speeds up this activity by allowing simultaneous filling and sending of different data buffers). The resources in need of management are the N data buffers shared by the processes FillBufs and SendBufs. FillBufs finds an empty buffer and fills it with data. SendBufs finds a full buffer and dispatches it. The private semaphores are BufAvail and BufFull. BufAvail indicates that a buffer has been sent and is available for filling; its initial value indicates that all buffers are initially available for filling. BufFull indicates that a buffer is full and available for transmission; its initial value reflects the lack of full buffers at the outset.

```
program Buffers;
const N = { number of available buffers };
var   BufFull, BufAvail: semaphore;

  process FillBufs;
  begin
    repeat
      wait(BufAvail);
      { ...  Select an empty buffer and fill it ...  }
      signal(BufFull);
    until false;
  end;

  process SendBufs;
  begin
    repeat
      wait(BufFull);
      { ...  Select a full buffer and send it ...  }
      signal(BufAvail);
    until false;
  end;

begin
  seminit(BufFull,0);
  seminit(BufAvail,N);
  start(FillBufs);
  start(SendBufs);
end {Buffers}.
```

3.0.2 Interrupts

The UCSD intrinsic ATTACH allows processes to be used as interrupt-driven device drivers. ATTACH assigns a machine-dependent hardware interrupt vector to a semaphore; from then on, the semaphore is signalled whenever the processor receives an interrupt through the indicated interrupt vector.

NOTE - Interrupt vector assignments are described in the Hardware User's Manual. Also see section 7.6.3 for more information.

WARNING - ATTACH treats semaphore arguments as permanent variables; therefore, semaphores attached to interrupt vectors must be declared in the outer block of either the main program or the appropriate device process. The processor knows only of the memory address of an interrupt vector's attached semaphore, and continues to signal this address after every interrupt. It has no way of determining whether it is actually signalling a semaphore variable or merely damaging some unsuspecting code or data which happens to reside in memory previously occupied by an attached semaphore variable. To wit, indiscreet use of ATTACH may adversely affect the system.

WARNING - Because the PDQ-3 system currently lacks a method for de-attaching semaphores, the system must be rebooted after running a user program containing attached semaphores if the devices causing interrupts cannot be disabled.

An example of interrupt processing may be found in section 7.6.3.

3.0.3 Time Slicing

Time slicing refers to the allocation of processor time to each task in the ready queue. Time slicing on the PDQ-3 is a side-effect of interactions between the system clock handler process and the task scheduling mechanism.

The system clock interrupts the processor 60 times per second. The clock handler process has higher priority than user tasks; it continually waits for clock interrupts in order to update the system time. When the processor receives a clock interrupt, a task switch occurs that activates the clock handler process, causing the current user task to be inserted in the ready queue behind other tasks of equal priority. When the clock handler suspends itself, the processor selects the task at the front of the ready queue as the current task. Thus, the processor circulates between tasks of equal priority.

NOTE - The sy3.1's may perform time slicing only if the clock driver is installed as a system driver (see section 2.3.1 of the System User's Manual for details). In its absence, task circulation is performed only as a result of a task blockage due to either the execution of the WAIT intrinsic or an interrupt-driven I/O.

The following example demonstrates time slicing on the PDQ-3; when executed, the program prints final counts that are approximately equal, indicating that the tasks receive similar amounts of processor time.

```
program RaceCondition;
const limit = 10000;
var   car1, car2, car3: integer;
      CheckeredFlag: boolean;

  process racer(var counter: integer);
  begin
    counter := 0;
    repeat
      counter := counter + 1;
      if not CheckeredFlag then
        CheckeredFlag := counter >= limit;
    until CheckeredFlag;
    write(counter:6);
  end;

begin
  CheckeredFlag := false;
  start(racer(car1));
  start(racer(car2));
  start(racer(car3));
end.
```

3.1 Program Segmentation

Program segmentation refers to the division of program code into disk-resident code segments. A code segment is memory-resident while it is executed; the system loads it into memory when necessary, and releases it from memory when possible. Memory occupied by a code segment is freed for other uses when the segment is released, ensuring efficient use of memory; thus, segmented programs can avoid the memory constraints normally imposed on large programs.

Program segmentation in UCSD Pascal is achieved on a procedural basis through the use of segments. A procedure, function, or process is specified to reside in a separate code segment by preceding its declaration with the UCSD Pascal reserved word SEGMENT. The code segment contains the segment's code along with the code belonging to its (nonsegmented) local procedures. Note that specifying a routine as a segment does not change the meaning of the enclosing program.

When a segment is called, the system reads the corresponding code segment into memory and executes the call; the code segment remains in memory until the call is terminated. Recursive segment calls proceed directly without loading the code segment, as it is already memory-resident. Immediately before a segment terminates, the system determines if the current segment invocation is recursive. If so, the code segment remains in memory; otherwise, no other invocations exist, and the code segment is released from memory.

Programs should be segmented with an eye towards minimizing both the amount of code in memory and the frequency with which segments must be loaded from disk. Segmenting a frequently called routine causes the system to thrash, as the code must be read from disk on each call. A more suitable candidiate for segmentation is initialization code, which is usually executed only once at the beginning of a program. Segments must be independent of each other in order to reap the benefits of segmentation. For example, envision a large piece of code requiring division into three segments (named A, B, and C) in order to conserve memory. If the division results in a program where A calls B and B then calls C, segmentation is fruitless - all three code segments are memory-resident while C executes! A proper division results in three mutually independent segments which are called sequentially (e.g. "A; B; C;"). The maximum memory required by this division is the size of the largest segment rather than the sum of all three. Program segmentation is most effective when it influences the design of large programs (as opposed to "tuning" existing programs).

NOTE - Within the main program or any segment declaration, the code comprising local segments must appear before code belonging to the enclosing segment or program. As can be seen in the example, this does not prevent unsegmented procedures from containing local segments, but does affect the order in which local procedures are declared. As with unsegmented procedures, segments may be declared forward to resolve interprocedural references. Forward declaration of segmented and unsegmented procedures may occur in any order.

NOTE - A program may contain between 1 and 128 code segments; one code segment is reserved for the program itself, while the remaining code segments are available either for segments or units (section 3.2). In the absence of units, a program may contain up to 127 segments. The program segment and all resident unit segments are automatically loaded into memory at program invocation time.

WARNING - When a program calls a disk-resident segment, the disk volume containing the program's code file (and thus its code segments) must be online and mounted in the same drive as when the program was started; otherwise, the system crashes.

Example of segment declarations:

```
program main;

  procedure p1; forward;
  segment function p2: integer; forward;

  segment procedure p3;

    procedure p3p1;

      segment function p3p1p1:boolean;
      begin
        ...
      end {p3p1p1};

    begin
      ...
    end {p3p1};

  begin
    ...
  end {p3};

  segment function p2{: integer};
  begin
    ...
  end;

  procedure p1;
  begin
    ...
  end;

begin
  ...
end {main}.
```

3.1.0 Alternate Segment Management Strategies

A segment is normally memory-resident only while it is executed. The $R compile option and the UCSD intrinsics MEMLOCK and MEMSWAP allow alternate segment management stategies. The $R compile option causes a list of segments to be resident throughout the execution of a given procedure. See section 5.0.9 for more information. The MEMLOCK and MEMSWAP intrinsics provide runtime control over the loading and unloading of segments. Use of these intrinsics must be accompanied by the use of the $H compile option, described in section 5.0.6.

The MEMLOCK and MEMSWAP intrinsics accept a string value parameter containing a list of segments to be loaded or unloaded, respectively. A segment list contains segment and unit identifiers (section 3.2) declared in the program and its used units, or in the operating system. Identifiers are separated by commas; spaces and invalid identifiers in the segment list are ignored. The form for a MEMLOCK or a MEMSWAP call is:

```
<memlock-call> ::= MEMLOCK(<segment-list>)
<memswap-call> ::= MEMSWAP(<segment-list>)

<segment-list> ::= <segment-name> {,<segment-name>} | <empty>
```

The MEMLOCK intrinsic causes each code segment in the segment list to be loaded onto the system heap. Subsequent calls to such segments use the MEMLOCKed copy of the code rather than loading it from disk. MEMLOCKing an already MEMLOCKed segment has no effect.

The MEMSWAP intrinsic causes each code segment in the segment list to be removed from memory. The memory occupied by the code segment is subsequently available for reallocation by the NEW and MEMLOCK intrinsics. MEMSWAP operates only on code segments loaded as a result of a MEMLOCK call. It does not unload a code segment until there has been a matching MEMSWAP call for each MEMLOCK call on that segment and there are no active calls to that code segment.

NOTE - MEMLOCKing a segment does not unload copies of the segment loaded as a result of prior loads and calls. However, once the non-MEMLOCKed copies are unloaded, subsequent calls use the MEMLOCKed copy of the code. For example, if a currently executing segment (resident on the system stack) MEMLOCKs itself, two copies of the code segment will exist in memory until the active segment invocation is terminated. At that point, the original copy of the code segment is removed from memory. Subsequent calls to the segment use the MEMLOCKed copy.

NOTE - All MEMLOCKed segments are unloaded at program termination.

WARNING - Attempts to MEMLOCK a segment whose identifer is shared by more than one segment has unpredictable results.

NOTE - MEMSWAPing a segment to which there are active calls causes the code segment to be unloaded after the termination of the last

call. Unfortunately, if the last call is terminated by the EXIT intrinsic, the code segment is not actually unloaded until the next call to that segment. Calls made to segments between the MEMSWAP call and the termination of the last active call are processed as if the segment was not MEMLOCKed.

WARNING - Since MEMLOCKed segments reside in the heap, indiscreet use of MEMLOCK may render the heap incapable of containing large buffers. Such use may also cause a stack overflow if a MEMLOCKed code segment is situated in memory so that extension of the system stack is impossible.

Example of MEMLOCK and MEMSWAP use:

```
program mems;

  segment procedure segl;
  begin
    <...  segment code ...>
  end;

  segment procedure seg2;
  begin
    <...  segment code ...>
    memswap('segl');
  end;

begin
  memlock('segl, seg2');
  seg2;
  memswap('seg2');
end {main}.
```

3.1.1 Segments and Tasks

A few restrictions are imposed on the use of segments in conjunction with concurrent tasks. These restrictions are due to architectural limitations.

Processes may be declared as segments; however, they operate somewhat differently. If the process executed by a task is declared as a segment, the code segment containing the process is read onto the task's stack, and remains there while the task executes. Unfortunately, when the main program terminates, the system is unable to shut down segmented tasks in an orderly fashion, and so must be rebooted; therefore, segment processes should only be used in dedicated (i.e. nonterminating) programs.

A task calling a swappable segment receives a private copy of the code segment on its task stack; code for MEMLOCKed segments is shared among tasks.

NOTE - Due to architectural limitations, calls by tasks to segment procedures and functions declared in units installed in the

intrinsics, drivers, and system support libraries (described in the System User's Manual) must either be treated as critical sections, or the segments must be MEMLOCKed; otherwise, system crashes may occur. See section 3.0.1.0 for details on critical sections.

3.2 Separate Compilation

Separate compilation (also known as "external compilation" or "modular programming") allows programs to be created from individually compiled modules. Some advantages resulting from separate compilation are:

-- New modules can be written, compiled, and combined with existing modules to create new programs. The new modules themselves might later be used in other programs. Thus, a growing catalog of precompiled software tools may become available for use in general software development.

-- Large programs constructed from separate modules are easily modified; changes are isolated to individual modules, allowing fast and reliable program maintenance.

-- Programs can be developed that are larger than could otherwise be compiled in one piece on the system.

Separately compiled modules are built in UCSD Pascal using the UNIT construct. Unit declaration is described in section 3.2.0. Section 3.2.1 explains how units are referenced by host programs. Section 3.2.2 provides information on unit linkage. Section 3.2.3 provides information on the memory management of unit code segments.

NOTE - This section provides a program-level description of units. Section 7.9 describes the philosophy and pragmatics of unit construction and usage. The System User's Manual presents a system-level description of units and libraries.

3.2.0 Units

Units are collections of uses-, constant-, type-, variable-, procedure-, function-, and process- declarations usually oriented toward some application. These objects become available for use when the unit is referenced by a host (a program or another unit). Units consist of four parts: an interface section, an implementation section, an initialization section, and a termination section. Objects declared in a unit's interface section are public; they are accessible to both the unit and the host which uses the unit. Objects declared in the implementation section are private; they are accessible only within the unit's implementation section. The initialization section is a code sequence that usually initializes unit variables and is automatically executed once at program invocation time. It is executed before any code in the host which uses the unit. The termination section is a code sequence that is automatically executed once at program termination time and usually performs any "shut-down" operations required. It is executed after the termination code of the host which uses the unit.

An example of a unit declaration appears on the next page. Note that the interface section may contain only procedure and function

headings - routine bodies are not allowed. Procedure and function headings in the interface section are similar to forward declarations; when the corresponding routines are defined in the implementation section, the parameter list is omitted.

```
unit mnemenos;
interface

  type mnemone = (truth, beauty, wisdom, knowledge, etc);

  procedure relapse;
  { forget all items learned }

  procedure learn (newentry: mnemone);
  { learn a new item }

  function recall (look: mnemone): boolean;
  { has item been learned? }

implementation

  type listentry = record
                     data: mnemone;
                     next: entryptr;
                   end;
   var listhead: ^listentry;

   procedure relapse;
   begin listhead := nil end;

   procedure learn;
   var entry: ^listentry;
   begin
     new (entry);
     with entry^ do
       begin data := newentry; next := listhead; end;
     listhead := entry;
   end;

   function recall;
   var entry: ^listentry;
   begin
     recall := false; entry := listhead;
     while entry <> nil do
       if entry^.data <> look then
         entry := entry^.next
       else recall := true;
   end;

begin
  listhead := nil;       {initialization section}
***;
  relapse;               {termination section}
end {mnemenos}.
```

The syntax for unit definition is shown below (it is loosely based

on the Pascal syntax in Appendix D of the User Manual and Report).

```
<compilation unit>     ::= <program> | <library>

<program>              ::= <program heading>;
                           <inline unit part>
                           <uses part>
                           <block>.

<library>              ::= <unit definition>
                           {;<unit definition>}.

<inline unit part>     ::= {<unit definition>;}

<uses part>            ::= [USES <unit identifier>
                           {,<unit identifier>};]

<unit definition>      ::= UNIT <unit identifier>;
                           <interface part>
                           [<implementation part>]
                           [BEGIN
                            [<initialization section>]
                            [***;
                             [<termination section>]]]
                           END

<interface part>       ::= INTERFACE
                           <declarations>
                           <procedure and function headings>

<implementation part>  ::= IMPLEMENTATION
                           <declarations>
                           <procedure and function bodies>

<declarations>         ::= <uses part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
```

The ***; statement is valid only when used to separate initialization and termination sections. It may not be contained in any statement or procedure body.

Note that labels may not be declared globally in units. Neither GOTO nor EXIT statements may occur in either the unit initialization section or termination section. Segment declarations are allowed in both the interface and implementation sections; they follow the conventions described in section 3.1 for forward declarations and procedure body declarations.

Note also that a unit may consist solely of an interface section (and possibly an initialization and/or termination section); these are known as data units. A data unit consists of only uses-, constant-, type-, and variable declarations which are accessible to a host program.

Example of a data unit:

```
unit ComplexData;
interface
  type complex = record
                   realpart, imaginary : real;
                 end;
  var one, i : complex;
begin   {initialization section}
  one.realpart := 1;  one.imaginary := 0;
  i.realpart := 0;    i.imaginary := 1;
end.  {ComplexData}
```

The compiler accepts the following combinations of units and programs:

1) A program.

2) A unit (as in the previous example).

3) A group of units.

4) A program containing one or more inline units.

NOTE - An interface section may be contained in an include file if the keyword INTERFACE is also contained in the include file. However, interface sections may not contain include file directives.

3.2.1 Using Units

A unit may be used in a host by naming it in a USES statement. In programs, the USES statement must appear after the program heading. In units, the USES statement must appear at the beginning of either the interface section or the implementation section. Objects declared in the interface section of a used unit become globally declared objects within the host. Objects imported by using a unit in the implementation section remain private to the using unit.

Code segments for units are normally memory-resident for the life of the host program. Units may be designated swappable by using the $N compile option described in section 5.0.9. Such units are resident in memory only when needed (similar to segment procedures), unless operated upon by the $R compile option or the MEMLOCK intrinsic (see section 3.1.0).

NOTE - In situations where a used unit uses other units in its interface section, the host must name the nested units in its USES statement before naming the unit which uses them. For example, if unit A uses unit B in its interface section, then a host using unit A must contain the USES statement "USES B,A;".

WARNING - Because objects imported from used units have global scope within the host, naming conflicts may arise between globally declared program identifiers and identifiers imported from used units.

NOTE - One copy of a unit's public and private variables exists for all USES of a unit by a program and its used units.

In the following example, the program uses the mnemenos unit declared in a previous example; identifiers imported from the unit are underlined for emphasis. Note that the initialization section of the unit is executed before the program is executed.

```
program UnitDemo;
uses mnemenos;
type charset = set of char;
var  finished: boolean;

  function GetCommand(valid: charset): char;
  var ch: char
  begin
    repeat
      read(keyboard,ch);
    until ch in valid;
    GetCommand := ch;
    writeln(ch);
  end;

  function GetCategory(command: string): mnemone;
  begin
    write(command,': T(ruth B(eauty W(isdom K(nowledge E(tc');
    case GetCommand(['T','B','W','K','E']) of
      'T': GetCategory := truth;
      'B': GetCategory := beauty;
      'W': GetCategory := wisdom;
      'K': GetCategory := knowledge;
      'E': GetCategory := etc;
    end;
  end;

begin {UnitDemo}
  finished := false;
  repeat
    write('Education: L(earn R(ecall F(orget G(raduate');
    case GetCommand(['L','R','F','G']) of
      'L': learn(GetCategory('Learn'));
      'R': if recall(GetCategory('Recall')) then
             writeln('Remembered')
           else
             writeln('Forgotten');
      'F': relapse;
      'G': finished := true;
    end;
  until finished;
end.
```

3.2.2 Unit Linkage

Linkage to units is performed both at compile time and at runtime. At compile time, the compiler imports the identifiers contained in a used unit's interface section. At runtime, the operating system loads code segments of used units and resolves unit references. Linkage information is maintained with unit code in files known as libraries. The library system contains an intrinsics library, a system library, and the user library. It is described in section 2.2 of the System User's Manual.

In order to import the interface section of a unit used by a host, the compiler must first locate the unit in the library system. The compiler searches the intrinsics library first, then the current code file (for in-line units), the system library, and finally the user libraries. If the search is unsuccessful, a compiler syntax error is emitted.

NOTE - In cases where an in-line unit's name matches one already present in the intrinsics library, the copy supplied by the intrinsics library is used.

If a unit is modified, it should be recompiled and reinstalled into the library system. If a unit's interface section is modified, its version number (see section 5.0.10) should be changed before recompilation, and all hosts using the unit should be recompiled with the new version installed in the library system.

In order to execute a program using units, the operating system must locate each unit used by the program and its used units. This search is performed in the same manner as the compile time search. If a unit is not found, or it is found and its version number has changed since the host was originally compiled, the system emits an error message and aborts the execution of the program.

WARNING - Maintenance of version numbers is an important safegaurd against accidental incompatibilities introduced by the modification of units. Failure to maintain version control may yield dangerous and unpredictable results.

NOTE - Units are compiled into code segments in a manner similar to programs; the unit occupies one segment and each segment procedure or function occupies a segment. Programs and units may directly access up to 127 units and segments. A program and all of its used units cannot access more than a total of 128 units and segments, excluding any units and segments installed in the intrinsics library. Units installed in the intrinsics library do not count against the 128 segment limit.

3.3 Files

UCSD Pascal provides a number of extensions for file handling. The extensions include:

-- Direct access to the file system from programs.

-- Interactive file I/O on the system terminal.

-- Random-access disk files.

-- Block-oriented files for systems programming.

Section 3.3.0 introduces the UCSD intrinsic CLOSE and describes extensions made to the standard procedures RESET and REWRITE. Together, these allow programs to access the file system. Section 3.3.1 describes the predeclared file type INTERACTIVE; when applied to interactive files, the standard procedures READ and RESET are redefined to accommodate interactive I/O. Section 3.3.2 describes the predeclared file KEYBOARD, which reads characters from the standard input without echoing them on the standard output. Section 3.3.3 describes block files. Block files are accessed with the UCSD intrinsics BLOCKREAD and BLOCKWRITE; these read and write data in integral numbers of blocks. Block files allow efficient manipulation of large, arbitrarily structured files. Section 3.3.4 introduces the UCSD intrinsic SEEK, which is used to randomly access the contents of disk files.

3.3.0 File System Access

UCSD Pascal provides direct access to the file system; this allows programs to manipulate disk files and perform file operations on I/O devices. It is useful to make a distinction between file variables and external files - a file variable is declared in a program, while an external file is either a disk file or an I/O device. File system access is accomplished by connecting a program's file variable with an external file. A file is open if it has been connected, and closed if it either has not yet been connected or has been connected and subsequently disconnected. File I/O operations may only be performed on open files.

The file system is described in section 2.1 of the System User's Manual.

The file system is accessed with the intrinsics RESET, REWRITE, and CLOSE. RESET and REWRITE connect files, while CLOSE disconnects files.

REWRITE creates new files. Its form is:

<rewrite-call> ::= REWRITE(<fileid>,<filename>)

... where <fileid> is a file variable identifier and <filename> is a string containing a file name. REWRITE creates a new external file with the given file name and prepares the file variable for subsequent file operations. Note that using REWRITE with a single argument (as defined in standard Pascal) causes a syntax error in UCSD Pascal.

NOTE - As mentioned in the file system specification, files on a disk volume must have distinct file names; an existing file is automatically deleted if another file with the same name is entered in the disk directory. A disk file created by REWRITE is assigned temporary status; it becomes a permanent file only if it is closed and locked (see below for details). Thus, programs which generate temporary files need not worry about inadvertently deleting permanent disk files.

RESET opens existing files for subsequent file operations, or resets the file position of an open disk file to its beginning. The form for RESET is:

<reset-call> ::= RESET(<fileid>[,<filename>])

... where <fileid> is a file variable identifier and <filename> is a string containing a file name. Calling RESET with the second parameter present opens an existing external file named by <filename> and prepares the file variable for subsequent file operations. Note that RESET with a single parameter (i.e. the file identifier) works as defined in standard Pascal, but is applicable only to open disk files.

CLOSE disconnects files. The form for close is:

<close-call> ::= CLOSE(<fileid>[,<option>])

<option> ::= NORMAL | LOCK | PURGE | CRUNCH

The options determine the final state of a file. NORMAL (which is the default option) preserves permanent files which were RESET, but deletes temporary files created by REWRITE. LOCK preserves files as permanent disk files. Locking a temporary file may delete an existing permanent file if they share the same name. PURGE deletes files from the directory. CRUNCH is equivalent to LOCK, but causes the file window position to become the end of the file. (See section 4.4 for more information.)

NOTE - An implicit CLOSE(<file>, NORMAL) is performed on files which are not explicitly closed.

NOTE - The UCSD intrinsics OPENOLD and OPENNEW are synonymous with RESET and REWRITE respectively. Chapter 4 contains detailed descriptions of the intrinsics mentioned in this section. Section

2.1 of the System User's Manual describes the file system and file naming conventions.

Examples of file system access using RESET, REWRITE, and CLOSE:

```
program FileDemo;
var infile,outfile: text;
    ch: char;
begin

  { open up the disk file named "master" }
  reset(infile,'master.text');

  { copy to a disk file named "copy1" }
  rewrite(outfile,'copy1');
  while not eof(infile) do
    begin read(infile,ch); write(outfile,ch); end;
  close(outfile,lock);

  { rewind master file for second pass }
  reset(infile);

  { copy to a disk file named "copy2" }
  rewrite(outfile,'copy2');
  while not eof(infile) do
    begin read(infile,ch); write(outfile,ch); end;
  close(outfile,lock);

  { close down master file }
  close(infile,normal);
end.
```

3.3.1 Interactive Files

UCSD Pascal provides the predeclared file type INTERACTIVE in order to facilitate use of the system terminal as an input file. Interactive files are structurally equivalent to text files; the only difference between them is the manner in which the standard procedures RESET, READ, and READLN are defined to act.

To explain the need for interactive files, it is first necessary to examine the definitions of text file operations in standard Pascal. Let ch be a character variable, and f a file of type TEXT; the following rules then hold for RESET and READ:

1) RESET(f) is defined to perform an implicit GET(f)

2) READ(f,ch) is equivalent to ch := f^; GET(f)

Using these standard definitions, the following program attempts to create a simple console prompt by writing a prompt message to the console screen and accepting a response from the console keyboard:

```
program prompter;
var infile,outfile: text;
    answer: char;
begin
  reset(infile,'console:');
  rewrite(outfile,'console:');
  write(outfile,'Are you sure this will work (y/n) ?');
  read(infile,answer);
  if answer = 'y' then
  { ...  };
end.
```

Unfortunately, this program doesn't work as expected; RESET performs an implicit GET, so the program must wait until a character is typed on the console. After a character is typed, the prompt appears; however, rather than pausing after the prompt to read a character from the terminal, READ uses the character typed in to satisfy the RESET operation as a response to the prompt. The program is obviously ill-suited for interactive use.

With an interactive file i, the following rules hold for RESET and READ:

1) RESET(i) does not perform an implicit GET(i)

2) READ(i,ch) is equivalent to GET(i); ch := i^

The program shown above executes more reasonably if infile is declared with type INTERACTIVE. The program does not hang when the input file is opened, and the prompt response is not read until after the prompt message is displayed.

The definition of interactive files affects the manner in which the standard functions EOLN and EOF are used. The following code pieces are functionally equivalent (f, i, and ch are defined above):

```
while not eoln(f) do
  read(f,ch);
read(f,ch); {EOLN marker}
```

```
while not eoln(i) do
  read(i,ch);
```

With file f, the window variable f^ is always a one-character lookahead for ch; thus, the end-of-line marker must be flushed after EOLN(f) becomes true. With file i, i^ and ch contain the same character after each READ; thus, ch contains the end-of-line marker when EOLN(i) becomes true.

3.3.2 The Keyboard File

UCSD Pascal contains the predeclared file KEYBOARD for reading characters directly from the terminal keyboard. KEYBOARD is an interactive file, and is the nonechoing equivalent to the predeclared file INPUT. For example, given ch as a character variable, the statements:

```
read(keyboard,ch);
write(output,ch);
```

... are equivalent to read(input,ch).

NOTE - EOF(KEYBOARD) becomes true only after typing <null>. The console end-of-file command is read as a normal character.

3.3.3 Block Files

Block files allow low-level access to the file system; they are intended for system programming. Block files are declared with the predeclared type FILE, and may be accessed only with the BLOCKREAD and BLOCKWRITE intrinsics. These are integer-valued functions. They accept as parameters a block file identifier, buffer address, number of blocks, and (optionally) a starting block number, and return the number of blocks actually transferred. (A block is 512 bytes long.) The optional starting block number parameter allows disk files to be randomly accessed by block number; in its absence, successive block I/O operations access consecutive blocks. A disk file is viewed as a group of contiguous blocks; the first block is block 0.

Example of block I/O using explicit I/O checks and implicit starting block:

```
program FileCopy1;
var  infile,outfile: file;
     buf: packed array [1..512] of char;
     junk, blksread: integer;
     endofile: boolean;
begin
  endofile := false;
  reset(infile,'source.data');
  rewrite(outfile,'dest.data');
  while not endofile do
    begin
      {$I-}
      blksread := blockread(infile,buf,1);
      if ioresult <> 0 then writeln('disk read error');
      endofile := blksread <> 1;
      if not endofile then
        begin
          junk := blockwrite(outfile,buf,1);
          if ioresult <> 0 then writeln('disk write error');
        end;
      {$I+}
    end;
  close(infile);
  close(outfile,lock);
end.
```

Example of block I/O using implicit I/O checks and explicit starting block:

```
program FileCopy2;
const N = 5;
var   infile,outfile: file;
      buf: packed array [1..N,1..512] of char;
      blknum, blksread: integer;
begin
  reset(infile,'source.data');
  rewrite(outfile,'dest.data');
  blknum := 0;
  repeat
    blksread := blockread(infile,buf,N,blknum);
    if blockwrite(outfile,buf,blksread,blknum) <> 0 then;
    blknum := blknum + N;
  until blksread < N;
  close(infile);
  close(outfile,lock);
end.
```

3.3.4 Random Access Files

UCSD Pascal provides the SEEK intrinsic for random accessing of records in a disk file. The file must be a structured file (i.e.

any standard Pascal file except text). SEEK accepts two parameters: a file identifier, and an integer indicating the record to be accessed. SEEK moves the file window so that a subsequent GET or PUT operation accesses the specified record. The first record in a file is record 0.

NOTE - In standard Pascal, an open file is either read or written exclusively. Random access files in UCSD Pascal are opened with RESET, but can be both read (using GET) or written (using PUT).

NOTE - The standard procedure EOF can be used to check if the specified record number exceeds the number of records in the file. Calling SEEK itself always sets EOF to false, but a subsequent GET operation reveals the presence or absence of a record in the file window. If GET causes EOF to become true, the file window is past the end of the file, and the buffer variable is undefined.

WARNING - SEEK disregards the end of a file when setting a new file position. After seeking to a record position past the end of a file, PUT may be called only if the file window immediately follows the last record in the file; otherwise, the file state becomes undefined.

Example of SEEK:

```
program DataBase;
var f: file of string;
    recnum: integer;
begin
  reset(f,'string.data');
  repeat
    write('Enter record number (-1 terminates) : ');
    readln(recnum);
    if recnum < 0 then exit(program);
    seek(f,recnum);
    get(f);
    if eof(f) then writeln(' No such record')
    else
      begin
        writeln(' Current value is: ',f^);
        seek(f,recnum);   { reseek record for update }
        write(' Enter new value: ');
        readln(f^);
        put(f);
      end;
  until false;
  close(f,lock);
end {DataBase};
```

3.4 Strings

UCSD Pascal contains the predeclared data type STRING. Variables and constants of type STRING contain character sequences; the length of the character sequence stored in a string variable may vary during the execution of a program. A number of operations are provided for strings:

-- The file operators READ, READLN, WRITE, and WRITELN accept string arguments.

-- The intrinsics CONCAT, COPY, DELETE, INSERT, LENGTH, and POS perform common string operations.

-- Individual characters in a string variable may be accessed similarly to a character array.

-- All comparison operators (e.g. <>) accept string arguments.

String types are declared with a static length attribute. The default static length is 80 characters. Length attributes are explicity assigned by following the predeclared identifier STRING with an unsigned integer constant (denoting the static length) enclosed in square brackets ([ ]). The maximum length attribute is 255 characters.

The dynamic length of a string may not exceed its static length; otherwise, an execution error (i.e. " String too long") occurs.

Examples of string type declarations:

```
type  normal = string;            { default static length }

      volname = string[7];        { static length = 7 chars }

      bigstring = string[255];    { static length = 255 chars }
```

NOTE - Static length attributes allow users to minimize the amount of space allocated to strings (disk space with respect to files of strings; memory space with respect to string variables). Strings are type-compatible regardless of their static length attribute.

Example of string assignment:

```
s1 := 'this is a string constant';
s2 := s1;
```

NOTE - String constants may not exceed 80 characters.

Individual characters within a string may be referenced by indexing into the string variable (e.g. s1[5] - note that string variables are equivalent to a PACKED ARRAY OF CHAR in this respect). Valid string indices range from 1 to the current dynamic length of the

string; indices outside of this range cause an execution error (i.e. "Invalid Index") to occur.

Example of an invalid string index:

```
s1 := '1234';
s1[5] := '5';
```

NOTE - If the length of a string is 0 (i.e. its value is ""), any string indexing causes an execution error.

The relational operators =, <>, <, <=, >, >= yield a Boolean result when applied to string operands. Comparisons are performed lexicographically (e.g. word order in a dictionary). Note that trailing spaces are significant. Appendix I displays the character order.

Examples of string comparison:

```
if 'write' < 'writeln' then writeln('strings compares work');
if s1 = s2 then writeln('string vars equal');
```

When a string variable is passed as an argument to READ or READLN, all characters up to an occurrence of the end-of-line character (carriage return) are assigned to the string variable. A carriage return must by typed to terminate a string entered from the console, whether or not the input was performed with READ or READLN. By definition, READLN swallows the carriage return. READ, however, leaves the carriage return in limbo; it is picked up by the next read operation. As a result, it is suggested that strings be read from the console only with READLN.

UCSD Pascal provides the following intrinsics for string manipulation: CONCAT, COPY, DELETE, INSERT, LENGTH, and POS. CONCAT accepts two or more strings as arguments and returns a single string containing the concatenation of the string arguments. COPY extracts a character sequence from a string and returns the sequence as a string. INSERT stuffs a string value into another string. DELETE removes characters from a string. LENGTH returns an integer containing the dynamic length of a string. POS returns an integer denoting the starting position of a character pattern within a string.

Example of string intrinsics:

```
program strings;
var s1, s2, s3: string;
    int: integer;
begin
  s1 := 'The quick brown system';
  s2 := 'jumped over the lazy document';

  writeln(length(s1),' ',length('Q'));

  int := pos('brown',s1);
  writeln(int,' ',pos(s1,s2));

  s3 := concat(s1,' ',s2);
  writeln(s3);

  writeln(copy(s1,1,4),copy(s2,pos('document',s2),8));

  writeln(s1);
  s3 := 'quick brown';
  delete(s1,pos(s3,s1),length(s3));
  writeln(s1);

  insert(' is a moving target',s1,succ(length(s1)));
  writeln(s1);
end {strings}.


** Program output **

22 1
11 0
The quick brown system jumped over the lazy document
The document
The quick brown system
The  system
The  system is a moving target
```

3.4.0 String Parameters

Strings may be passed as value and variable parameters; however, the compatibility of strings having different static lengths can cause some subtle problems.

First, note that string types possessing a length attribute specification are considered structured types, and thus may not appear in the formal parameter list of a procedure or function; according to standard Pascal, only type identifiers may appear here.

Example of strings as formal parameters:

```
type bigstring = string[132];

procedure trans(param1: string; param2: bigstring);
```

Strings passed as value parameters are copied into local data areas by the called procedure. The code for this task is produced automatically by the compiler; it is executed when the procedure is first entered. If the actual parameter's dynamic length exceeds the formal parameter's static length, the execution error "String too long" occurs when the string is copied.

Example of an execution error during string copying:

```
program example;
type shortstring = string[4];

  procedure crash(param: shortstring);
  begin
    { string-copying code causes error here }
  end;

begin
  crash('oversized actual parameter');
end {example}.
```

WARNING - Strings passed as variable parameters can cause serious problems as a result of poor type-checking in UCSD Pascal. Formal parameter references within a procedure become indirect references to the actual string parameter. Within the procedure, however, the formal parameter's static length attribute overrides the string's static length; if the formal parameter's static length exceeds the string's static length, the formal parameter may be assigned values that overrun the string's data space without causing an execution error. This results in either a system crash or damage to the contents of an adjacent variable.

Example of integrity violation from poor type checking:

```
program features;

type bigstring = string[250];

var smallstring: string[10];
    victim: string;

  procedure whackstring(var param: bigstring);
  begin
    param := 'this string is larger than ten characters';
  end;

begin
  victim := 'this string will be overwritten';
  writeln('before: ',victim);
  whackstring(smallstring);
  writeln('after: ',victim);
end {features}.
```

3.5 Dynamic Variable Management

UCSD Pascal provides two sets of intrinsics for dynamic variable allocation and deallocation: the UCSD version II.0 intrinsics and the UCSD version IV.0 intrinsics. The II.0 intrinsics require less memory and execute faster than the IV.0 intrinsics. However, the IV.0 intrinsics allow the deallocation of single dynamic variables and provide support for variable-sized buffer allocation. The II.0 intrinsics are the default; the IV.0 intrinsics become available when the $H compile option (section 5.0.6) is used. The II.0 intrinsics include NEW, MARK and RELEASE (section 3.5.0). The IV.0 intrinsics include VARNEW, DISPOSE, VARDISPOSE, VARAVAIL and variations on the II.0 intrinsics (section 3.5.1).

3.5.0 The II.0 Heap

All dynamic variable allocation is performed in an area of memory known as the heap. The heap starts in low memory and grows towards high memory. The system stack starts in high memory and grows towards low memory. The NEW intrinsic is used for the allocation of a single dynamically allocated variable. Successive calls to NEW allocate variables in successive ascending memory locations, thus advancing the heap towards the stack. If the heap and stack collide, a stack overflow error occurs.

The MARK and RELEASE intrinsics are used for the deallocation of dynamically allocated variables. MARK and RELEASE accept pointer variables of any type as arguments. Given a pointer variable p, MARK(p) opens a new heap for dynamically allocated variables. The heap is identified by the value assigned to p by MARK. Subsequent calls to NEW allocate dynamic variables only in the new heap. RELEASE(p) deallocates all dynamic variables in the heap designated by p.

NOTE - New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

WARNING - Careless use of MARK and RELEASE leads to "dangling references" (i.e. pointer variables pointing to deallocated dynamic variables). Use of dangling references can cause unpredictable results, including system crashes.

NOTE - MARK and RELEASE do not check the validity of their arguments. Pointers passed to MARK must only be used as an argument to a subsequent call to RELEASE. Pointers passed to RELEASE must be initialized by a previous call to MARK.

Example of MARK and RELEASE:

```
program dynamic;
type citizenptr = ^citizen;
     citizen = record
                 name: string;
                 number: integer;
                 neighbor: citizenptr;
               end;
var list, listhead: citizenptr;

  procedure add(cloname: string; ID: integer);
  var cloneunit: citizenptr;
  begin
    new(cloneunit);
    with cloneunit^ do
      begin
        name := cloname;
        number := ID;
        neighbor := listhead;
      end;
    listhead := cloneunit;
  end {add};

begin
  mark(list);                      {allocate space for list}
  listhead := nil;
  add('Clone, Norman Q.      ' ,32763);
  add('Dumptruck, T.         ' ,32764);
  add('Maton, Otto F. S.     ' ,32765);
  add('Gleahaves, Flying R. ' ,32766);
  listhead := nil;
  release(list);                   {deallocate entire list}
end {dynamic}.
```

3.5.1 The IV.0 Heap

Dissimilarities between the II.0 heap and the IV.0 heap occur as a result of the DISPOSE intrinsic. This intrinsic is used for the deallocation of a single dynamically allocated variable. The memory space occupied by the deallocated variable is recycled by subsequent calls to NEW, assuming the space occurs in the current heap and it is large enough to accomodate the new variable. Note that subsequent calls to NEW are not guaranteed to allocate variables adjacently in memory, as is the case with the II.0 heap.

The VARNEW and VARDISPOSE intrinsics are used for the allocation and deallocation of variable-sized buffers. They accept two parameters: a pointer variable of any type and an unsigned word count. (Unsigned integers are discussed in section 7.2). The VARNEW function attempts to allocate a buffer of the requested number of words. If there is sufficient memory for such a buffer, the pointer is returned pointing to the buffer, and the requested word count is returned as the function value; otherwise the function value is zero. The VARDISPOSE procedure deallocates a

buffer of a specified size.

The VARAVAIL function accepts a string value containing a list of segments and returns the size of the largest available memory space, assuming all specified segments are memory-resident. The segment list is of the same form as that used by the MEMLOCK intrinsic described in section 3.1. VARAVAIL is calculated assuming that any specified segments that are currently nonresident would be loaded onto the system stack rather than MEMLOCKed.

NOTE - Pointer variables passed to the RELEASE, DISPOSE, and VARDISPOSE intrinsics are returned containing the value NIL.

NOTE - Pointer values passed to RELEASE must be the result of a prior MARK; otherwise, an "Invalid Heap Operation" error occurs.

NOTE - Calls to DISPOSE or VARDISPOSE must be made with the same sized structure as was used with the corresponding NEW or VARNEW call; otherwise, a system crash may occur.

NOTE - Calling NEW or VARNEW to allocate a one-word structure actually allocates two words; corresponding calls to DISPOSE and VARDISPOSE deallocate two words. Calls to MARK allocate three words in addition to a new heap; calls to RELEASE deallocate that space.

NOTE - Calls to RELEASE do not disturb MEMLOCKed segments resident on the released heap. Calls to NEW and VARNEW do not reallocate this memory until the segment has been MEMSWAPPed.

WARNING - A host and its used units must all use either the II.0 intrinsics or the IV.0 intrinsics, but not both. Units that don't use any dynamic variable allocation intrinsics are compatible with both sets of intrinsics. This constraint is enforced by the system at program invocation time. Intrinsic units escape this check and intermixing of heap mechanisms is done at the risk of the user.

NOTE - The IV.0 intrinsics are maintained in a nonresident unit (called HEAPOPS) located on the system disk. They are loaded into memory along with any program that uses them. Thus, the system disk must be in the system drive when such programs are invoked; otherwise, a system I/O error occurs. HEAPOPS may be made permanently resident by transferring it from the system support library to the intrinsics library. See section 2.3.5 of the System User's Manual for details.

Example of extended memory management usage:

```
{$H+}
program extended;
type buffer = record
                case integer of
                  2: twoblocks  : array [0..511] of integer;
                  4: fourblocks : array [0..1023] of integer;
              end;

  procedure method1(size: integer);
  var bufptr: ^buffer;
  begin
    if size = 2 then
      new (bufptr, 2)
    else
      new (bufptr, 4);
    {use buffer for something}
    if size = 2 then
      dispose (bufptr, 2)
    else
      dispose (bufptr, 4);
  end;

  procedure method2(size: integer);
  var bufptr: ^buffer;
  begin
    if varnew (bufptr, size*256) <> 0 then
      begin
        {use buffer for something}
        vardispose (bufptr, size*256);
      end;
  end;

begin
  method1(2);
  method2(4);
end {extended}.
```

3.6 Extended Precision Arithmetic

UCSD Pascal provides a data type known as "long integer" for extended precision arithmetic. Long integers are used like standard integers, but may contain up to 36 digits.

Long integer types are defined by appending a length attribute to the predeclared type INTEGER. Length attributes are similar to those used in string types: an unsigned integer constant delimited by square brackets ([and]). The maximum length attribute is 36. Long integer length attributes specify the maximum number of digits expected; they do not impose a strict upper bound on the number of digits allowed. Length attributes are used to minimize the amount of space allocated to long integers (disk space with respect to files containing long integers; memory space with respect to long integer variables). Long integers are type compatible regardless of their length attribute.

Examples of long integer type defintions:

```
type  shortint = integer[3];
      longint = integer[36];     { max size }
```

Depending on their value, integer constants become either integer constants or long integer constants. Constants in the range -32767..32767 default to integer constants; constants outside this range are treated as long integer constants.

Examples of integer constants:

```
const Rydberg = 10973731;    { long integer }
      Hoffman = 0;           { integer      }
```

In general, long integers may be used anywhere it is syntatically correct to use REAL types. For instance, long integers and integers may be mixed in arithmetic expressions; integers are implicitly converted to long integers in mixed expressions. Integers may be assigned to long integers; however, long integers must be explicity converted to integers (with the standard function TRUNC). Note that direct conversion between long integers and reals is impossible.

WARNING - See Appendix G for some problems arising from the use of mixed expressions.

The arithmetic operators +, -, *, and DIV yield a long integer result when applied to long integer operands. (Note that MOD is not defined.) The relational operators =, <>, <, <=, >, >= yield a Boolean result when applied to long integer operands.

Unlike integers, long integers enforce overflow checking; when a long integer variable is assigned a value larger than it can contain, the execution error "Integer Overflow" occurs.

WARNING - Intermediate expression results should not exceed 36 digits - integer overflow may not be detected.

Example of a program using long integers:

```
program example;
var long: integer[36];
begin
  long := 1;
  repeat
    writeln(long);
    long := long * 2;
  until long > 20000000;
end.
```

All file I/O operators (including READ and WRITE) accept long integer arguments.

WARNING - Backspace does not work when reading a long integer from the console.

NOTE - The long integer intrinsics are maintained in a nonresident unit (called LONGINTS) located on the system disk. They are loaded into memory along with any program that uses them. Thus, the system disk must be in the system drive when such programs are invoked or a system I/O error occurs. LONGINTS may be made permanently resident by transferring it from the system support library to the intrinsics library. See section 2.3.5 of the System User's Manual for details.

The standard function TRUNC is extended to accept long integers as arguments (as with reals, an execution error occurs if the argument is outside of the range for integers). The UCSD intrinsic STR converts long integers to strings. Given a long integer L and a string S, STR(L,S) assigns to S a character string representation of the value in L (complete with minus sign, if required).

Example of STR:

```
program money;
type bucks = integer[30];
var CashFlow: bucks;

  procedure PrintDough(amount: bucks);
  var dollars: string;
  begin
    str(amount, dollars);
    insert('.', dollars, pred(length(dollars)));
    writeln('$',dollars);
  end {PrintDough};

begin
  CashFlow := 2323972233;
  PrintDough(CashFlow);
  PrintDough(199);
end {money}.

**Program output**

$23239722.33
$1.99
```

3.6.0 Long Integer Parameters

Long integers may be passed as value and variable parameters; however, the compatibility of types with different length attributes may cause some subtle problems.

First, note that long integer types are considered structured types, and thus may not appear in the formal parameter list of a procedure or function; according to standard Pascal, only type identifiers may appear here.

Example of long integers as formal parameters:

```
type longint = integer[32];

procedure trans(param1, param2: longint);
```

When long integers are passed as variable parameters, long integer types with different length attributes lose their type compatibility. The formal and actual parameter types must possess identical length attributes.

Long integers passed as value parameters are adjusted by the caller to the size declared in the callee's formal parameter list. The code for this task is produced automatically by the compiler; it is executed by the calling procedure. If the value of the actual parameter is too large to fit in the formal parameter, the execution error "Integer overflow" occurs when the long integer is adjusted.

Example of an execution error during parameter adjust:

```
program example;
type shortint = integer[4];

  procedure crash(param: shortint);
  begin
    { ...  }
  end;

begin
  { adjust code causes error here }
  crash(3294875938475);
end {example}.
```

3.7 Extended Comparisons

UCSD Pascal extends the relational operators to accept pointer, array, and record types as operands.

3.7.0 Records and Arrays

The relational operators = and <> yield a Boolean result when applied to array and record operands. Operands must be type compatible (see section 2.13). Operators compare entire structured variables; structures are equal if and only if the fields comprising the structures are equal.

WARNING - Structured comparisons are implemented as multiword comparisons; structured variables which fail to completely utilize their allocated data space render structured comparisons useless. Relational operators should not be used in the following cases:

-- Records containing string types.

-- Most packed arrays and records.

Data space for strings is allocated statically; string values expand and contract in their data area at run time. The area between the end of a string value and the end of its data space is undefined, but is considered significant in a structured comparison; thus, comparison of records containing strings does not work correctly.

The UCSD Pascal compiler's packing algorithm may leave unused bit fields in the words comprising the data space allocated for packed records and arrays. Because the unused bit fields contain undefined values, comparison of packed records and arrays may not work correctly. The exceptions to this restriction are byte arrays (e.g. PACKED ARRAY OF CHAR) and packed variables which (by chance or design) completely utilize their allocated data spaces. See section 7.0 for a description of the packing algorithm.

Example of record and array comparison:

```
program compare;
var a,b: record
           i,j: integer;
           r: real;
         end;
    x,y: array[0..150] of integer;
    count: integer;
begin
  for count := 0 to 150 do
    begin x[count] := 4; y[count] := 4 end;
  with a do
    begin i := 4; j := 6; r := 3.14159 end;
  with b do
    begin i := 4; j := 6; r := 2.71828 end;
  if (x = y) and (a <> b) then
    writeln('truth is beauty')
  else
    writeln('truth is rude');
end {compare}.
```

3.7.1 Pointers

The relational operators =, <>, <, <=, >, >= yield a Boolean result when applied to pointer operands. These operators are implemented as unsigned integer comparisons.

3.8 Byte Array Manipulation

UCSD Pascal provides the intrinsics MOVELEFT, MOVERIGHT, SCAN, FILLCHAR, and SIZEOF for efficient manipulation of large arrays of data. MOVELEFT and MOVERIGHT perform mass movement of data within arrays. FILLCHAR initializes arrays. SCAN searches an array for the presence (or absence of) a byte value. These intrinsics are intended for use with byte (e.g. character) arrays; however, the lack of type checking on their parameters allows them to be used as general purpose data manipulators (with the understanding that the price of freedom is responsibility). These intrinsics are byte-oriented: address parameters are resolved to byte addresses; count values are byte counts; byte values are characters or integers in the range 0..255.

WARNING - Count values are treated as signed integers. Negative count values in MOVELEFT, MOVERIGHT, and FILLCHAR are treated as zero byte counts. Be wary of large unsigned count values; it may be necessary in some cases to divide an operation into two parts in order to avoid this problem.

SIZEOF is a (compile time) function which accepts either a variable or type identifier as an argument and returns an integer value indicating the number of bytes allocated for the data type denoted by the identifier. SIZEOF shifts the burden of determining the size of a data type onto the compiler, thus making it safer and easier to use the byte array intrinsics.

NOTE - If a record contains variant fields, SIZEOF uses the largest variant when determining its size.

WARNING - SIZEOF ignores variable references. For example:

```
SIZEOF(p^)
```

... returns the size of the pointer variable p rather than the size of the object which p points to; in this case, it is necessary to pass SIZEOF the identifier denoting p's base type.

FILLCHAR accepts a starting address, integer byte count, and byte value. Beginning with the starting address, it initializes <byte count> bytes to the indicated byte value.

MOVELEFT and MOVERIGHT perform mass movement of data; both accept a source address, destination address, and integer byte count. The bytes between the source address and the address formed by <source address> + <byte count> - 1 comprise the source array. The bytes between the destination address and the address formed by <dest address> + <byte count> - 1 comprise the destination array.

WARNING - Array indices on the PDQ-3 are treated as signed integers. Use of an index whose value is less than the declared lower bound of the the source or destination array may yield unexpected or fatal results.

MOVELEFT and MOVERIGHT move data from the source array to the

destination array one byte at a time. MOVERIGHT starts at the left end of both arrays and copies bytes traveling right; it is used to prevent data moved to the left (i.e. lower addresses) from overwriting source bytes that haven't been moved to their destination. MOVERIGHT starts at the right end of both arrays and copies bytes traveling left; it is used to prevent data moved to the right (i.e. higher addresses) from prematurely overwriting source bytes.

NOTE - Movement of data blocks between nonoverlapping arrays is usually performed with MOVELEFT, as it represents a more natural style of moving data. Certain combinations of MOVELEFT and MOVERIGHT with overlapping source and destination addresses produce complex results; their use is not recommended without some forethought (see example below).

Example of byte array manipulators:

```
program blockmove;
var source1, source2: packed array[0..511] of char;
    dest: packed array[0..1023] of char;
    int: integer;
begin
  fillchar(source1, sizeof(source1), 0);
  fillchar(source2, sizeof(source2), 1);
  moveleft(source1[0], dest[0], 512);
  moveright(source2[0], dest[512], 512);
  moveleft(dest[511], int, 2);
end.
```

Example of shady use of MOVELEFT and MOVERIGHT:

```
program boggle;
var bytes: packed array [1..30] of char;
begin
  bytes := 'this is the text in this array';
  writeln ('123456789012345678901234567890');
  writeln(bytes);
  moveright(bytes[10],bytes[1],10);
  writeln(bytes);
  moveleft(bytes[1],bytes[3],10);
  writeln(bytes);
  moveleft(bytes[23],bytes[2],8);
  writeln(bytes);
end.

** Program Output **

123456789012345678901234567890
this is the text in this array
ne text ine text in this array
nenenenenenenetext in this array
nis arrayenetext in this array
```

SCAN is a function which accepts an integer scan length, a partial Boolean expression, and a starting address.

A partial Boolean expression has the following form:

```
<partial Boolean expression> ::= <relop> <target>
<relop> ::= "=" | "<>"
<target> ::= <character variable> | <character constant>
```

Partial expressions appear as half of a Boolean expression; the missing operand is defined to be the byte value currently pointed to by SCAN. The partial expression is evaluated for each byte examined by SCAN; if it evaluates to true, SCAN returns immediately.

Starting with the byte at the starting address, SCAN examines successive bytes until it either finds a byte value satisfying the partial expression or exceeds the scan length. SCAN returns an integer value indicating the number of bytes examined.

The scan length may be positive or negative. If the scan length is negative, SCAN proceeds backwards (i.e. towards lower addresses) from the starting address; otherwise, SCAN proceeds forward through memory. SCAN returns the offset from the starting address; this value is negative when scanning backwards, and positive when scanning forwards. SCAN returns 0 when it terminates on the byte at the starting address.

Example of SCAN:

```
program SCANdemo;
var farkle: string;
begin
  farkle := '.....the pterac is a member of the USCD family';
  writeln(scan(-26, = ':', farkle[30]));          { writes -26 }
  writeln(scan(100, <>'.', farkle[1] ));          { writes 5   }
  writeln(scan(15 , = ' ', farkle[1] ));          { writes 8   }
end.
```

3.9 Device I/O

UCSD Pascal provides the intrinsics UNITREAD, UNITWRITE, UNITCLEAR, UNITSTATUS, UNITBUSY, and UNITWAIT for accessing I/O devices. These intrinsics comprise the I/O level known as "unit I/O"; this is the lowest level of I/O available to the system, and must be used with care. UNITREAD and UNITWRITE are described in section 3.9.0. UNITCLEAR, UNITBUSY, and UNITWAIT are described in section 3.9.1. UNITSTATUS is described in section 3.9.2.

The primary argument to the unit I/O intrinsics is the unit number, specifying an I/O device. Unit numbers and device assignments for the PDQ-3 are described in Appendix D.

NOTE - The compiler does not generate I/O checks (section 5.0.4) after calls to unit I/O intrinsics; I/O checks must be explicitly performed by examining the I/O completion status after every operation. (I/O completion status is examined with the IORESULT intrinsic - see section 3.11.8 for details.)

NOTE - The system routines implementing unit I/O are protected from task contention.

3.9.0 UNITREAD and UNITWRITE

The UNITREAD and UNITWRITE intrinsics (in most cases) transfer data between memory and an I/O device. They accept a unit number, I/O buffer, byte count, and two optional parameters: a block number and a control word. The I/O buffer is specified by either an indexed or an unindexed variable name (e.g. Arr[Index] or Arr). The byte count is an unsigned integer in the range 0..65535. The block number is a signed integer used in I/O involving block-structured devices; its default value is zero. The control word specifies special processing options; its default value is zero. The syntax for UNITREAD and UNITWRITE is described in sections 4.42 and 4.45. Their semantics are device dependent and are described in Appendix D.

WARNING - The most common results of incorrect use of UNITREAD and UNITWRITE are damaged disk files and/or directories and program crashes caused by overrunning data buffers on read operations. No range checking is performed on accesses to the I/O buffer.

WARNING - Array indices on the PDQ-3 are treated as signed integers. Use of an index whose value is less than the I/O buffer's declared lower bound may yield unexpected or fatal results.

NOTE - On the PDQ-3, a variable of type CHAR occupies a full word; the actual character value occupies the low-order byte. UNITREADs and UNITWRITEs on these quantities also operate on the low-order byte. This fortunate circumstance does not necessarily occur on other processors. Instances of these intrinsics operating exclusively on PACKED ARRAYs of CHAR are guaranteed to be portable to those processors.

NOTE - Variables of type CHAR used with a UNITREAD call for 1 byte should be initialized before the UNITREAD call. This sets the high-order byte of the variable to zero so the variable may be correctly compared to other characters.

NOTE - Unit I/O intrinsics are used in a few cases for system actions unrelated to device I/O. UNITREAD on unit 0 implements a feature known as "time delay" (see appendix D and the SysUtil unit in the Library User's Manual). The keyboard type-ahead buffer can be manipulated by applying UNITWRITE to unit 3 (see appendix D for details).

Example of UNITREAD and UNITWRITE:

```
program unitdemo;
var buff: packed array[0..2047] of char;
    ch: char;

  procedure putline(msg: string);
  var cr: packed array [0..0] of char;
  begin
    if length(msg) > 0 then
      unitwrite(1,msg[1],length(msg));
    cr[0] := chr(13);
    unitwrite(1,cr,1);
  end {putconsole};

  procedure getkey(var key: char);
  begin
    key := ' ';
    unitread(2,key,1);
  end {getkey};

begin
  putline('*** Screen Garbage Generator ***');
  putline('');
  putline('       G(arbage E(xit ');
  repeat
    getkey(ch);
  until ch in ['g','G','e','E'];
  if ch in ['e','E'] then exit(program);
  unitread(4,buff,2048,2);
  if ioresult <> 0 then
    begin
      putline('>>> I/O error detected');
      exit(program);
    end;
  unitwrite(1,buff,2048);
  putline('That''s all, folks...');
end.
```

3.9.1 UNITCLEAR, UNITBUSY, and UNITWAIT

UNITCLEAR, UNITBUSY, and UNITWAIT accept a unit number as a

parameter. Their syntax is described in chapter 4. Their semantics are device dependent and are described in Appendix D.

The UNITCLEAR procedure resets and/or initializes I/O devices. In the case of serial input devices it usually clears the type-ahead buffer. The value of IORESULT after a UNITCLEAR call is often used to test the existence of the device.

The UNITBUSY function is used to poll the status of an I/O device. It returns TRUE if the device has not completed a pending I/O; otherwise it returns FALSE. When it is used on a serial input device, it returns TRUE if no character has been received; otherwise it returns FALSE. The UNITSTATUS procedure returns more complete information than UNITBUSY (see section 3.9.2).

The UNITWAIT procedure performs no actions on the PDQ-3.

Example of UNITCLEAR and UNITBUSY:

```
program serialdemo;
var buff: packed array[0..0] of char;
begin
  unitclear(2);          {clear keyboard type-ahead}
  while unitbusy(2) do
    writeln('please type a character:');
  unitread(2,buff,1);
  writeln ('character received: ', buff);
end.
```

3.9.2 UNITSTATUS

The UNITSTATUS procedure accepts a unit number, a status record, and an unused integer as parameters. It returns status information on the specified device. The format of the status record depends on the device being polled; it may be of any type, but should occupy at least 30 words. The unused parameter should be passed as either zero or one.

The IsBlocked field is located in the same place relative to the beginning of the record in each format. It indicates the format being used.

The format of a status record for a serial device is:

```
SerialStatus = record                {IsBlocked = FALSE}
                 CharsQueued     : integer;
                 QueueSize       : integer;
                 Filler1         : array [0..5] of integer;
                 DeviceName      : string[7];
                 case IsBlocked : boolean of
                   false: (VolName : string[7];
                           InUnit  : boolean;
                           Safety  : array [0..11]
                                       of integer);
               end; {SerialStatus}
```

In this format, the IsBlocked boolean is always FALSE. CharsQueued gives the number of characters currently available for input. QueueSize contains the maximum number of characters that can be queued. DeviceName contains a string indicating the type of device being polled (e.g. "DLV-11J" for a DEC DLV-11J serial card). VolName contains the volume name for the device (e.g. CONSOLE for unit 1). InUnit is TRUE if the device is an input device; otherwise FALSE.

The format of a status record for a block-structured device is:

```
BlockedStatus = record               {IsBlocked = TRUE}
                  Filler2         : integer;
                  BytesSector     : integer;
                  SectorsTrack    : integer;
                  TracksDisk      : integer;
                  Filler3         : array [0..3] of integer;
                  DeviceName      : string[7];
                  case IsBlocked : boolean of
                    true: (HardDisk     : boolean;
                           PhysicalUnit : integer;
                           Safety       : Array [0..14]
                                            of integer);
                end; {BlockedStatus}
```

In this format, the IsBlocked boolean is always TRUE. BytesSector gives the number of bytes per sector on the device as of the last time it was accessed. SectorsTrack contains the number of sectors per track. TracksDisk contains the number of tracks per disk. DeviceName is a string indicating the type of device being polled (e.g. "RL-02" for a DEC RL-02 disk drive). HardDisk contains TRUE if the device is a hard disk drive. In this case, the value of TracksDisk does not account for a bootstrap track (see Appendix D for details). PhysicalUnit denotes the hardware controller's address for the device.

The format of a status record for the system clock (unit 0):

```
SystemStatus = record              {IsBlocked = FALSE}
                 LastWord          : integer;
                 LowTime           : integer;
                 HighTime          : integer;
                 Filler4           : array [0..4] of integer;
                 DeviceName        : string[7];
                 case IsBlocked    : boolean of
                   true: (VolName  : string[7];
                          InUnit   : boolean;
                          Safety   : array [0..11]
                                       of integer);
               end; {SystemStatus}
```

LastWord gives the address of the last location in memory. LowTime and HighTime are the current value of the system clock (see the TIME intrinsic - section 3.11.4). DeviceName is a string indicating the type of clock being polled (e.g. "PDQ Clk" for the onboard PDQ-3 clock). IsBlocked is FALSE, VolName contains "CLOCK", and InUnit is FALSE.

NOTE - The value of IORESULT returned by the UNITSTATUS intrinsic reflects the presence of device handlers for the device polled. It does not indicate device readiness.

3.10 Inline Machine Code

UCSD Pascal provides the PMACHINE intrinsic for generating in-line machine code within Pascal programs. In-line machine code is used for programming low level operations which cannot be expressed efficiently (if at all) in the Pascal language.

WARNING - PMACHINE is the lowest level intrinsic in UCSD Pascal. Its use requires familiarity with the PDQ-3 instruction set (see the Architecture Guide for details) and extreme care in specifying code sequences.

DISCLAIMER - Incorrect use of PMACHINE in any program invalidates all warranties on ACD system software.

PMACHINE accepts a series of items; multiple items are separated by commas. An item is either code, an expression, or an address reference.

A code item consists of a constant value or constant identifier denoting an integer between 0 and 255. PMACHINE emits a single byte in the code with the specified value. Values outside of this range (e.g. signed constants) have only their least significant byte emitted. Code items are used for emitting P-code instructions and instruction operands.

An expression is any valid Pascal expression enclosed in parentheses. PMACHINE generates code to evaluate the expression and leave the result on the stack.

An address reference is any valid Pascal variable reference preceded by the character '^'. PMACHINE generates code to leave the address of the specified variable on the stack.

Example of PMACHINE:

```
program PcodeDemo;
const STM = 142;
type complex = record re,im: real end;
     vector =  array [0..10] of complex;
var  speed: ^vector;
     x: real;
 begin
   new(speed);
   x := 3.14159;
   pmachine(^speed^[7].re, (x / 1.0), STM, 2);
   writeln('result is: ', speed^[7].re);
 end {PcodeDemo}.
```

Standard PMACHINE operations:

```
const   STO = 196;      { store indirect   }
        IXA = 215;      { index array      }
        SIND0 = 120;    { load indirect    }
        BNOT = 159;     { Boolean negation }
        LEUSW = 180;    { unsigned <=      }
        GEUSW = 181;    { unsigned >=      }

var     i,j,index: integer;
        b: boolean;
        p: ^integer;
        pb: ^boolean;
        a: array[0..0] of integer;

{ p^ will reference memory address FC24 hex }
PMACHINE ( ^p, (-988), STO);

{ p^ will reference a[index] }
PMACHINE ( ^p, ^a, (index), IXA, 1, STO);

{ p := pb }
PMACHINE ( ^p, (pb), STO);

{ i := p^ }
PMACHINE ( ^i, (p), SIND0, STO);

{ b := i <= j (unsigned) }
PMACHINE ( ^b, (i), (j), LEUSW, STO);

{ b := i < j (unsigned) }
PMACHINE ( ^b, (i), (j), GEUSW, BNOT, STO);
```

3.11 Miscellaneous Extensions

This section describes miscellaneous extensions to standard Pascal. Sections 3.11.0 and 3.11.1 describe alterations of the syntax rules for identifiers and declaration parts respectively. Section 3.11.2 describes extension of the standard function ORD to perform pointer to integer type conversion.

The remaining extensions are the following UCSD intrinsics: GOTOXY for console cursor positioning (section 3.11.3), TIME for reading the system clock (section 3.11.4), PWROFTEN for real powers of ten (section 3.11.5), ATAN as an alternative name for the standard function ARCTAN (section 3.11.6), EXIT for terminating procedures or programs (section 3.11.7), IORESULT for checking the system I/O completion status (section 3.11.8), MEMAVAIL and RMEMAVAIL for checking the amount of unused memory (section 3.11.9), HALT for invoking the system monitor (section 3.11.10), and the compiler intrinsics TREESEARCH and IDSEARCH (section 3.11.11).

3.11.0 Identifiers

Identifiers in UCSD Pascal may contain the underscore character "_". Occurrences of "_" are ignored by the compiler; thus, the identifiers "procnum" and "proc_num" are equivalent.

NOTE - Identifiers are significant to 8 characters.

WARNING - Although they contribute to program readability, long variable names should be used carefully in UCSD Pascal, as it is disconcertingly easy for two "different" long variable names to map into the same identifier because of the eight character rule. Identifier aliases can cause mysterious compiler syntax errors and/or elusive program bugs.

The following identifiers are equivalent in UCSD Pascal:

```
identifier
i_dent_i_fi_er
Identifier
IDENTIFI
I_dent_I_fire
identifier_of_sparse_matrix_
```

3.11.1 Declaration Parts

Suites of related programs often must share a common group of uses-, constant-, type-, variable-, and procedure declarations. In UCSD Pascal, this can be done by placing the source code defining the group in a separate include file (section 5.0.1), and having each program in the suite include the declarations into its declaration part.

Standard Pascal restricts the ordering of declarations in a declaration part so that uses are declared before constants, constants are declared before types, and so on. An include file containing a set of related declarations would not compile successfully when included in a host program's declaration part, as the declaration order would be violated. (e.g. given an include file containing uses-, constant-, type-, and variable declarations, what is the proper location in the host program's uses-, constant-, type-, and variable declarations for the include file directive?) Thus, UCSD Pascal relaxes the restrictions on declaration order for include files appearing in declaration parts.

NOTE - Files containing uses, constant, type, or variable declarations may not be included after a procedure body; however, they may be included after forward declarations.

Example of included declarations:

*** The include file (named INC1.TEXT):

```
const H = 'Hi, guys! This is Eddy, your shipboard computer!';
type  car = record
              make: string;
              license: integer;
            end;
var   c, a: car;

  procedure rice(r: car);
  begin
    a := c;
    writeln(H);
  end;
```

*** The host program:

```
program margorp;
const N = 5;
var i,j,k: integer;

{$I INC1.TEXT}

  procedure useless;
  begin
  end;

begin
  with c do
    begin make := 'Edsel'; license := 10101 end;
  writeln(H);
  for i := 1 to N do
    begin writeln('and again...'); rice(c) end;
end {margorp}.
```

3.11.2 Pointer Type Conversion

In UCSD Pascal, the standard procedure ORD is extended to accept pointer types. This extension should only be used in machine-specific tasks requiring pointer-to-integer type conversion.

NOTE - Pointer values on the PDQ-3 may be considered scalar values (unsigned 16-bit) in the range 0..65535 which correspond to memory addresses. Integers (signed 16-bit) returned by ORD(<pointertype>) are in the range -32768..32767. See section 7.2 for more information.

3.11.3 Screen Control

The UCSD intrinsic GOTOXY provides terminal-independent X-Y coordinate cursor positioning. GOTOXY is used in conjunction with READ, READLN, WRITE, and WRITELN to create formatted screen displays and prompt lines. See section 8.3 in the System User's Manual for more information on the GOTOXY intrinsic.

Example of GOTOXY:

```
program SControl;
type horz = 0..79;
     vert = 0..23;

  procedure putline(x: horz; y: vert; line:string);
  var len: integer;
  begin
    gotoxy(x,y);
    write(line);
    gotoxy(x,y+1);
    if length(line) > 0 then
      for len := 1 to length(line) do write('-');
  end;

begin
  putline(37, 2,'North');
  putline(76,12,'East' );
  putline(37,21,'South');
  putline(0 ,12,'West' );
end {SControl}.
```

3.11.4 Clock Access

The UCSD intrinsic TIME provides access to the system clock. The clock is defined as a 32-bit unsigned integer incremented every 60'th of a second. The clock value is returned in a pair of integers passed as variable parameters. Note that the integers contain unsigned values (see section 7.2).

NOTE - The UNITSTATUS intrinsic described in section 3.9.2 may also be used to read the system clock.

Example of TIME:

```
program timer;
var notused, starttime, endtime: integer;
    count, limit, elapsedtime: integer;
begin
  writeln('** Clock Demo **');
  repeat
    write('enter delay (0 terminates): ');
    readln(limit);
    time(notused,starttime);
    for count := 1 to limit do
      notused := notused * 100 div 100;
    time(notused,endtime);
    elapsedtime := endtime - starttime;
    writeln(' elapsed time = ', (elapsedtime +30) div 60,
            ' seconds');
  until limit = 0;
end {timer}.
```

3.11.5 Powers of Ten

The UCSD intrinsic PWROFTEN (short for "power of ten") accepts an integer argument whose value is in the range 0..38 and returns a real result equal to ten raised to the power of the argument.

NOTE - Arguments less than 0 cause PWROFTEN to always return ten to the zero'th power (i.e. 1). Arguments larger than 38 cause the execution error "Floating point error".

Example of PWROFTEN:

```
program powers;
var i: integer;
begin
  repeat
    write('enter arg: ');
    readln(i);
    writeln('arg is ',i,' result is ',pwroften(i));
  until i = 0;
end {powers}.
```

3.11.6 Arctangent Synonym

Standard Pascal defines ARCTAN as the standard function for the arctangent function. In UCSD Pascal, both ARCTAN and ATAN denote the arctangent function.

3.11.7 Procedure Termination

The UCSD intrinsic EXIT accepts a procedure, function, or program identifier as an argument; it causes execution to continue at the end of the block named by the identifier. The simplest case of EXIT occurs when the identifier denotes the current block; EXIT jumps to the end of the current block. If the EXIT argument specifies a routine at an outer level, all routines on the call chain between the current routine and the specified routine are also terminated. The entire program can be terminated by calling EXIT with either the program identifier or the reserved word PROGRAM.

NOTE - If EXIT specifies a recursively invoked procedure, only the most recent invocation is terminated.

NOTE - A process may not EXIT out of its block. Attempts to do so result in the termination of the process.

WARNING - EXIT statements are not legal in either the initialization or termination sections of a unit. Moreover, attempts to EXIT(PROGRAM) from an implementation procedure during the execution of either an initialization or a termination section result in a runtime error.

NOTE - GOTO statements naming a label outside of the current block (called "out-of-block" GOTO's) are not implemented in UCSD Pascal; the EXIT intrinsic is used to provide an alternative (albeit limited) form of out-of-block GOTO.

NOTE - A CLOSE(<file>, NORMAL) is performed on all files local to a procedure terminated by a call to EXIT.

WARNING - When EXIT is used to terminate a function, the function value must have been assigned beforehand; otherwise, the function returns an undefined value when exited.

Example of EXIT:

```
program exitdemo;
var num: integer;

  procedure readNatural(var int: integer);
  var ch: char;

    procedure blowout(errmsg: string);
    var ch: char;
    begin
      writeln;
      writeln('>>>Error: ',errmsg);
      write('   type <space> to continue, "!" to escape');
      repeat read(keyboard,ch) until ch in [' ','!'];
      writeln(ch);
      if ch = '!' then exit(program)
      else exit (readNatural);
    end {blowout};

  begin
    int := 0;
    repeat
      read(ch);
      if not (ch in ['0'..'9',' ']) then
        blowout('Input format');
      if ch = ' ' then exit(readNatural);
      if (int > maxint div 10) or  ((int = maxint div 10) and
         (ord(ch) - ord('0') > maxint mod 10)) then
           blowout('Integer too large');
      int := int * 10 + ord(ch) - ord('0');
    until false;
  end {readNatural};

begin
  repeat
    write(' enter nonnegative number (17 terminates): ');
    readNatural(num);
    writeln(' number entered is: ',num);
  until num = 17;
end {exitdemo}.
```

3.11.8 I/O Completion Status

The UCSD intrinsic IORESULT returns an integer result containing the current value of the system I/O completion status. The status is reset after every I/O operation (including file operations). Calling IORESULT is usually unnecessary, as the compiler automatically generates I/O checks after every file I/O operation; however, if I/O checks are suppressed (using the $I compile option - see section 5.0.4), the value returned by IORESULT should be explicitly checked after each I/O operation to prevent I/O errors from causing program errors.

IORESULT is used in programs which substitute their own error

checking and recovery for the system's error handling facilities.

NOTE - Appendix A lists the standard I/O result values and their definitions. Appendix C lists conditions causing bad I/O results.

NOTE - Nonstandard I/O results may be defined using the PROGOPS and EXCEPINFO system units. See the Library User's Manual for details.

NOTE - Each task maintains its own I/O status; thus, I/O results are protected from task contention.

NOTE - The low-level unit I/O intrinsics (section 3.9) always require explicit I/O checks. The compiler does not generate I/O checks after occurrences of these intrinsics.

WARNING - Because the I/O completion status word is reset after every I/O operation, care must be taken to preserve I/O result values between their detection and subsequent actions. In the following example, the bad I/O result is inadvertently obliterated (by the preceding string write) before it reaches the console:

```
{$I-}
program inoperative;
var f: file;
begin
  reset(f,'nonexistent.file.text');
  writeln('I/O result after file open is ',ioresult);
end {inoperative}.
```

Section 7.5 contains another example using IORESULT.

Example of IORESULT:

```
program IOdemo;
var num: integer;

  procedure getnum(prompt: string; var int: integer);
  begin
    {$I-}       { suppress I/O checks }
    repeat
      write(prompt);
      readln(int);
    until ioresult = 0;
    {$I+}
  end {getnum};

begin
repeat
  getnum('Enter number (-1 terminates): ',num);
  writeln(' number returned is: ',num);
until num = -1;
end {IOdemo}.
```

3.11.9 Memory Available

The UCSD intrinsics RMEMAVAIL and MEMAVAIL return the number of unused words in memory. RMEMAVAIL returns a real value; MEMAVAIL returns an unsigned integer.

NOTE - Values returned by MEMAVAIL should be treated as 16-bit values in the range 0..65535; unfortunately, values in the range 32768..65535 are considered negative numbers in the signed 16-bit representation used for integer types and operators. See section 7.2 for more information.

NOTE - Values returned by RMEMAVAIL and MEMAVAIL are best used in conjunction with the II.0 heap intrinsics. They return the number of words between the system stack and current heap. When using the IV.0 heap intrinsics, there may be additional unused space in the heap as a result of calls to DISPOSE. Therefore, the VARAVAIL intrinsic should be used in conjunction with the IV.0 heap intrinsics. See section 3.5 for further information.

Example of MEMAVAIL and RMEMAVAIL:

```
program memDisplay;
begin
  writeln(' System has ',rmemavail,' words available');
  writeln(' Integer memavail value = ',memavail);
end {memDisplay}.
```

Section 7.4 contains other examples.

3.11.10 Breakpoint Trap

The UCSD intrinsic HALT invokes the system breakpoint handler. The default breakpoint handler prints a halt message and prompts for an input from the standard input. Typing <return> resumes program execution.

NOTE - The system breakpoint handler may be altered by replacing the HALTUNIT system unit. See section 6.4 for further details.

Example of HALT:

```
program succinct;
begin
  halt;
end {succinct}.
```

3.11.11 Compiler Support

The compiler uses the UCSD intrinsics IDSEARCH and TREESEARCH for scanning identifiers and maintaining symbol trees respectively. IDSEARCH is unsuited for use outside of the compiler, and is not described in this manual; however, TREESEARCH performs a sufficiently generalized task to merit description.

TREESEARCH manages binary trees ordered by the contents of an 8 character field. A node in the tree must have the following structure:

```
type nodeptr = ^node;
     node = record
              name: packed array [1..8] of char;
              left_link, right_link: nodeptr;
              ...
              { user-defined record fields }
              ...
            end {node};
```

TREESEARCH accepts a tree root pointer, node pointer, and an 8 character packed array as arguments; it returns an integer as a function result, and also assigns a value to the node pointer (variable) parameter. If a node in the tree matches the array argument, TREESEARCH returns the value 0 as a function result; the node pointer is set to the node. If no node in the tree matches the array argument, TREESEARCH returns either 1 or -1; the node pointer is set to the last node searched. 1 indicates that the array argument is greater than the value of the last node (and should be inserted on its right link); -1 indicates that the argument is less than the last node (and should be inserted on its left link).

NOTE - Trees are constructed by the PDQ-3 TREESEARCH intrinsic such that right (post-order) traversals of a tree visit the nodes in lexicographical order of their name fields. This ordering may vary in other UCSD Pascal implementations.

Example of TREESEARCH:

```
program tree_demo;
type alpha = packed array [1..8] of char;
     nodeptr = ^node;
     node = record
              name: alpha;
              llink, rlink: nodeptr;
              value: integer;
            end {node};
var  root: nodeptr;
     cmd: char;

  procedure get_name(var name: alpha);
  var s: string;
      cnt: integer;
  begin
    readln(s); name := '        ';
    for cnt := 1 to length(s) do
      if cnt <= 8 then name[cnt] := s[cnt];
  end {get_name};

  procedure find_node;
  var entry: nodeptr;
      search: alpha;
  begin
    write(' Enter name: ');
    get_name(search);
    if treesearch(root,entry,search) <> 0 then
      writeln('  Entry not found')
    else with entry^ do
      writeln('  Name: ',name,' Value: ',value);
  end {find_node};

  procedure print_nodes(tree: nodeptr);
  begin          {print nodes in ascending lexigraphic order}
    if tree^.rlink <> nil then
      print_nodes(tree^.rlink);
    with tree^ do
      writeln('  Name: ',name,' Value: ',value);
    if tree^.llink <> nil then
      print_nodes(tree^.llink);
  end {print_nodes};
```

```
  procedure add_node;
  var new_node, entry: nodeptr;
      result, new_val: integer;
      new_name: alpha;
  begin
    write(' Enter name: ');
    get_name(new_name);
    result := treesearch(root,entry,new_name);
    if result = 0 then
      writeln('  Entry already exists')
    else
      begin
        new(new_node);
        with new_node^ do
          begin
            write(' Enter value: ');
            readln(value);
            name := new_name;
            llink := nil; rlink := nil;
            if result = 1 then
              entry^.rlink := new_node
            else
              entry^.llink := new_node
          end;
      end {else};
  end {add_node};

begin
  new(root);
  with root^ do
    begin
      name := '         '; value := 0;
      llink := nil; rlink := nil;
    end;
  repeat
    write('A(dd node F(ind node P(rint node Q(uit');
    read(keyboard,cmd); writeln(cmd);
    case cmd of
      'a': add_node;
      'f': find_node;
      'p': print_node(root);
    end;
  until cmd = 'q';
end {tree_demo}.
```

IV. UCSD INTRINSICS

This chapter contains descriptions of all UCSD intrinsics. The intrinsics are listed in alphabetic order. Each description contains a reference to a related section in chapter 3, which describes the intrinsics in terms of the features they implement (and also presents programming examples).

NOTE - Users unfamiliar with the UCSD intrinsics should peruse chapter 3 before reading this chapter.

With two exceptions, the identifiers chosen to denote UCSD intrinsics are distinct from the standard procedures defined in standard Pascal; RESET and REWRITE are sufficiently altered to warrant inclusion in this section as intrinsics.

NOTE - In order to completely specify the UCSD Pascal intrinsics, this section embellishes Pascal syntax with the metasymbols defined in section 1.2 and two special type identifiers. Metasymbols are used to indicate optional parameters ([<opt-param>]) and sequences of one or more parameters ({<param-seq>}). The special type identifiers indicate relaxed type checking on the corresponding parameter(s). The type UNIV denotes a universal type; formal parameters declared with UNIV accept actual parameters of any type. The type FILEID is compatible with all file types.

NOTE - All extensions described in this chapter are recognized by the compiler and are hence part of the UCSD PASCAL base language. Another class of extensions is available through the use of the library modules described in the Library User's Manual. Routines that allow program chaining, extended directory management, screen control, and other system-oriented functions are documented there, but are listed in Appendix K for convenience.

4.0 ATAN

Syntax:

```
function atan(X: real): real;
```

ATAN is equivalent to the standard procedure ARCTAN.

See section 3.11.6 for more information.

4.1 ATTACH

Syntax:

```
procedure attach(var SEM    : semaphore;
                     VECTOR : integer);
```

ATTACH associates the semaphore variable SEM with the PDQ-3 interrupt vector specified by VECTOR; SEM is signalled whenever the processor receives an interrupt through the interrupt vector.

Interrupt vectors are described in the Hardware User's Manual.

Only one semaphore may be attached to an interrupt vector at one time; attaching a new semaphore to an interrupt vector implicitly detaches the old semaphore. Interrupts through the specified vector must be disabled before an attached semaphore variable is deallocated; otherwise, system crashes may occur.

See sections 3.0.2 and 7.6 for more information.

4.2 BLOCKREAD

Syntax:

```
function blockread(var F       : file;
                   var BUFF    : univ;
                       BLOCKS  : integer
                    [; RELBLOCK : integer]) : integer;
```

BLOCKREAD attempts to read the number of blocks specified by BLOCKS from the file F into the variable BUFF; it returns the number of blocks actually read. If the number of blocks returned is less than the number of blocks requested, BLOCKREAD encountered either the end of the file or an I/O error while reading the data.

I/O checks are automatically generated for BLOCKREAD calls; if I/O checks are suppressed, IORESULT should be used to check the completion status after BLOCKREAD calls.

The optional parameter RELBLOCK applies only when reading from block-structured (disk) files; when specified, it indicates the block in the file where BLOCKREAD starts reading. The starting block is relative to the front of the disk file, with block 0 being the first block in the file.

RELBLOCK is ignored when reading from serial devices.

In the absence of a RELBLOCK parameter, blocks are read from the file consecutively; the first BLOCKREAD after F is opened reads from block 0.

Users must ensure that BUFF specifies a word-aligned address as the starting address of the buffer; this is most important when the starting buffer address is an indexed address in a packed array of char. Users are responsible for not overrunning the buffer specified by BUFF; no range checking is provided.

See section 3.3.3 for more information.

4.3 BLOCKWRITE

Syntax:

```
function blockwrite(var F        : file;
                    var BUFF     : univ;
                        BLOCKS   : integer
                     [; RELBLOCK : integer]) : integer;
```

BLOCKWRITE attempts to write the number of blocks specified by BLOCKS from the variable BUFF to the file F; it returns the number of blocks actually written. If the number of blocks returned is less than the number of blocks requested, BLOCKWRITE encountered an I/O error while writing the data.

I/O checks are automatically generated for BLOCKWRITE calls; if I/O checks are suppressed, IORESULT should be used to check the completion status after BLOCKWRITE calls.

The optional parameter RELBLOCK applies only when writing to block-structured (disk) files; when specified, it indicates the starting block in the file where BLOCKWRITE starts writing to. The starting block is relative to the front of the disk file, with block 0 being the first block in the file.

RELBLOCK is ignored when writing to serial devices.

In the absence of a RELBLOCK parameter, blocks are written to the file consecutively; the first BLOCKWRITE after F is opened writes to block 0.

Users must ensure that BUFF specifies a word-aligned address as the starting address of the buffer; this is most important when the starting buffer address is an index into a packed array of char.

See section 3.3.3 for more information.

4.4 CLOSE

Syntax:

```
procedure close(var F : fileid [; OPTION]);
```

CLOSE sets the file state of the (arbitrarily typed) file F to "closed". Disk file options include "LOCK", "NORMAL", "PURGE", and "CRUNCH"; these determine the final state of the associated disk file. (All options except PURGE are ignored when the external file is not a disk file.)

NORMAL preserves modifications to files opened with RESET; however, if the file was extended, the extension is deleted. If the file was modified, the file date attribute is assigned the current system date. Temporary files created with REWRITE are deleted. NORMAL is the default option.

LOCK preserves files opened with RESET. If the file was extended, the extension is saved. If the file was modified, the file date attribute is assigned the current system date. Temporary files created with REWRITE become permanent files. Note that if the temporary file's name matches an existing file's name, the existing file is deleted when the temporary file becomes permanent.

PURGE deletes the specified disk file. If the external file is an entire volume, the volume is taken off line; serial volumes cannot be opened for subsequent I/O operations until the system is reinitialized.

CRUNCH is equivalent to LOCK except that the file is truncated by designating the file window position as the end of the file. (The position of the file window is determined by the last file operation.) All data between the file window and the original end of the file is deleted.

When a structured file is closed, the contents of the file window become undefined. Closing a non-open file causes an I/O error. At procedure termination, a NORMAL close is performed on all open files declared in that procedure.

See section 3.3.0 for more information.

4.5 CONCAT

Syntax:

```
function concat(S : string
            {; S : string}) : string;
```

CONCAT returns a string containing the concatentation of the string values of its arguments. Note that CONCAT accepts one or more string parameters.

NOTE - The length of the string result is not allowed to exceed 255 characters.

See section 3.4 for more information.

4.6 COPY

Syntax:

```
function copy(SOURCE : string;
              INDEX  : integer;
              SIZE   : integer) : string;
```

COPY returns a string containing SIZE characters copied from SOURCE, starting at the INDEXth character position in SOURCE.

NOTE - COPY returns an empty string if the string specified by SIZE and INDEX exceeds the length of the string in SOURCE.

See section 3.4 for more information.

4.7 DELETE

Syntax:

```
procedure delete(var S     : string;
                     INDEX : integer;
                     SIZE  : integer);
```

DELETE removes SIZE characters from the string in S, starting at the INDEXth character position in S.

NOTE - DELETE leaves S unaltered if the string specified by SIZE and INDEX exceeds the length of the string in S.

See section 3.4 for more information.

4.8 EXIT

Syntax:

```
procedure exit(<routine>);
```

EXIT causes program execution to continue at the end of the block associated with <routine>; acceptable arguments are procedure or function identifiers, program names, or the reserved word PROGRAM. If EXIT specifies a recursively invoked routine, only the most recent invocation is terminated.

NOTE - Calls to EXIT are not legal inside of unit initialization and termination sections. EXIT(PROGRAM) should not be performed from implementation procedures during the execution of either of these sections.

NOTE - Exiting through an uncalled procedure results in execution error 3 ("Exit from uncalled procedure").

WARNING - When EXIT is used to terminate a function, the function result must have been assigned beforehand; otherwise, the function returns an undefined value when exited.

See section 3.11.7 for more information.

4.9 FILLCHAR

Syntax:

```
procedure fillchar(var BUFFER : univ;
                       BYTES  : integer;
                       CH     : character);
```

FILLCHAR initializes BYTES bytes in memory with the value in CH; the starting address is specified by BUFFER.

NOTE - The SIZEOF intrinsic is often used in FILLCHAR to specify the size of BUFFER.

WARNING - Negative values in BYTES are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING - FILLCHAR offers no range or type checking.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.8 for more information.

4.10 GOTOXY

Syntax:

```
procedure gotoxy(X : integer;
                 Y : integer);
```

GOTOXY moves the cursor to the position specified by its arguments. X determines the horizontal displacement, while Y determines the vertical displacement; the upper left corner of the screen is defined to be (0,0). If parameter values exceed the maximum values defined for the system terminal, they are truncated to the maximum values.

NOTE - GOTOXY is a terminal-dependent procedure; it usually requires redefinition when a new terminal is incorporated into the system. See section 8.3 in the System User's Manual for details on redefining GOTOXY.

See section 3.11.3 for more information.

4.11 HALT

Syntax:

```
procedure halt;
```

HALT suspends the current program and prints a halt prompt.

NOTE - The behavior of the HALT intrinsic may be changed by replacing the operating system HALTUNIT. See section 6.4 for details.

NOTE - Programs may be terminated with the EXIT intrinsic.

See section 3.11.10 for more information.

4.12 IDSEARCH

Syntax:

```
procedure idsearch(INX    : integer;
                   var BUFFER : univ);
```

A description of IDSEARCH is beyond the scope of this document; see the Architecture Guide for details.

See section 3.11.11 for more information.

4.13 INSERT

Syntax:

```
procedure insert(SUBSTRING : string;
             var S         : string;
                 INDEX     : integer);
```

INSERT stuffs the string in SUBSTRING into the string contained in S at the INDEXth character position in S.

NOTE - INSERT leaves S unaltered if the position specified by INDEX exceeds the length of the string in S by more than one character.

See section 3.4 for more information.

4.14 IORESULT

Syntax:

```
function ioresult : integer;
```

IORESULT returns an integer value indicating the result of the last I/O operation performed by the current task (section 3.0.0).

NOTE - I/O results are updated after every I/O operation; therefore, an I/O result value must be saved in a variable if subsequent I/O operations occur before it can be manipulated.

NOTE - A table of standard I/O error numbers and their corresponding messages is displayed in Appendix A. Conditions causing bad I/O results are listed in Appendix C.

NOTE - Nonstandard I/O error numbers may be defined by the user. See the EXCEPINFO and PROGOPS unit documentation in the Library User's Manual for details.

See section 3.11.8 for more information.

4.15 LENGTH

Syntax:

```
function length(S : string) : integer;
```

LENGTH returns the dynamic length of the string contained in S. See section 3.4 for more information.

4.16 MARK

Syntax:

```
procedure mark(var MARKP : ^integer);
```

MARK opens a heap for dynamically allocated variables. Subsequent calls to NEW allocate dynamic variables only in the new heap. The heap is identified by the value assigned to MARKP. The RELEASE intrinsic is used to deallocate all dynamic variables in a heap opened by MARK.

NOTE - New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

WARNING - Pointers passed to MARK must only be used as arguments to subsequent calls to RELEASE. Careless use of MARK and RELEASE leads to "dangling references" (i.e. pointers to deallocated dynamic variables, which may or may not be overwritten by subsequent system actions).

NOTE - When the $H+ compile option (section 5.0.6) is in effect, MARK allocates a three word mark record in addition to allocating a new heap.

See section 3.5 for more information.

4.17 MEMAVAIL

Syntax:

```
function memavail : integer;
```

MEMAVAIL returns the number of unused words in memory. The integer result contains an unsigned value; if large, it may be interpreted as a negative value unless specifically treated as an unsigned integer result (see section 7.2).

NOTE - The MEMAVAIL intrinsic returns the number of words between the system stack and heap. It should be used in conjunction with the II.0 heap intrinsics; VARAVAIL (section 3.5.1) is intended for use with the IV.0 heap intrinsics.

See section 3.11.9 for more information.

4.18 MEMLOCK

Syntax:

```
procedure memlock(SEGLIST : string);
```

MEMLOCK loads the code of each segment named in the SEGLIST. A MEMLOCKed segment remains in memory until it is named in a call to MEMSWAP (section 4.19). The SEGLIST consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and the units it uses, or in the operating system. Unrecognized segment names are ignored.

NOTE - The MEMLOCK intrinsic may be used only when the $H+ compile option (section 5.0.6) is in effect.

WARNING - MEMLOCKed segment code resides in the current heap. Indiscreet calls to MEMLOCK may render the heap incapable of providing large continuous blocks of memory.

See section 3.2.0 for more information.

4.19 MEMSWAP

Syntax:

```
procedure memswap(SEGLIST : string);
```

MEMSWAP unloads the code of each MEMLOCKed segment (section 4.18) named in the SEGLIST. The SEGLIST consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and its used units, or in the operating system. Unrecognized segment names are ignored.

NOTE - The MEMSWAP intrinsic may be used only when the $H+ compile option (section 5.0.6) is in effect.

NOTE - MEMSWAP operates only on MEMLOCKed segments. A MEMLOCKed code segment is not unloaded until a MEMSWAP call has been performed for each MEMLOCK call naming that segment and all active calls are complete.

See section 3.2.0 for more information.

4.20 MOVELEFT

Syntax:

```
procedure moveleft(var SOURCE      : univ;
                       DESTINATION : univ;
                       BYTES       : integer);
```

MOVELEFT moves BYTES bytes of data from the buffer addressed by SOURCE to the buffer addressed by DESTINATION. The data is moved one byte at a time, starting with the bytes addressed by SOURCE and DESTINATION, and moving successively higher-addressed bytes until BYTES bytes have been moved.

WARNING - Negative values in BYTES are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING - MOVELEFT does not perform type or range checking on its parameters.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.8 for more information.

4.21 MOVERIGHT

Syntax:

```
procedure moveright(var SOURCE      : univ;
                        DESTINATION : univ;
                        BYTES       : integer);
```

MOVERIGHT moves BYTES bytes of data from the buffer addressed by SOURCE to the buffer addressed by DESTINATION. The data is moved one byte at a time, starting with the bytes addressed by the expressions (SOURCE + BYTES - 1) and (DESTINATION + BYTES - 1), and moving successively lower-addressed bytes until BYTES bytes have been moved.

WARNING - Negative values in BYTES are treated as zero byte counts; hence, large unsigned values may not work as expected.

WARNING - MOVERIGHT does not perform type or range checking on its parameters.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.8 for more information.

4.22 OPENNEW

Syntax:

```
procedure opennew(var F        : fileid;
                      FILENAME : string);
```

OPENNEW is equivalent to the REWRITE intrinsic. Section 4.29 describes REWRITE.

4.23 OPENOLD

Syntax:

```
procedure openold(var F        : fileid
                  [; FILENAME : string]);
```

OPENOLD is equivalent to the RESET intrinsic. Section 4.28 describes RESET.

4.24 PMACHINE

Syntax:

```
procedure pmachine(<item> {,<item>});
```

PMACHINE generates inline machine code corresponding to the items specified in the parameter list. See the Architecture Guide for a description of the PDQ-3 instruction set.

```
<item> ::= <code>              |
           <expression>        |
           <address-reference>
```

<code> ::= A constant value or constant identifier denoting an integer between 0 and 255; PMACHINE emits a single byte with the specified value. Values outside this range have only their least significant byte emitted. Code bytes represent either P-code instructions or instruction operands.

<expression> ::= (<Pascal expression>)

<Pascal expression> ::= An expression (e.g. the right-hand side of an assignment statement). PMACHINE generates code which evaluates the expression, leaving the result on the stack.

<address-reference> ::= ^<variable reference>

<variable reference> ::= A referenced variable (e.g. the left-hand side of an assignment statement). PMACHINE generates code which evaluates the variable reference, leaving the address on the stack.

DISCLAIMER - Incorrect use of PMACHINE invalidates all warranties on PDQ-3 system software.

See section 3.10 for more information.

4.25 POS

Syntax:

```
function pos(SUBSTRING : string;
             S         : string) : integer;
```

POS searches S for an occurrence of SUBSTRING; it returns an index indicating the start of the matched substring. If S contains multiple occurrences of SUBSTRING, POS locates the first occurrence. If S contains no occurrences of SUBSTRING, POS returns 0.

See section 3.4 for more information.

4.26 PWROFTEN

Syntax:

```
function pwroften(EXPONENT : integer) : real;
```

PWROFTEN returns the floating point representation of ten raised to the EXPONENT'th power. EXPONENT must be in the range 0..38; negative arguments always return 1, while arguments greater than 38 cause execution error 12 (" Floating point error ").

See section 3.11.5 for more information.

4.27 RELEASE

Syntax:

```
procedure release(var MARKP : ^integer);
```

RELEASE deallocates all dynamic variables in the heap associated with MARKP.

NOTE - New heaps are allocated within the current heap; thus, heaps are nested. Deallocating a given heap results in the deallocation of all subsequently opened heaps.

WARNING - Pointers passed to RELEASE must be initialized by a previous call to the MARK intrinsic. Careless use of MARK and RELEASE leads to "dangling references" (i.e. pointers to deallocated dynamic variables, which may or may not be overwritten by subsequent system actions).

NOTE - MEMLOCKed segment code (section 3.2.0) is maintained in the current heap. RELEASEing the current heap does not deallocate the memory occupied by the segment code.

NOTE - When the $H+ compile option (section 5.0.6) is in effect, RELEASE deallocates a three word mark record in addition to deallocating the specified heap.

See section 3.5 for more information.

4.28 RESET

Syntax:

```
procedure reset(var F        : fileid
                [; FILENAME : string]);
```

RESET opens the external file named in FILENAME, and prepares the file variable F for subsequent operations on the external file. If F does not denote an interactive file, RESET performs an implicit GET; this is consistent with the standard procedure RESET in standard Pascal.

RESET is used to open existing files for reading and/or writing.

RESET generates an I/O error in the following cases:

-- The file variable F is already open.

-- FILENAME specifies a nonexistent external file.

-- FILENAME specifies a write-only device.

RESET without the file name parameter rewinds the file window to the beginning of the (open) file.

External files opened with RESET may be closed with the CLOSE intrinsic.

See section 3.3.0 for more information.

4.29 REWRITE

Syntax:

```
procedure rewrite(var F        : fileid;
                      FILENAME : string);
```

REWRITE creates a temporary external file named FILENAME, and prepares the file variable F for subsequent operations on the external file.

REWRITE is used to open new files for writing.

REWRITE generates an I/O error in the following cases:

-- The file variable F is already open.

-- Insufficient room on disk to create the file.

External files opened with REWRITE may be saved with the CLOSE intrinsic.

See section 3.3.0 for more information.

4.30 RMEMAVAIL

Syntax:

```
function rmemavail : real;
```

RMEMAVAIL returns the number of unused words in memory.

NOTE - The RMEMAVAIL intrinsic returns the number of words between the system stack and heap. It should be used in conjunction with the II.0 heap intrinsics; VARAVAIL (section 3.5.1) is intended for use with the IV.0 heap intrinsics.

See section 3.11.9 for more information.

4.31 SCAN

Syntax:

```
function scan(BYTES : integer;
              <partial expression>;
          var BUFF  : univ) : integer;
```

Starting at the address specified by the variable BUFF, SCAN examines successive bytes in memory until one of the following conditions becomes true:

-- The current byte contains a value which satisfies the partial expression.

-- BYTES bytes have been examined without finding a value that satisfies the partial expression.

Partial expressions are incomplete Boolean expressions; the left-hand operand is defined to be the current byte being examined by SCAN. A partial expression is satisfied when it evaluates to true. The partial expression is evaluated for each byte examined by SCAN; if it becomes true, SCAN returns immediately.

<partial expression> ::= = | <> <character expression>

<character expression> ::= character variable or constant

SCAN returns the number of bytes examined. If the byte pointed at by the starting address contains a value satisfying the partial expression, SCAN returns 0. If the value in BYTES is negative, SCAN scans backwards (towards lower addresses) from its starting address searching for the target byte, and returns a negative number (in the range BYTES..0) whose magnitude indicates the number of bytes examined.

WARNING - Negative values in BYTES cause backwards scanning; hence, large unsigned values may not work as expected.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.8 for more information.

4.32 SEEK

Syntax:

```
procedure seek(var F      : fileid;
                   RECNUM : integer);
```

SEEK moves the file window in F so that a subsequent GET or PUT accesses the RECNUM'th record in the file. F must be a structured disk file (i.e. any standard Pascal file except text). The first record in a file is record 0. The standard procedure EOF is used to detect seeks off the end of the file. Though SEEK itself always sets EOF to false, a subsequent GET sets EOF to true if the new file position is at (or past) the end of the file.

WARNING - The result of SEEK is undefined if the file position is moved more than one record past the final record in the file. If SEEK moves to the first empty record in the file, a subsequent PUT extends the file in a normal fashion; however, if records are written at file positions more than one record past the end of the file, the file itself becomes undefined (resulting in subsequent program errors). Note that EOF alone is insufficient to distinguish these cases.

See section 3.3.4 for more information.

4.33 SEMINIT

Syntax:

```
procedure seminit(var SEM   : semaphore;
                      COUNT : integer);
```

SEMINIT initializes SEM with the value COUNT.

WARNING - Calling SIGNAL or WAIT with an uninitialized semaphore variable may crash the system. Calling SEMINIT with a semaphore holding suspended tasks causes the system to lose the tasks.

See section 3.0.1 for more information.

4.34 SIGNAL

Syntax:

```
procedure signal(var SEM : semaphore);
```

If no tasks are waiting on SEM, SIGNAL increments the semaphore's count; otherwise, SIGNAL selects the highest priority waiting task and inserts it in the ready queue.

See section 3.0.1 for more information.

4.35 SIZEOF

Syntax:

```
function sizeof(<identifier>) : integer;
```

SIZEOF returns the number of bytes of memory allocated for the data object denoted by <identifier>. <identifier> may be either a variable or type identifier. SIZEOF is used in conjunction with the intrinsics MOVELEFT, MOVERIGHT, and FILLCHAR.

NOTE - SIZEOF is evaluated by the compiler; it replaces each call with a constant containing the result. Thus, SIZEOF cannot return the size of runtime variable references.

NOTE - If a record contains variant fields, SIZEOF uses the longest variant when determining its size.

See section 3.8 for more information.

4.36 START

Syntax:

```
procedure start(<process call>
         [; var PID       : processid
             [; STACKSIZE : integer
             [; PRIORITY  : integer ]]]);
```

START initiates tasks.

The main parameter to START is a process call; it resembles a procedure call, and may contain parameters passed to the task.

The remaining parameters define various task attributes. PID is assigned a value which uniquely identifies the new task. STACKSIZE indicates the number of words of memory to be allocated for a stack space; if absent, START uses 200 as a default stack size. PRIORITY indicates the task priority to be assigned the new task; if it is not in the range 0..255, an execution error occurs. The default priority is 128.

See section 3.0.0 for more information.

4.37 STR

Syntax:

```
procedure str(L : integer[36];
          var S : string);
```

STR converts the value in L into a string in S; it is used to format long integer values for output. If the value in L is negative, the first character placed in S is "-".

See section 3.6 for more information.

4.38 TIME

Syntax:

```
procedure time(var HIWORD : integer;
               var LOWORD : integer);
```

TIME returns the current value in the system clock in the integer pair HIWORD and LOWORD. The system clock is an unsigned 32-bit integer incremented every 60'th of a second. HIWORD contains the most significant word.

NOTE - HIWORD and LOWORD contain unsigned values; they may be treated as negative numbers by some integer operations unless specifically treated as unsigned integers (section 7.2).

See section 3.11.4 for more information.

4.39 TREESEARCH

Syntax:

```
type alpha = packed array [1..8] of char;
     nodeptr = ^node;
     node = record
              name: alpha;
              left_link, right_link: nodeptr;
              { user-defined record fields }
            end {node};

function treesearch(ROOT : nodeptr;
                    var NODE : nodeptr;
                        NAME : alpha) : integer;
```

TREESEARCH manages binary trees ordered by the contents of an 8 character field. TREESEARCH searches the tree rooted at ROOT for a record whose name field matches NAME; on return, NODE contains a pointer to the last record examined, and the function result indicates the results of the search.

If a record in the tree matches the array argument, TREESEARCH returns 0 as a function result; NODE points to the matching record. If no record in the tree matches the array argument, TREESEARCH returns either 1 or -1; NODE is set to the last node searched. 1 indicates that NAME is greater than the name in the record pointed at by NODE (and would be inserted on its right link); -1 indicates that the argument is less than NODE (and would be inserted on its left link).

NOTE - The PDQ-3 TREESEARCH intrinsic constructs trees so that right (post order) traversals visit the records in lexicographical order of their name fields. This ordering may differ on other UCSD Pascal implementations.

See section 3.11.11 for more information.

4.40 UNITBUSY

Syntax:

```
function unitbusy(UNITNUM : integer) : boolean;
```

UNITBUSY indicates whether the specified device is waiting for an I/O operation to finish.

See section 3.9 and Appendix D for more information.

4.41 UNITCLEAR

Syntax:

```
procedure unitclear(UNITNUM : integer);
```

UNITCLEAR cancels any I/O operations occuring on the specified device, and resets the unit to its initial (i.e. power-up) state.

See section 3.9 and Appendix D for more information.

4.42 UNITREAD

Syntax:

```
procedure unitread(UNITNUM   : integer;
                   var BUFF      : univ;
                       BYTES     : integer
                    [; BLOCKNUM  : integer
                    [; CONTROL   : integer]]);
```

UNITREAD reads BYTES bytes from the device UNITNUM into the variable BUFF. BLOCKNUM is applicable only when reading from block-structured units; it specifies the starting block of the transfer. (Block numbers start at 0.) CONTROL is treated as a bit array; certain bits in the control word are defined to select various I/O options (depending on the unit specified - see Appendix D for details).

The BLOCKNUM parameter is ignored when UNITNUM specifies a serial unit. Though it is not specified in the syntax definition above, UNITREAD accepts a CONTROL parameter in the absence of a BLOCKNUM parameter. The form is:

```
UNITREAD(<unit>,<buffer>,<length>,,<control>)
```

NOTE - BUFF is constrained to start on a word address when the specified unit is block-structured; reading into an odd byte address causes I/O error 18 (illegal buffer address).

WARNING - UNITREAD performs no type or range checks on its parameters.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.9 and Appendix D for more information.

4.43 UNITSTATUS

Syntax:

```
procedure unitstatus(UNITNUM : integer;
                     var STATREC : univ;
                         DIR     : integer);
```

UNITSTATUS returns the status of the device UNITNUM in the STATREC record. The format of the STATREC record depends on the type of device being polled. It may be either a serial device record, block-structured device record, or a system clock record. The record should occupy at least 30 words to allow for future expansion.

The DIR parameter is not used and should be passed as zero.

See section 3.9 and Appendix D for more information.

4.44 UNITWAIT

Syntax:

```
procedure unitwait(UNITNUM : integer);
```

UNITWAIT waits for the device specified by UNITNUM to finish its current I/O operation.

NOTE - UNITWAIT is not implemented on the PDQ-3; if called, it returns immediately.

See section 3.9 and Appendix D for more information.

4.45 UNITWRITE

Syntax:

```
procedure unitwrite(UNITNUM  : integer;
                    var BUFF     : univ;
                        BYTES    : integer
                     [; BLOCKNUM : integer
                     [; CONTROL  : integer]]);
```

UNITWRITE writes BYTES bytes to the device UNITNUM from the variable BUFF. BLOCKNUM is applicable only when writing to block-structured units; it specifies the starting block for the transfer. (Block numbers start at 0.) CONTROL is treated as a bit array; certain bits in the control word are defined to select various I/O options (depending on the unit specified - see Appendix D for details).

The BLOCKNUM parameter is ignored when UNITNUM specifies a serial unit. Though it is not specified in the syntax definition above, UNITWRITE accepts a CONTROL parameter in the absence of a BLOCKNUM parameter. The form is:

UNITWRITE(<unit>,<buffer>,<length>,,<control>)

NOTE - BUFF is constrained to start on a word address when the specified unit is block-structured; writing from an odd byte address causes I/O error 18 (illegal buffer address).

WARNING - UNITWRITE performs no type or range checks on its parameters.

WARNING - Array indices on the PDQ-3 are treated as signed integers. In the specification of the starting buffer address, use of an array index whose value is less than the buffer's declared lower bound may yield unexpected or fatal results.

See section 3.9 and Appendix D for more information.

4.46 VARAVAIL

Syntax:

```
function varavail(SEGLIST : string) : integer;
```

The VARAVAIL function returns the size of the largest continuous free space in memory assuming that all segments named in the SEGLIST are resident. The SEGLIST consists of a list of segment names separated by commas; spaces are ignored. It may contain any segment name declared either in the program and the units it uses, or in the operating system. Unrecognized segment names are ignored. In calculating VARAVAIL, it is assumed that all currently nonresident segments named in the SEGLIST would be loaded onto the system stack rather than be MEMLOCKed.

NOTE - The VARAVAIL intrinsic may be used only when the $H+ compile option (section 5.0.6) is in effect.

See section 3.5 for more information.

4.47 VARDISPOSE

Syntax:

```
procedure vardispose(var P         : ^univ;
                         WORDCOUNT : integer);
```

The VARDISPOSE procedure deallocates the buffer referenced by P. The buffer size is specified by the unsigned integer parameter, WORDCOUNT. P is returned containing nil.

WARNING - Deallocating a buffer of a different size than was originally allocated could lead to a system crash.

NOTE - The VARDISPOSE intrinsic may be used only when the $H+ compile option (section 5.0.6) is in effect.

NOTE - Attempts to deallocate a one word buffer actually deallocate two words.

See section 3.5 for more information.

4.48 VARNEW

Syntax:

```
function varnew(var P         : ^univ;
                    WORDCOUNT : integer) : integer;
```

The VARNEW function attempts to allocate a buffer of WORDCOUNT words on the heap and return P as a pointer to the buffer. If there is enough contiguous free memory for the buffer, it is allocated and the value of VARNEW is returned equal to WORDCOUNT; otherwise VARNEW is returned zero.

NOTE - The VARNEW intrinsic may be used only when the $H+ compile option (section 5.0.6) is in effect.

NOTE - Attempts to allocate a one word buffer actually allocate two words.

See section 3.5 for more information.

4.49 WAIT

Syntax:

```
procedure wait(var SEM : semaphore);
```

If the semaphore count of SEM is greater than zero, it is decremented, and the current task continues to execute; otherwise, the current task is suspended, and waits for a SIGNAL on SEM.

See section 3.0.1 for more information.

V. COMPILE OPTIONS

This chapter describes the compile options in UCSD Pascal. Compile options affect both compiler operation and the execution characteristics of code produced by the compiler. Compile options are controlled by directives embedded in the text of source programs; these are processed by the compiler as they are encountered in the program. Section 5.0 describes the use of compile options, and provides a detailed description of each compile option. Section 5.1 summarizes the compile options.

5.0 Options

Compile options appear as directives in a source program; these directives are called pseudo-comments. A pseudo-comment is a comment (as defined in UCSD Pascal) which contains a "$" character immediately following the left-hand comment delimiter. Following the "$" is a list of one or more compile options; multiple options are delimited by commas. Each compile option consists of a single alphabetic character (upper or lower case) denoting a specific option, possibly followed by an argument.

An option which may accept "+", "-" or "^" is known as a switch option. Compile options which accept alpha-numeric arguments are known as string options. These arguments may be integers, lists of UCSD Pascal identifiers, file names, or simply text strings. String options are terminated by the right-hand comment delimiter. Note that a single pseudo-comment cannot contain more than one string option.

NOTE - String arguments may not contain the character "*" when the right-hand comment delimiter is "*)", and may not contain the character "}" when the right-hand delimiter is "}".

WARNING - Generally, invalid compile options are ignored by the compiler; the pseudo-comment is treated as a normal comment. One exception to this is when a string option contains an argument violating the restriction described above (in the NOTE); the string argument is erroneously truncated. If the illegal character is the first character in the string, and the option character happens to be used for both string and switch options, the string option is incorrectly treated as a switch option - beware!

Examples of compile options:

```
{$I+}
(*$I yeenly.text *)
{$L+,U-,S+}
(*$L+,U-,V 34 *)
(*$B CondIdent- *)
{$P}
```

Example of string option incorrectly treated as switch option:

```
(*$I*dysfunc.text *)
```

All switch options accept the "+" and "-" switches as arguments. The "+" switch enables the option (on); "-" disables the option (off). Values for the D, I, J and R options may be stacked up to 16 levels deep. Thus, when a directive is set (e.g. R- or I+), the new value is pushed onto the top of its stack. An option's current value is the value on the top of its stack. The "^" switch pops the option's stack, causing the option to be restored to its prior value. Stacked options are useful when a short section of code requires the assertion of an option value, but the option value of the enclosing program is unknown or subject to change. A directive pair of the form: {$I-} ... {$I^} asserts the desired compile option value without affecting the option value in the enclosing program.

Example of stacked compile option values:

```
program stack;
var i: integer;
begin
  repeat
    {$I-}
    readln(i);
    {$I^}
  until ioresult = 0;
  writeln ('value is: ',i);
end {stack}.
```

Syntax for compile options:

```
<pseudo-comment> ::=  <L-delim>$<options><R-delim>

<L-delim> ::=  UCSD Pascal comment delimiter: "{" or "(*"

<R-delim> ::=  UCSD Pascal comment delimiter: "}" or "*)"

<options> ::=  <string-option>  |  <option-list>

<option-list> ::=  <switch-option-list>[,<string-option>]

<switch-option-list> ::=  <switch-option>{,<switch-option>}

<switch-option> ::=  <switch-directive><switch>  |
                     <button-directive>

<string-option> ::=  <string-directive><string>  |
                     <number-directive><integer>  |
                     <idlist-directive><idlist>  |
                     <cond-directive><flag>[<switch>]

<string> ::=  any sequence of characters other than
              "*" or "}" (see previous NOTE)

<idlist> ::=  <identifier>{,<identifier>}

<flag> ::=  A compile flag identifier which follows the same
            rules as a UCSD Pascal identifier.

<switch> ::=  "+"  |  "-"  |  "^"

<switch-directive> ::=  H  |  I  |  J  |  L  |  O  |
                        Q  |  R  |  S  |  U

<button-directive> ::=  P

<string-directive> ::=  C  |  I  |  L

<number-directive> ::=  V

<idlist-directive> ::=  N  |  R

<cond-directive> ::=  B  |  D  |  E
```

Switch directive	Compile option
H	Heap intrinsics (section 5.0.6)
I	I/O checking (section 5.0.4)
J	Boolean NOT evaluation (section 5.0.13)
L	Listing (section 5.0.0)
O	Operating system (section 5.0.12)
Q	Console display (section 5.0.8)
R	Range checking (section 5.0.5)
S	Swapping compiler (section 5.0.2)
U	User lex level (section 5.0.11)

Button directive	Compile option
P	Page eject during listing (section 5.0.0)

String directive	Compile option
C	Copyright notice (section 5.0.7)
I	Include file (section 5.0.1)
L	Compiled listing (section 5.0.0)

Number directive	Compile option
V	Version control (section 5.0.10)

Id List directive	Compile option
N	Nonresident unit (section 5.0.9)
R	Resident segment (section 5.0.9)

Conditional directive	Compile option
B	Begin of section (section 5.0.3)
D	Identifier declaration (section 5.0.3)
E	End of section (section 5.0.3)

5.0.0 Compiled Listings

Compiled listings serve two purposes in UCSD Pascal. First, they provide a complete listing of the program source in a single text file; this is useful when the program source itself resides in a number of text files which are included during compilation. Second (and more importantly), they serve as a debugging tool; a compiled

listing contains information used to locate the Pascal source statement responsible for causing an execution error (see section 7.8 for details).

Example of a compiled listing:

```
  1  128    1:D     1 {$L look.text}
  2  128    1:D     1 program example;
  3  128    1:D     1 var i,j,k: integer;
  4  128    1:D     4     s1,s2: string;
  5  128    1:D    86     r: real;
  6  128    1:D    88
  7  129    1:D     1   segment procedure stuff;
  8  129    1:D     1   var l1,l2: integer;
  9  129    1:D     3
 10  129    2:D     1     procedure local;
 11  129    2:0     0     begin
 12  129    2:1     0       writeln('in stuff');
 13  129    2:1    21       exit(program);
 14  129    2:0    27     end;
 15  129    2:0    30
 16  129    1:0     0   begin
 17  129    1:1     0     if i = 45 then local;
 18  129    1:0     8   end {stuff};
 19  129    1:0    10
 20  130    1:D     1   segment function max(a,b:integer):integer;
 21  130    1:0     0   begin
 22  130    1:1     0     if a < b then max := b
 23  130    1:1     6     else max := a;
 24  130    1:0    14   end {max};
 25  130    1:0    16
 26  128    1:0     0 begin
 27  128    1:1     0   i := 45;
 28  128    1:1     8   r := 4.4E1;
 29  128    1:1    15   if max(i,trunc(r)) = i then i := 45;
 30  128    1:1    41   stuff;
 31  128    1:0    52 end {example}.
```

The first column displays the line number (in the listing) of the current source line. The second column displays the segment number of the code segment containing the code corresponding to the source line.

The third column displays two numbers separated by a colon. The left-hand number displays the procedure number of the procedure which contains the code corresponding to the source line. The right-hand number displays the current nesting level of the source statement. The nesting level is determined by the number of unterminated BEGIN - END pairs enclosing the source statement. Note that the nesting level is replaced by the letter "D" when the corresponding source line contains declarations rather than statements.

The fourth column displays the code offset of the corresponding source statement, or the data offset of the corresponding declara-

tions (indicated by the presence of a "D" in the previous column). Code offsets are byte offsets from the beginning of the current procedure. Data offsets are word offsets into the data segment of the enclosing block. In both cases, the value displayed represents the offset before the code or data associated with the current line is compiled.

NOTE - Odd messages occasionally appear in a compiled listing:

Code range: <x> - <y> moved <n> bytes

The compiler emits these messages when it is forced to change the position of previously generated code; this spoils the offsets displayed on previous lines in the compiled listing. <x> and <y> are code offsets within the procedure, and <n> is an integer value. The messages indicate that the compiler has moved the code between <x> and <y> down by <n> bytes; code offsets between <x> and <y> are no longer listed correctly. Offset values can be corrected by adding <n> to the displayed offsets.

Compiled listings are generated when the List option is enabled. The List option is controlled by the pseudo-comment directive "L", which is used as both a switch option and a string option.

The default setting of the List option is off. "L+" enables the List option, and produces a compiled listing written to the disk file *SYSTEM.LST.TEXT. The compiled listing may be written to a different file name by using "L" as a string option; this enables the List option and specifies a user-defined list file name.

Portions of a program may be listed by selectively enabling ("L+") and disabling ("L-") the List option.

List files are saved whether or not the compiler flags syntax errors; if errors occur, error messages are embedded in the list file.

NOTE - "L" may be used only once during the course of a compilation as a string option.

Page breaks may be placed in a compiled listing by using the Page option. "P" emits a single page break.

Example of listing directives:

```
{$L mylist.text }
{$L+}
{$L-}
{$P}
```

5.0.1 Include Files

Include files allow the source comprising a large program to be distributed among a number of relatively small and easy-to-manage

text files. The compiler accepts only one source file as an input file; however, the input file may contain an include directive for each include file required. When the compiler finds an include directive, it includes the contents of the specified text file as program source; when the end of the include file is reached, the compiler returns to the source following the original include directive.

NOTE - Include files may be nested up to three deep. Certain restrictions on the use of include files arise when compiling units (see section 3.2 for details). Include files adversely affect compiler operation in some circumstances - see section 5.2.1 in the System User's Manual for details.

Include files are specified by the pseudo-comment directive "I" used as a string option. The string contains an include file name, which does not require a file suffix. If the file cannot be opened as specified, the compiler appends ".TEXT" to the file name and attempts to reopen it; if this also fails, syntax error 403 occurs.

Example of include directives:

```
{$I foon.text}
{$I 3.2:globals }
```

See section 3.11.1 for an example of include files.

5.0.2 Swapping Compiler

The compiler may assume an alternate mode of operation for compiling large programs. The compiler normally operates as a single memory-resident segment; this mode is used to compile programs that don't tax the system's compile-time memory resources. The Swapping option transforms the compiler into two separate disk-resident segments, thus providing extra memory space for the compilation of large programs.

Swapping mode saves about four thousand words of memory during compilation, but halves the compile speed when the compiler code file resides on a floppy.

The Swapping option is controlled by the pseudo-comment directive "S" used as a switch option. The default setting of the Swapping option is off. "S+" enables swapping.

NOTE - Swapping option directives must appear before the program heading; unlike other options, swapping cannot be selectively turned on and off during compilation.

Example of swapping directive:

```
{$S+}
```

5.0.3 Conditional Compilation

Conditional compilation allows the selective inclusion of sections of source text during compilation. Conditional compilation is controlled by the "B", "D" and "E" pseudo-comment directives and the boolean values associated with compile-time identifiers known as compile flags.

Compile flags are declared by using the Declare option before the program heading. The pseudo-comment directive "D" is followed by a unique flag name which must conform to the same syntax as a UCSD Pascal identifier (see section 3.11.0). The initial value of the flag is set by a trailing "+" (indicating TRUE) or "-" (indicating FALSE). In the absence of a trailing "+" or "-", the flag value defaults to TRUE. Compile flag values may be redefined within the program. Attempts to redefine flags not declared before the beginning of the program or unit generate a syntax error.

Examples of compile flag declaration and value assignment:

```
{$D debug}      -- Declare flag "debug" with value TRUE
{$D Z80+}       -- Declare flag "Z80" with value TRUE
{$D listing-}   -- Declare flag "listing" with value FALSE
{$D debug^}     -- Set flag "debug" to its prior value
```

Source code is selectively included in a compilation by using the Begin and End options. These are analogous to BEGIN and END in Pascal. When the compiler scans a "B" pseudo-comment directive which contains a valid compile flag, the value of the flag expression determines whether the source text between the "B" directive and its corresponding "E" directive is to be compiled. If the flag expression evaluates to FALSE, the compiler skips over source text until it encounters an "E" directive containing the flag identifier. If the compile flag in the "B" directive is followed by a "-" switch, the flag expression is equal to the logical negation of the flag identifier value.

NOTE - The "^" switch is ignored if it appears in the flag expression of the "B" directive. All switches are ignored in the "E" directive, but are useful for documentation purposes.

WARNING - Unit interface text is stored in a library without regard for the values of imbedded flag expressions. Thus, conditional compilation directives can be found in imported interface text. All such flags should be defined before the beginning of the host using the unit. For added security, it is recommended that the value of the flag be redefined in the interface text so as to avoid any inconsistency between the unit's actual interface and the interface perceived by the host.

Example of conditional compilation:

```
{$D debug-}      {Declare flag "debug" with value FALSE}
program demo;
begin
  {$B debug}     {The following statement is not compiled}
  writeln ('there is a bug');
  {$E debug}

  {$D debug+}    {Set debug to TRUE}

  {$B debug}     {The following statement is compiled}
  writeln ('now there really is a bug');
  {$E debug}

  {$D debug^}    {Restore debug to FALSE}
end {demo}.
```

5.0.4 I/O Checks

The compiler normally emits I/O checks after every file I/O operation; these checks cause an execution error if the I/O result (section 3.11.8) reveals that an I/O error occurred during the operation.

I/O checks are emitted when the I/O Check option is enabled. The I/O Check option is controlled by the pseudo-comment directive "I" used as a switch option.

The default setting of the I/O Check option is on. "I-" disables the option, and suppresses the generation of I/O check code. I/O checking may be restricted to portions of a program by selectively enabling ("I+") and disabling ("I-") the I/O Check option.

NOTE - Programs compiled with the I/O Check option disabled require explicit I/O checks. Failure to provide these checks in some form leaves a program susceptible to unexpected actions of both a human and mechanical nature.

Example of I/O check directives:

```
{$I+}
{$I-}
{$I^}
```

See section 3.11.8 for more information. An example using I/O check directives appears in section 7.5.

5.0.5 Range Checks

The compiler normally emits range checks before every indexed array reference or subrange assignment. These checks cause an execution

error if an array is indexed outside of its declared bounds, or if a subrange variable is assigned a value outside of its declared range.

Range checks are emitted when the Range Check option is enabled. The Range Check option is controlled by the pseudo-comment directive "R" used as a switch option.

The default setting of the Range Check option is on. "R-" disables the option, and suppresses the generation of range check code. Range checking may be restricted to portions of a program by selectively enabling ("R+") and disabling ("R-") the Range Check option.

NOTE - Programs compiled with the Range Check option disabled are smaller and faster than their cautious counterparts; however, they must be correct at the outset, for undetected range errors can propagate various and sundry species of nasty and elusive bugs. Proofs of program correctness are left to the user.

NOTE - The Range Check option only affects the generation of execution error 1 ("Value range error") in the cases mentioned above. The I/O Check option affects the generation of execution errors due to I/O faults. Suppression of other execution errors requires the modification of the system exception handler. See section 6.3 for further details.

NOTE - If the second argument to the MOD operator is negative, a nonsuppressable value range execution error occurs.

Example of range check directives:

```
{$R+}
{$R-}
{$R^}
```

An example using range check directives appears in section 7.3.

5.0.6 Heap Intrinsics

UCSD Pascal provides two sets of intrinsics for dynamic variable allocation: the II.0 and IV.0 heap intrinsics. A particular set of heap intrinsics may be selected using the Heap compile option. This option is controlled by the pseudo-comment directive "H" used as a switch option. The II.0 heap intrinsics may be used when the Heap compile option is disabled ("H-"); this is the default. The IV.0 heap intrinsics are available when the Heap option is enabled ("H+").

NOTE - The Heap option directive must appear before the program heading. Unlike other options, the Heap option may not be selectively enabled and disabled during compilation.

WARNING - A host and its used units must all use either the II.0

intrinsics or the IV.0 intrinsics, but not both. Units that don't use any dynamic variable allocation intrinsics are compatible with both sets of intrinsics. Heap compatibility is enforced by the system at program invocation time. Intrinsic units escape this check; intermixing of heap mechanisms is done at the risk of the user.

NOTE - The IV.0 intrinsics are maintained in a nonresident unit (called HEAPOPS) located on the system disk. They are loaded into memory along with any program that uses them. Thus, the system disk must be in the system drive when such programs are invoked or a system I/O error occurs. HEAPOPS may be made permanently resident by transferring it from the system support library to intrinsics library using the Library utility. See section 2.3.5 of the System User's Manual for further details.

Example of heap directives:

```
{$H+}
{$H-}
```

See section 3.5 for more information.

5.0.7 Copyright Notices

Copyright notices (or other textual information) may be embedded in a program's code file with the Copyright option. The notice is placed in block 0 of the code file (see the Architecture Guide for details).

The Copyright option is controlled by the pseudo-comment directive "C" used as a string option. The string may contain up to 80 characters.

NOTE - Copyright directives must appear before the program heading.

Example of copyright directive:

```
{$C copyright (c) 1982 by SurfDreck MondoSystems, Inc.}
```

5.0.8 Console Display Suppression

The compiler normally displays a running account of the compiler's progress on the console screen. Enabling the Quiet option suppresses the console display, resulting in faster compilations (due to the time saved by not writing to the console).

The console display is suppressed when the Quiet option is enabled. The Quiet option is controlled by the pseudo-comment directive "Q" used as a switch option.

The default setting of the Quiet option is off. "Q+" enables the option, and suppresses the console display. The display may be restricted to portions of a compilation by selectively enabling ("Q+") and disabling ("Q-") the Quiet option.

Example of quiet compile directives:

```
{$Q+}
{$Q-}
```

See section 5.1.1 in the System User's Manual for a description of the compiler console display.

5.0.9 Segment Residency

Normally, segment procedure code is resident only during it execution, and unit code is resident throughout the program's execution. The default behavior for segment residency may be altered by using the Noload and Resident compile options.

The Noload option allows used units to be swapped as if they were segment procedures. It is controlled by the pseudo-comment directive "N" used as a string option. The directive appears immediately after a USES statement and contains a list enumerating the swappable units. The unit identifiers are separated by commas. Spaces and unrecognizable unit identifiers are ignored.

NOTE - In order for a unit to become swappable, each host using the unit must list it in a Noload directive.

The Resident option allows segments and/or swappable units to be memory-resident throughout the execution of a given procedure. Segment residency is controlled by the pseudo-comment directive "R" used as a string option. The directive appears immediately after the first BEGIN of the desired procedure and contains a list of segments and swappable units to be made memory-resident. Segment and unit identifiers are separated by commas, and spaces are ignored.

NOTE - A Resident option applied to a segment or unit that is already memory-resident has no effect.

NOTE - Misplaced Noload and Resident options are ignored.

NOTE - The MEMLOCK and MEMSWAP intrinsics may also be used to control segment residency. See section 3.1.0 for details.

Example of segment residency directives:

```
program favoritethings;
uses raindrops, roses, sashes;
 {$N raindrops, roses}          {only sashes is resident}

  segment procedure snowflakes;
  begin
  end {snowflakes};

  procedure music;       {snowflakes and raindrops remain}
  begin                  { resident throughout call to music}
    {$R snowflakes, raindrops}
    snowflakes;
  end {music};

begin
  b;
end {favoritethings}.
```

5.0.10 Version Control

Use of the Version Control compile option insures runtime compatibility between hosts and their used units. This option is used to assign a version number to a unit. A unit's version number should be changed whenever its interface section is modified. Execution of a host program is not allowed if the current version of a used unit differs with the version available when the host was compiled. Such hosts must be recompiled with the new version of the unit before they may be executed.

A version number is specified as a non-negative integer argument to the pseudo-commend directive "V". The default version is 0.

NOTE - The Version Control option must occur before the unit heading.

NOTE - Changing a unit's version number may inadvertently force recompilation of several hosts. This overhead may be avoided by using the Library utility (described in the System User's Manual) to construct a code file containing a host program and a copy of the old version of the unit. Assuming the new version does not reside in the intrinsics library, the obsolete, but compatible, version is used when the code file is executed. See section 2.2 in the System User's Manual and section 3.2 for details on unit library searches.

NOTE - The Libmap utility reports unit and host version information. See the System User's Manual for details.

WARNING - Failure to faithfully maintain version control may result

in unpredictable system crashes.

WARNING - Version control is not enforced at bootstrap time. It is the user's responsibility to enforce version compatibility between units installed in the drivers library and the intrinsics library.

Example of version control use:

```
{$V 77}
unit filesystem;
interface

  <...>

end {filesystem}.
```

5.0.11 System Programs

Programs are normally compiled to execute at the lexical level defined for user programs. The User Program option changes the lexical level to that of the operating system; programs compiled at the system level may access system variables outside the scope of user programs.

A program is compiled at the system lexical level when the User Program option is disabled. The User Program option is controlled by the pseudo-comment directive "U" used as a switch option. The default setting of the User Program option is on. "U-" disables the option and also sets the following options: "R-" and "I-".

If the User Program option is off, the user program should be declared as a segment procedure inside of a pseudo-operating system. The pseudo-operating system may contain system variable declarations (usually the real operating system's global declarations), but should not contain any code other than the user program segments.

NOTE - Unlike other versions of UCSD Pascal, no imbedded filler segment declarations are required to avoid conflict with reserved operating segment declarations. In addition, no forward-declared procedures may go undefined.

NOTE - Code files generated for system level programs contain an extra block due to the emission of a dummy segment for the pseudo-operating system level. This block is automatically removed when the Library program (described in the System User's Manual) is used to create a copy of the code file.

NOTE - The User Program option directive must appear before the program header. Unlike other options, User Program may not be selectively enabled and disabled during compilation.

Example of user-level directive use:

```
{$U-}
program fakeOS;
var OSvariable: integer;

  segment procedure userprogram;
  var progglobal: integer;

    segment procedure progseg;
    begin
    end {progseg};

    procedure progproc;
    begin
    end {progproc};

  begin
  end {userprogram};

begin {should contain NO code}
end {fakeOS}.
```

5.0.12 Operating System

Code segment numbers for user programs and units start at 128 and continue through 255. Operating system code segments are numbered 0 through 127. The Operating System compile option determines which segment numbering is to be generated.

The Operating System option is controlled by the pseudo-comment directive "O" used as a switch option. The default setting, "O-", selects a user program compilation.

NOTE - The Operating System option directive must appear before the program heading. The "O+" directive must be accompanied by the "U-" directive (section 5.0.11).

Example of operating system directive use:

```
{$O-}
{$O+,U-}
```

5.0.13 Boolean Negation

In UCSD Pascal, the boolean NOT operator normally calculates the 16-bit 1's complement of its argument. In versions II.0, II.1 and IV.0 it returns this result; in version III.0 it returns the low-order bit and sets all other bits to zero. The Boolean Negation option is used to select the result of boolean NOT evaluations.

The Boolean Negation option is controlled by the pseudo-comment directive "J" used as a switch option. Enabling the option returns

a III.0-style result; disabling the option returns a II.0-style result. The default setting is off ("J-").

The Boolean Negation option may be selectively enabled or disabled anywhere in a program.

NOTE - Most existing UCSD Pascal systems evaluate boolean NOTs in the II.0-manner. However, this has the effect of generating boolean values outside of the range FALSE..TRUE. Such results may cause invalid indexing on arrays whose indices are defined over the BOOLEAN range.

Example boolean negation directive use:

```
{$J-}
{$J+}
{$J^}
```

5.1 Option Summary

B	Starts a conditional compilation section based on the following flag expression.
C	The following string is embedded in the code file as a copyright notice.
D	Declares/defines a conditional compilation flag and sets its value. Default value: TRUE
E	Terminates a conditional compilation section based on the following flag.
H	"H+" selects the IV.0 heap intrinsics. "H-" selects the II.0 heap intrinsics. Default value: H-
I	"I+" generates I/O checks after file I/O operations. "I-" suppresses checks. Default value: I+ The following string contains the name of a text file to be "included" into the source.
J	J+ selects 1-bit NOT result. J- selects 16-bit NOT result. Default value: J-
L	"L+" enables listing. "L-" suppresses listing. Default value: L- The following string contains the name of the compiled listing file.
N	The following unit list is to be made swappable during program execution.
O	O+ selects operating system compilation. O- selects user program compilation. Default value: O-
P	"P" emits a page break in listing.
Q	"Q+" enables the console display. "Q-" suppresses the display. Default value: Q+

R — "R+" generates range checks on array indeces and subrange variables. "R-" suppresses checks. Default value: R+

The following segment/unit list is to be made resident throughout the current procedure.

S — "S+" specifies swapping compiler. Default value: S-

U — "U+" specifies a user-level program. "U-" specifies a system level program and "I-,R-". Default value: U+

V — The following integer is the version number of the unit.

VI. OPERATING SYSTEM CUSTOMIZATION

This chapter describes ways in which users may customize the Advanced Operating System. Customization may be achieved with either user-programmed extensions to the operating system or changes to the user interface.

Section 6.0 describes how to program and install operating system extensions. Section 6.1 describes how to change the system prompt line and program execution processing. Section 6.2 shows how to construct and install system I/O device drivers. Section 6.3 describes how the system execution error processing may be modified. Section 6.4 shows how to transform the system breakpoint processor into a custom applications debugger.

NOTE - Operating system customization relies heavily on a thorough understanding of units and the library system. Units are described in section 3.2. The library system is described in the System User's Manual.

6.0 Operating System Extensions

Generally speaking, the UCSD Pascal operating system is a collection of subsystems that provide runtime support for the execution of a program. Such subsystems are usually loaded and initialized at system bootstrap time. They are available for use by any program at any time, and are terminated at system halt time. (For example, the file system is loaded and its variables are initialized at bootstrap time. It is available for use by any program.)

A user-defined subsystem is programmed as a unit, called an intrinsic unit, and is installed in the intrinsics library, *SYSTEM.INTRINS. Any user unit may be designated as an intrinsic unit merely by including it in the intrinsics library (using the Library utility described in the System User's Manual) and rebooting. Facilities provided by intrinsic units are available simply by using the unit; when attempting to locate a used unit during both program compilation and execution, the compiler and the operating system look first in the intrinsics library and then elsewhere.

Code segments for intrinsic units are loaded and their global data spaces are allocated at system bootstrap time. Their initialization sections are called after all operating system initialization is complete; any operating system facilities may be used anywhere in intrinsic units. Both unit code and data spaces remain memory-resident for the duration of system execution. Unit termination sections are executed when the H(alt command is invoked from the system prompt.

NOTE - A trade-off is made between the maintenance of intrinsic units and free memory space. The M(emory command may be invoked at the system prompt line to help monitor the amount of free memory available under various configurations of the intrinsics library.

NOTE - Since intrinsic unit initialization and termination sections are executed at system bootstrap and halt times, they are not executed as a result of the invocation of programs that use intrinsic units. Intrinsic units requiring reinitialization after use by a program should provide a reinitialization procedure in their interface sections.

Intrinsic units may be used for a number of purposes. Library units used by many programs may be designated as intrinsic units in order to reduce the time required to invoke the programs, as disk accesses and program setup time are reduced when used units are already memory-resident and initialized. Large applications may use intrinsic units to maintain data structures and procedure calls common to a number of (possibly called or chained) programs. A spooler may be created by programming the initialization section of an intrinsic unit to start background tasks which continue to execute during normal system operation.

NOTE - Intrinsic units and their segments do not occupy any of the 128 segment positions allocated to user programs.

WARNING - Unit version control is not enforced at bootstrap time. It is the user's responsibility to assure version compatibility between units installed in the intrinsics library. Version control is, however, enforced at program invocation time. See section 5.0.10 for further details.

WARNING - Heap usage consistency involving intrinsic units is not enforced. It is the user's responsibility to assure heap compatibility between units installed in the intrinsics library. See section 5.0.6 for further details.

WARNING - Simultaneous calls from concurrent tasks to segment procedures declared within intrinsic units should be protected by critical sections, or the segment should be MEMLOCKed; otherwise the system may crash.

6.1 System Prompt Line and Program Execution

A shell is a program that performs user interface functions and is capable of starting programs in response to user commands. The system shell program, *SYSTEM.SHELL, performs all system-level user interface and program invocation processing (i.e. it maintains the system prompt line, executes system and user programs, prints the bootstrap welcome message, invokes *SYSTEM.STARTUP or *PROFILE .TEXT, performs program chaining, etc). The system shell is executed by the operating system at system bootstrap time and is re-executed at system re-initialization time.

User-written programs may function as shells by using the facilities provided by the PROGOPS unit described in the Library User's Manual. A shell executes a program as if the program were a procedure of the shell. A shell may be executed from the system prompt line (or any other shell) in the same way as any normal program, or it may be executed at system bootstrap time by naming it *SYSTEM.SHELL. When a shell terminates, the shell that invoked it resumes. The standard system shell terminates when the H(alt command is entered at the system prompt line. This causes the termination code for each intrinsic and driver unit to be executed (see sections 6.0 and 6.2) and the system to halt.

The shell facility is useful in constructing dedicated application systems and custom operating systems. It is also useful in calling entire programs as if they were procedures. See section 7.10 for an example of a shell.

NOTE - Section 2.4.4 of the System User's Manual describes the effects of using I/O redirection options when X(ecuting the system shell.

NOTE - Termination of a program because of an execution error does not cause the re-initialization of the system. Instead, the execution error number is returned to the shell that invoked the program.

NOTE - The source for the standard system shell is available for modification from ACD.

6.2 System Device Drivers

System device drivers are collections of routines called by the unit I/O intrinsics (section 3.9) to perform I/O functions. These routines are packaged as a unit and reside in the drivers library file, *SYSTEM.DRIVERS. The *SYSTEM.DRVINFO file contains the mapping between I/O unit numbers and logical device numbers of specified device driver units. A new system driver is installed by using the Library utility to include it in the driver library and then by using the Drvr.Info utility to specify the I/O unit numbers that address it. See section 2.3.1 in the System User's Manual for further details.

Each driver unit interface section contains a standard set of declarations, including a read, write, clear, status, initialization, termination, and power fail restart routine. The interface section must have the following form:

```
type DevWindow  = packed array [0..0] of char;
     DevStatRec = array [0..29] of integer;

  procedure DevInit;

  function DevRead (UnitNo     : integer;
                    StartBlock : integer;
                Var Buffer     : DevWindow;
                    Index      : integer;
                    BytesLeft  : integer;
                    Control    : integer)   : integer;

  function DevWrite (UnitNo     : integer;
                     StartBlock : integer;
                 Var Buffer     : DevWindow;
                     Index      : integer;
                     BytesLeft  : integer;
                     Control    : integer)   : integer;

  function DevClear (UnitNo : integer) : integer;

  function DevPower (UnitNo : integer) : integer;

  function DevStatus (UnitNo  : integer;
                  Var StatRec : DevStatRec;
                      Direct  : boolean) : integer;

  procedure DevTerm;
```

The DevInit procedure initializes the driver and device state; it should be called from the driver unit initialization section. The DevTerm procedure terminates the driver and shuts down the device; it should be called from the driver unit termination section. The DevPower function is currently unused. The DevRead, DevWrite, DevClear, and DevStatus functions provide the UNITREAD, UNITWRITE, UNITCLEAR, and UNITSTATUS routines, respectively for the device.

They should operate in the manner described in section 3.9 and Appendix D. Note that device driver construction is discussed in section 7.6.

Most parameters passed to the unit I/O intrinsics are passed to the driver routines without modification, but the UnitNo, Buffer, and Index parameters are the results of intermediate calculations. The UnitNo parameter specifies a driver-dependent logical device number derived from the mapping provided in the *SYSTEM.DRVINFO file (i.e. 0 for virtual floppy 0). The buffer address for the DevRead and DevWrite routines consists of a buffer base pointer (Buffer) and a byte-offset (Index). The actual buffer starts at the byte addressed by Buffer[Index]. All other parameters match those passed to the unit I/O intrinsics. Section 7.6 describes how to construct Pascal code to communicate with I/O devices.

Drivers enumerated in the *SYSTEM.DRVINFO file are loaded and their data space is allocated at system bootstrap time. The initialization section of each driver is called before the system itself is initialized. The driver may be called at any time during system operation. The termination section of each driver is called when the system halts (i.e. when the H(alt command is invoked from the system prompt line); it is executed after the termination sections of any intrinsic units (see section 6.0).

NOTE - Normally, user programs may access drivers installed in the driver library only through the unit I/O intrinsics. A driver may be called by both the system and user programs by including it in the intrinsic library, instead.

NOTE - Drivers may use routines provided by other drivers when necessary. The ALL.DRIVERS library contains a copy of the code and interface section of each available driver. The interface section may be examined using the Libmap utility described in the System User's Manual. A special driver, called SYSDRIVER, provides the INTRENABLE procedure which enables the interrupt system. Driver sources for several devices are available from ACD.

NOTE - The Tester utility (TESTER.TEXT on the AOS release disk) may be used to test disk drivers during driver debug and validation. It tests the data transfer functions most often used during system execution.

WARNING - Drivers should not call segment procedures during the execution of their initialization or termination sections unless the driver for the bootstrap device has been used within the calling driver (e.g. USES FLOPPYDRIVER). This assures that the bootstrap device driver has been initialized and is capable of loading the segment procedure. Drivers calling unit I/O intrinsics must also use the unit containing the driver associated with the unit I/O call. Also, since drivers are initialized and called before system initialization is complete, no file system functions (i.e. READ, REWRITE, etc) should be used within the driver.

WARNING - Unit version control is not enforced at bootstrap time. It is the user's responsibility to assure version compatibility

between units installed in the drivers library. See section 5.0.10 for further details.

WARNING - Simultaneous device accesses by concurrent tasks resulting in segment procedure calls by system drivers should be protected by critical sections, or the segment should be MEMLOCKed; otherwise the system may crash.

6.2.0 Bootstrap Drivers

Bootstrapping the AOS on the PDQ-3 involves three stages. In the first stage, the system monitor (Chapter 7 in the System User's Manual) reads the bootstrap off the boot device and executes it. In the second stage, the bootstrap reads the system off the boot device and executes it. In the third stage, the system prepares itself for normal execution. Each bootstrap stage requires its own copy of the boot device driver.

The device driver required by the first stage is provided as a procedure burned into the system monitor prom (the HDT Prom described in the Hardware User's Manual). The procedure accepts the boot drive number and the bootstrap memory address as parameters. It reads the bootstrap from the boot drive into the specified memory buffer. The driver is declared as follows:

```
procedure readboot (device, address : integer);
```

A prom burner, the Prom utility, and the sources for the HDT Prom are required to burn a new HDT Prom. Call ACD for assistance.

The device driver required by the second stage is a copy of the system device driver. The Make.Boot utility (described in the System User's Manual) constructs a bootstrap from the BOOT.CODE file, the bootstrap serial device driver, and a given system device driver; it writes the bootstrap onto the bootstrap area of the boot device.

The third stage uses the drivers installed in the driver library.

6.3 Exception Handling

Execution errors are processed by the EXCEPTION unit installed in the system support library file. The standard exception handler calculates the site of an error and prints an error message on the system console (see section 2.0 of the System User's Manual). It uses the EXCEPINFO unit, also installed in the system support library file, to translate an execution error number (and possibly an I/O error number) into error message text.

Custom exception handling may be provided by reprogramming the EXCEPTION unit and substituting it for the standard exception handler. Note that the exception handler may use units installed in the intrinsic library containing the global data of an application; when an execution error occurs, it may perform an orderly termination of the application.

The EXCEPTION unit provides a single routine, called HandleException, in its interface section. The interface section appears as follows:

```
Unit Exception;
Interface

  Function HandleException (Error: Integer): Boolean;
```

The HandleException function accepts the number of the execution error (see Appendix B for a list of standard execution error numbers), processes the error, and returns a boolean function result. The function value is returned FALSE if the exception handler has determined that the program must be terminated; otherwise the function value is returned TRUE.

Custom execution errors and I/O errors may be programmed into the Pascal system by adding the error description text to the EXCEPINFO unit. These errors are raised by calling the PROGEXCEPTION and PROGIOSET procedures provided by the PROGOPS unit. The EXCEPINFO and PROGOPS units are documented in the Library User's Manual.

The sources to the standard EXCEPTION and EXCEPINFO units are available from ACD.

NOTE - The termination sections of units used by a program are executed regardless of the occurance of an execution error.

NOTE - Stack overflow errors are not processed by the exception handler. The erroneous program is terminated without recourse.

WARNING - The caveats applying to system driver units stated in section 6.2 apply to the exception handler as well.

6.4 Breakpoint Processor

Execution of the HALT intrinsic (section 3.11.10) invokes the system breakpoint handler. Breakpoints are processed by the DoHalt routine in the HALTUNIT unit installed in the system support library. The standard breakpoint processor prints a halt message on the console and waits for keyboard input.

A new HALTUNIT unit may be constructed and substituted for the standard breakpoint processor, allowing custom breakpoint handling in applications requiring debugging output based on complex conditions. For example, the breakpoint processor may use intrinsic units containing the global data of an application; when a breakpoint is encountered, it may perform debugging operations based on the states of the global variables.

NOTE - The SYSDRIVER driver unit contains the ENTERHDT procedure, which may be used to invoke the system monitor (Chapter 7 of the System User's Manual). The SYSDRIVER unit contained in the standard driver library calls the system monitor resident in the PDQ-3 HDT prom. The SYSDRIVER unit in the HDT.DRVR.CODE file (provided on the AOS release disk) calls a copy of the system monitor contained in the SYSDRIVER unit; this version provides memory display primitives for use on systems whose HDT prom does not contain such primitives. Section 2.3.1 of the System User's Manual describes how to install a new driver.

WARNING - The caveats stated in section 6.2 apply to the breakpoint handler as well.

The standard HALTUNIT unit is listed below:

```
unit HaltUnit;
interface
  procedure DoHalt;
implementation
  procedure DoHalt;
  begin
    write ('Halted.  Hit <return>.');
    readln;
  end {DoHalt};
end.
```

VII. PROGRAMMING PRACTICES

This chapter describes common UCSD Pascal programming practices. Note that these practices are implementation dependent - they should not be used in programs intended for use outside of the UCSD Pascal system.

Section 7.0 describes packed variable allocation (knowledge of which aids the design of compact data structures). Section 7.1 explains how to access arbitrary words, bit fields, and bits in memory. Section 7.2 explains how to perform unsigned integer arithmetic and comparisons. Section 7.3 explains how to perform logical operations on word quantities (e.g. integers). Section 7.4 shows how the UCSD Pascal heap implementation can be exploited to create dynamic arrays. Section 7.5 describes the implementation of data prompts suitable for an interactive environment. Section 7.6 explains how to write asynchronous device drivers in UCSD Pascal. Section 7.7 explains how the PDQ-3's concurrent I/O system can be used to create multiterminal programs. Section 7.8 describes methods for locating execution errors in Pascal programs. Section 7.9 describes how and why to use separate compilation units. Section 7.10 explains how programs may be called as procedures.

7.0 Packed Variables

This section describes the implementation of packed variables in UCSD Pascal. Record and array data are stored in a packed representation when their type declaration is preceded by the reserved word PACKED. Packing is not performed on files and sets. (Note that bit strings are the default representation of sets in UCSD Pascal; specifying a set as packed is unnecessary and thus ignored.)

Packed format is usually chosen for data that occupies large amounts of space, but which is accessed relatively infrequently. A decision to use packed data should be influenced by two distinct tradeoffs: speed versus space, and code space versus data space. The code-versus-data tradeoff is the increase in program size (caused by extra code for packing and unpacking data at every variable reference) versus the space saved by compressing the data representation. The space-versus-space tradeoff is the space saved by compressing the data representation versus the slower access time (caused by packing and unpacking data during every variable reference). Note that the first tradeoff is a function of the static variable references contained in a program, while the second is a function of the dynamic variable references executed by a program.

Sections 7.0.0 and 7.0.1 present examples of packed arrays and records respectively.

Users should be aware of the packing rules (and restrictions) in order to construct packed data structures consuming minimal amounts of space. Section 7.0.2 presents the packing rules and restrictions for UCSD Pascal.

NOTE - The SIZEOF intrinsic is useful for determining the size of a packed type. See section 4.35 for details.

7.0.0 Packed Arrays

UCSD Pascal performs packing of arrays if the array type definition is preceded by the reserved word PACKED. Consider the following type definitions:

```
type
     large = array[0..9] of char;
     small = packed array[0..9] of char;
```

Character variables are normally allocated a full word, but can fit in a single byte. A variable of type "large" is allocated ten words of data space; each character element is allocated a full word for storage. A variable of type "small" is allocated five words of data space. Two character elements are packed into each word; each element is allocated a single byte for storage.

Examples of packed arrays:

```
type
     one = packed array [0..7] of 0..3;
     two = packed array [0..2] of 0..31;
     zip = packed array [0..2] of set of 0..8;
```

Variables of type "one" fit in one word; each element is two bits long, and the eight elements fit in a single word. Variables of type "two" are one word long; each element is five bits long, and three of them fit in a word (with the high order bit unused). Variables of type "zip" require three words. Packing is not performed, as the base type is nine bits long; each element is allocated a full word (9 bits for the set, 7 bits unused).

The following type definitions are not equivalent in UCSD Pascal:

```
type
    a = packed array[0..5] of array[0..7] of char;
    b = array[0..5] of packed array[0..7] of char;
```

Type definitions containing nested arrays are packed only if the last occurrence of the reserved word ARRAY is preceded by PACKED; in the example above, packing is performed on variables of type "b", but not on variables of type "a". To ensure packing of types containing mixes of arrays and records, precede all occurrences of ARRAY and RECORD with PACKED.

NOTE - String constants are type-compatible only with packed character arrays; they are incompatible with unpacked character arrays.

7.0.1 Packed Records

UCSD Pascal performs packing of records if the record type definition is preceded by the reserved word PACKED. Consider the following type definitions:

```
type large = record
               a,b,c,d: char;
             end;
     small = packed record
               a,b,c,d: char;
             end;
```

Character variables are normally allocated a full word, but can fit in a single byte. A variable of type "large" is allocated four words of data space; each character field is allocated a full word for storage. A variable of type "small" is allocated two words of data space. Two character fields are packed into each word; each field is allocated a single byte for storage.

Examples of packed records:

```
type
     one = packed record
             f1,f2,f3,f4: 0..3;
             byte: 0..255;
           end;
     two = packed record
             f1,f2,f3: 0..31;
           end;
     zip = packed record
             f1,f2,f3: set of 0..8;
           end;
```

Variables of type "one" fit in one word. Each "f" field is two bits long; the four fields fit into a single byte. The "byte" field occupies the other byte in the word. Variables of type "two" are one word long; each field is five bits long, and three of them fit in a word (with one bit unused). Variables of type "zip" require three words. Packing is not performed, as the fields are larger than 8 bits; each field is allocated a full word (9 bits for the set, 7 bits unused).

The following type definitions are not equivalent in UCSD Pascal:

```
type
     a = packed record
           i: integer;
           r: packed record
                r1,r2: char;
              end;
         end;
     b = packed record
           i: integer;
           r: record
                r1,r2: char;
              end;
         end;
```

Type definitions containing nested records are packed only if the innermost occurrence of the reserved word RECORD is preceded by PACKED; in the example above, packing is performed on variables of type "a", but not on variables of type "b". To ensure packing of types containing mixes of records and arrays, precede all occurrences of ARRAY and RECORD with PACKED.

NOTE - When a record contains a variant part, it is allocated enough space to contain the largest variant (unless dynamically allocated with NEW(<variant list>)).

7.0.2 Packing Rules

This section describes the rules for packing variables in UCSD Pascal; these consist of constraints imposed on variable packing and optimizations performed within those constraints.

The following table displays packed and unpacked sizes for some common types:

Type	Unpacked	Packed
integer	word	word
boolean	word	1 bit
char	word	8 bits
real	2 words	2 words
subrange a..b	word	log base 2 (b - a) bits
set 0..n : n<16	word	n bits

NOTE - Subranges are packable only if their range values are nonnegative; if either bound is negative, they are not packed. With the exception of sub-word sets, structured fields (i.e. records, arrays, and multiword sets as record fields) begin on word boundaries, and are thus unpackable.

The primary constraint on packed variables is that fields in packed variables cannot be packed across word boundaries.

Records benefit from packing only if they contain a number of scalar, subrange, or set fields, each of which can be stored in 8 bits or less. Consecutively declared fields needing more than 8 bits apiece cannot be packed (because of the word boundary restriction); they are allocated one or more words for storage, and are accessed as unpacked fields. Fields are allocated storage space in the order in which they are declared in a record; thus, rearranging the fields in a record sometimes results in a smaller record, as fields can be packed only if they are declared adjacently to other packable fields (see example below).

NOTE - When a packable field is forced to occupy a full word because of adjacent word-aligned fields, it is accessed as an unpacked field.

NOTE - To wit: packed records may contain both packed and unpacked fields - packing is determined by the sizes and declaration order of the record's fields.

Example of rearranging record fields:

```
type foon = packed record
              b1: boolean;     {1 word }
              i1: integer;     {1 word }
              b2: boolean;     {1 word }
              r1: real;        {2 words}
              b3: boolean;     {1 word }
            end;       {total = 6 words}

newfoon = packed record
            b1,b2,b3: boolean;  {1 word }
            i1: integer;        {1 word }
            r1: real;           {2 words}
          end;          {total = 4 words}
```

In foon, i1 and r1 are constrained to word boundaries because they occupy integral numbers of words (1 and 2 words respectively). The Boolean fields are unpackable because of their adjacence to word-aligned fields. In newfoon, the adjacency of the Boolean fields allows them to be packed into a single word (note that there is space in the word for up to 13 more (packed) Boolean fields).

Arrays benefit from packing only if their base type is a scalar, subrange, or set type which can be stored in 8 bits or less. Packing is not performed if an array element is larger than 8 bits.

See section 7.1.1 (bit fields) and the Architecture Guide for more information on variable packing.

7.1 Accessing Words, Bits, and Bit Fields

This section presents methods enabling UCSD Pascal programs to access arbitrary words, bit fields, and bits in PDQ-3 memory. Very few programs require direct access to memory; it is generally restricted to system programs (see section 7.6 for an example).

NOTE - Memory access can also be accomplished with in-line machine code (section 3.10); however, the methods presented here are less error-prone and don't require knowledge of the PDQ-3 instruction set.

The following examples utilize the standard Pascal feature known as a record variant; it is used here to provide a controlled form of type conversion, which in turn allows exploitation of the machine representations of various Pascal types for gaining direct access to the machine.

WARNING - The methods presented here work on other UCSD Pascal implementations; however, they are highly nonportable because of different system memory configurations and/or machine architectures (see the Archiecture Guide for details).

7.1.0 Words

Example of arbitrary word access:

```
program c1;
type address = integer;
     word    = integer;

  function peek(location: address): word;
  {  return value at specified address  }
  type trick = record case boolean of
                 true:  (addr: address);
                 false: (wordptr: ^word);
               end;
  var  access: trick;
  begin
    access.addr := location;
    peek := access.wordptr^;
  end {peek};

begin
  ...
end {c1}.
```

In this example, the record variant overlays an integer variable and a pointer variable (both of which are word quantities). The integer variant is used to assign a memory address into the pointer; the pointer variant is used to access the memory word. Note that the integer value specifies a word address on the PDQ-3. One drawback of this method is the necessity of specifying large (i.e. > 8000 hex) addresses as negative integer values. For

example, the word at address FFFF hex is returned by calling peek(-1). (A hexadecimal calculator such as the TI Programmer is useful in these situations.) A second drawback is the specification of address 8000 hex; it cannot be represented as an integer constant because its decimal value is -32768, which is treated by the compiler as a long integer constant. 8000 hex can be specified by the integer expression 32767 + 1 (remember? - no integer overflow checks in UCSD Pascal).

7.1.1 Bit Fields

Bit fields are accessed in a similar fashion using packed records. The pointer variant is defined to point to a one-word packed record, which is declared so that the desired bit fields are accessible. See section 7.0 for details on record packing.

Example of arbitrary bit field access:

```
program c2;
type address    = integer;
     byte       = 0..255;
     nibble     = 0..15;

  procedure doio(deviceNum: nibble; command: integer);

  { synchronous device driver  }

  const deviceAddr = -16833;

  type deviceWord = packed record
                      switch:  0..1;
                      control: 0..7;
                      device:  nibble;
                      unused:  byte;
                    end;

       trick = record case boolean of
                 true:  (addr: address);
                 false: (wordptr: ^deviceWord);
               end;

  var  access: trick
  begin
    access.addr := deviceAddr;
    with access.wordptr^ do
      begin
        switch := 1;
        device := deviceNum;
        control := command;
        repeat until switch = 0;
      end;
  end {doio};

begin
  ...
end {c2}.
```

In this example, the record variant overlays an integer and a pointer to a packed record (both of which are word quantities). The integer variant is used to assign a memory address into the pointer; the pointer variant is used to access the fields of the record. Note that the integer value specifies a word address on the PDQ-3.

Bit field allocation in a packed record starts from the least significant bit of a word. Defining bit 0 and bit 15 as the least and most significant bits respectively, the record declared in the previous example is allocated in the following fashion: switch is in bit 0, control occupies bits 1-3, device occupies bits 4-7, and unused occupies bits 8-15.

NOTE - Fields in a packed record are not always allocated in the order of their appearance in the record declaration; the exception is a list of variables (separated by commas) which share the same type declaration. These variables are allocated in reverse order of their appearance in the list.

Example of reverse field allocation in records:

```
type widgetWord = packed record
                    switch:  0..1;
                    control: 0..7;
                    b1,b2,b3,b4: boolean;
                    device:  nibble;
                    unused:  nibble;
                  end;
```

widgetWord is allocated in the following fashion: switch is in bit 0, control occupies bits 1-3, b4 is in bit 4, b3 in bit 5, b2 in bit 6, b1 in bit 7, device occupies bits 8-11, and unused occupies bits 12-15.

NOTE - A data access optimization applies to packed records containing a quantity of up to 8 bits immediately before a word-aligned field. If this quantity is not already word-aligned, it is adjusted to occupy the byte preceding the word-aligned field and is accessed as an 8 bit field.

Example of byte-alignment optimization:

```
type twimp = packed record
               b1: boolean;     {starts on bit 0}
               ch: char;        {starts on bit 1}
             end;

     newtwimp = packed record
                  b1: boolean;     {starts on bit 0}
                  ch: char;        {starts on bit 8}
                  i: integer;      {starts next word}
                end;
```

7.1.2 Bits

Bits are accessed in a similar fashion to bit fields; sets are used instead of packed records. The pointer variant is defined to point to a one-word set; set operations are used to test and set individual bits in the word.

Example of arbitrary bit access:

```
unit bitter;
interface
  type address = integer;
       bits    = 0..15;

  procedure setbit(addr: address; bit: bits);

  procedure clearbit(addr: address; bit: bits);

  function bitset(addr: address; bit: bits): boolean;

implementation

  type bitstring = set of bits;
       trick     = record case boolean of
                     true:  (addr: address);
                     false: (bitsPtr: ^bitstring);
                   end;

  var access: trick;

  procedure setbit;
  begin
    access.addr := addr;
    access.bitsPtr^ := access.bitsPtr^ + [bit];
  end {setbit};

  procedure clearbit;
  begin
    access.addr := addr;
    access.bitsPtr^ := access.bitsPtr^ - [bit];
  end {clearbit};

  function bitset;
  begin
    access.addr := addr;
    bitset := bit in access.bitsPtr^;
  end {bitset};

end {bitter}.
```

In this example, the record variant overlays an integer and a pointer to a set (both of which are word quantities). The integer variant is used to assign a memory address into the pointer; the pointer variant is used to access the set.

7.2 Unsigned Integer Manipulation

It is occasionally necessary to treat the contents of an integer variable as an unsigned value. Integer operators expect signed values as arguments; thus, they must be used carefully when dealing with unsigned integers.

Integer variables are defined to contain values in the range -32768 .. 32767, while unsigned integers are defined to contain values in the range 0 .. 65535. Both representations are equivalent in the range 0 .. 32767; however, unsigned values in the range 32768 .. 65535 are treated by integer operations as signed values in the corresponding range -32768 .. -1.

Integer arithmetic operators are +, -, *, div, and mod. Only +, - and * may be used with unsigned integers (as a consequence of their lack of overflow checking); div and mod do not work correctly with large unsigned integers.

Integer comparision operators are =, <>, <, <=, >, and >=. Only = and <> may be used with unsigned integers; the remaining operators do not work correctly with large unsigned integers. Unsigned comparison operators may be programmed by the user.

Example of unsigned comparisons:

```
function uLess(a,b: integer): boolean;
begin
  if (a > 0) and (b < 0) then
    uLess := true
  else if (a < 0) and (b > 0) then
    uLess := false
  else
    uLess := a < b;
end {uLess};

function uGtr(a,b: integer): boolean;
begin
  if (a > 0) and (b < 0) then
    uGtr := false
  else if (a < 0) and (b > 0) then
    uGtr := true
  else
    uGtr := a > b;
end {uGtr};
```

NOTE - The example above assumes that the < and > operators produce consistent results when treating unsigned integers as signed integers. This is not entirely the case. These operators produce inconsistent results when comparing the value -32768 (only bit 15 set). Correct versions of uGtr and uLess account for this at the expense of additional code. More efficient unsigned comparison operators may be constructed using the PMACHINE intrinsic. See section 3.10 for examples. Unsigned operators are also available in the NumCon library unit (described in the Library User's

Manual).

The system integer read routines accept both signed and unsigned integers. The integer write routine expects signed integers, but may be used for unsigned integer I/O with the understanding that the user is responsible for translating between signed and unsigned representations. (This is not especially difficult if one has access to a calculator such as the TI Programmer.) If this is not sufficient, unsigned integer write operations may be programmed by the user.

Example of an unsigned write routine:

```
program uwrite;
const uint = -20000;        {B1E0 hex or 45536}
var ones,tens: integer;
    ureal: real;
begin
  if uint < 0 then
    begin
      ureal := 65536.0 + uint;
      uint := uint + 32767 + 1;
      tens := uint Div 10 + 3276;
      ones := uint Mod 10 + 8;
    end
  else
    begin
      ureal := uint;
      tens := 0; ones := uint;
    end;
  writeln ('unsigned integer = ', ureal:1:1);
  writeln ('unsigned integer = ', tens + ones div 10,
                                  ones mod 10);
end.
```

Two different methods are used to write the unsigned integer uint. The first method translates the integer to a real and prints the real value. Unfortunately, this requires the memory-residence of the real I/O routines. The second method breaks the integer into two parts and reconstructs it on output.

Unsigned integer values are returned by the MEMAVAIL, RMEMAVAIL, VARNEW, and TIME intrinsics. Section 7.1 describes one use of unsigned integers.

7.3 Full-word Logical Operations

UCSD Pascal allows logical operations on word quantities (e.g. integers, characters, and pointers). The standard functions ORD, ODD, and CHR are defined as type transfer functions in UCSD Pascal; they do not modify the ordinal value of their arguments (see section 2.4). The logical operators AND and OR are full-word operators; they do not mask off the high order bits of their result.

Example of full-word logical operations:

```
program logical;
var I: integer;
begin
  I := 556;
  I := ord(odd(I) and odd (255));
  { The high byte of I has been masked off }
  { I now contains the integer value 44    }
end {logical}.
```

The NOT operator may be used either to perform a 16-bit 1's complement or to complement the low order bit and mask off the other bits. Its behavior is selected using the Boolean Negation compile option described in section 5.0.13.

7.4 Variable-sized Buffer Allocation

The use of variable-sized buffers increases the efficiency of programs that access byte arrays whose size is unknown at compile time. Variable-sized buffers may be allocated at runtime by taking advantage of the system's dynamic variable allocation mechanisms.

A variable-sized buffer is created in the heap by coercing the system heap intrinsics to generate a pointer to a dynamic array variable contained in a memory space the size of the desired buffer. Memory is accessed beyond the declared size of the dynamic array variable by indexing into the array with compile-time range checks suppressed (section 5.0.5). A program must maintain its own range checks by keeping track of the size of the allocated memory space and not indexing beyond its upper bound.

NOTE - The FILLCHAR, MOVELEFT, and MOVERIGHT intrinsics are useful in manipulating variable-sized arrays. However, the byte count arguments to these intrinsics must be in the range 0..32767. Calls containing negative byte counts perform no action.

WARNING - Array indices on the PDQ-3 are treated as signed integers. Indexing array elements greater than 32767 yield unexpected and often unfortunate results.

Two different buffer allocation strategies are available; one uses the II.0 heap intrinsics, the other uses the IV.0 heap intrinsics.

7.4.0 II.0 Heap Strategy

The II.0 heap centers around the heap pointer. Dynamic variables are created by calls to the NEW intrinsic, which assigns the current value of the heap pointer to the associated pointer variable, and then moves the heap pointer up by the number of words allocated for the variable. The heap grows from lower addresses to higher addresses; as a result, consecutively NEWed dynamic variables are allocated contiguously in memory.

Programs may take advantage of this implementation feature to create variable-length arrays; a variable-length array is constructed by creating a series of dynamic variables at runtime, and then treating them as a single, large array.

The UCSD filer and editor use this method to create large arrays for manipulating file information; their arrays are allocated in integral numbers of blocks. In order to create as large an array as possible, these programs use the MEMAVAIL intrinsic to determine the maximum buffer size allowed by the host system's configuration. MEMAVAIL returns the number of unused words in system memory; construction of a variable-length array consists of repeatedly NEW'ing one-block arrays until MEMAVAIL returns a value less than or equal to the program's memory threshold.

A memory threshold is defined as the minimum amount of memory required to execute a program (independent of the variable-length

buffer); this includes space for variables and code segments used by the program in the course of its execution. The memory threshold for a given program is determined by making a rough estimate of the maximum space consumed by the program, and then tuning the estimate (usually through trial and error) to a minimum. A conservative minimum threshold for any program is 500 words (for system overhead) plus the program's requirements.

WARNING - This method of buffer allocation must be treated as a critical section (section 3.0.1.0) when tasks contend for heap space. Refer to section 7.4.1 for a method that is more secure in a multitasking environment.

Example of the II.0 variable-sized buffer strategy:

```
program makeAbuffer;
const
     threshold = 1000;
     maxblks   = 60;
type
     block  = array [0..255] of integer;
     bufptr = ^block;
var
     buffer, bufblock: bufptr;
     bufblks, index: integer;
begin
  bufblks := 1;
  new(buffer);
  repeat
    new(bufblock);
    bufblks := bufblks + 1;
  until (bufblks >= maxblks) or
  ((memavail > 0) and (memavail <= threshold));

  { Note that bufblks * 512 < 32767 }
  { array bounds = 0..(bufblks * 512 - 1) }

  fillchar(buffer^,bufblks * 512,0);
  {$R-}
  ...
  buffer^[index] := 4;
  ...
end {makeAbuffer}.
```

7.4.1 IV.0 Heap Strategy

When using the IV.0 heap, successive NEWs are not guaranteed to allocate dynamic variables adjacently in memory. The VARNEW function is available for variable-sized buffer allocation. It accepts a buffer size request (in words) and allocates a buffer of that size. See section 3.5 for details.

The size of the largest possible memory buffer may be determined using the VARAVAIL function. It accepts a list of segments that might be memory-resident during the life of the variable-sized

buffer. VARAVAIL returns the size of the largest memory space available, subject to the residency of all of the enumerated segments. The VARNEW intrinsic may be called to allocate a buffer of that size.

NOTE - It is indeterminable whether the result of the VARAVAIL function refers to an area between the stack and heap, or to a memory space recycled by the DISPOSE, VARDISPOSE, or MEMLOCK intrinsics. Thus, it is difficult to decide whether to apply the program threshold calculations described in the II.0 strategy. If no calls to DISPOSE or VARDISPOSE have been made, the result of the VARAVAIL function refers to an area between the stack and heap; thus, the program threshold calculation should be applied.

NOTE - Although the VARNEW intrinsic is protected from task contention, combination of the VARAVAIL and VARNEW functions may yield unexpected results in a multitasking environment. A temporal window exists between a call to VARAVAIL and a call to VARNEW during which the memory reported by VARAVAIL may be allocated to another task. In this case, the call to VARNEW fails (VARNEW returns 0) and another call to VARAVAIL is required before the VARNEW may be retried.

Example of the IV.0 variable-sized buffer strategy:

```
program makeAbuffer;
const
     threshold = 1000;
     maxblks   = 60;
type
     block  = array [0..255] of integer;
     bufptr = ^block;
var
     buffer: bufptr;
     trash, bufsize, index: integer;
begin
  bufsize := varavail('') - threshold;
  if (bufsize < 0) or (bufsize > maxblks*256) then
    bufsize := maxblks*256
  else
    bufsize := bufsize - bufsize mod 256;
  trash := varnew(buffer, bufsize);

  { array bounds = 0..(bufsize - 1) }
  fillchar(buffer^,bufsize*2,0);
  {$R-}
  ...
  buffer^[index] := 4;
  ...
end {makeAbuffer}.
```

7.5 Data Prompts

This section describes the implementation of interactive data prompts. UCSD Pascal provides some support for interactive data prompts; one example of this is the acceptance of backspace characters when reading integers and strings from the console. One feature conspicuously absent is the ability to respond to invalid user inputs such as illegal characters in an integer or illegal file names in a file name; the system responds to invalid data and file names with an execution error. Therefore, responsibility for detection of and responses to invalid inputs falls to the program itself; this is accomplished by suppressing I/O checks and using the IORESULT intrinsic to implement user-proof error recovery.

The following three sections present robust implementations for single character prompts, integer prompts, and file prompts. The file prompt uses the file system conventions described in section 2.1.6 in the System User's Manual.

NOTE - Though not demonstrated in this section, the GOTOXY intrinsic is useful for constructing interactive screen displays. See section 3.11.3 for details.

NOTE - Many of the inadequecies of the UCSD Pascal system support for data prompts are solved in the NUMCON, REALCON, and SCCNTRL library units described in the Library User's Manual.

NOTE - The UNITCLEAR intrinsic may be used to flush the keyboard type-ahead buffer before issuing a critical data prompt. This ensures that the prompt receives an explicit response from the user (rather than soaking up whatever characters happen to be queued for input). Examples of this feature can be found in the filer commands K(runch and R(emove.

7.5.0 Character Prompts

Example of character prompt:

```
program p1;
var ch: char;
    done: boolean;
begin
  done := false;
  repeat
    write('Do you wish to continue? (Y/N) ');
    repeat
      read(keyboard,ch);
    until ch in ['n','N','y','Y'];
    writeln(ch);
    done := ch in ['y','Y'];
  until done;
end {p1}.
```

This example demonstrates secure input checking. The prompt indicates acceptable responses to the question. The keyboard file is used to filter out invalid responses before they can reach the screen; only when a valid response is received is the input echoed to the console. Note that the prompt accepts both lower- and upper-case characters as valid responses.

7.5.1 Integer Prompts

Example of integer prompt:

```
program p2;
var int: integer;
    done: boolean;
begin
  done := false;
  repeat
    {$I-}
    repeat
      write('Type a number (0 exits) : ');
      readln(int);
    until ioresult = 0;
    {$I^}
    writeln(' You typed: ',int);
    done := (int = 0);
  until done;
end {p2}.
```

This example demonstrates explicit, user-defined error recovery. If I/O checks were enabled, READLN would cause an execution error whenever the input didn't match the format defined for an integer (e.g. input containing alphabetic characters); in this example, the input prompt merely repeats itself if an invalid integer is entered.

7.5.2 File Prompts

Example of file prompts:

```
program p3;
var infile,outfile: file;
    filename, inname: string[30];
    result: integer;

  procedure addSuffix(var fname:  string;
                          suffix: string);
  begin
    if fname[length(fname)] = '.' then
      delete(fname,length(fname),1)
    else
      fname := concat(fname,suffix);
  end {addSuffix};

begin
  repeat
    write(input file (<cr> to escape): ');
    readln(filename);
    inname := filename;
    if length(filename) = 0 then exit(program);
    addSuffix(filename, '.TEXT');
    {$I-}
    reset(infile,filename);
    result := ioresult;
    {$I^}
    if result <> 0 then
      writeln(' Cannot open ',filename);
  until result = 0;
  repeat
    write(output file (<cr> for same): ');
    readln(filename);
    if length(filename) = 0 then
      filename := inname;
    addSuffix(filename, '.CODE');
    {$I-}
    rewrite(outfile,filename);
    result := ioresult;
    {$I^}
    if result <> 0 then
      writeln(' Cannot open ',filename);
  until result = 0;
end {p3}.
```

This example again demonstrates explicit, user-defined error recovery. If I/O checks were enabled, RESET and REWRITE would cause an execution error whenever an invalid file name was entered; in this example, the prompts reappear after responding with an error message. Note the use of file name suffixes; this conforms to the file system's naming conventions for file name prompts (as des-

cribed in section 2.1.6 of the System User's Manual). Also note the presence of a standard escape response for the input prompt (typing a carriage return escapes the prompt), and a short-circuit on the output prompt (typing a carriage return uses the input file title in the output file name).

7.6 Device Drivers

A device driver is a set of one or more routines which provides an interface between a program and a peripheral device. The program initiates a device operation by calling the device driver with parameters describing the desired operation. The device driver performs the actions necessary to perform the device operation, and notifies the program of the device status.

This section describes how to write drivers for Q-bus compatible I/O devices. Unlike most computers, all device drivers on the PDQ-3 are written in a high-level language (i.e. UCSD Pascal); thus, they are less incomprehensible than drivers written in assembly language.

Section 7.5.0 describes the interface between programs and device drivers. Section 7.5.1 explains how to access devices in UCSD Pascal. Section 7.5.2 describes the handling of direct memory access (DMA) devices. Section 7.5.3 describes the handling of device interrupts.

NOTE - This section describes how to write device drivers for use in programs; operating system device drivers are discussed in section 6.2.

7.6.0 Driver Interface

A driver typically consists of a set of functions or procedures declared in a program or unit. I/O driver parameters usually include a device identifier, a source or destination address, and a data transfer length; depending on the driver, any one of these parameters may be implicit in the driver's definition.

Examples of driver interfaces:

```
TapeRead (1{device number}, buffer, 512{byte count});
LPWrite (buffer, SizeOf(buffer){byte count});
TapeRewind (2{device number});
```

The source or destination address is normally an area of memory corresponding to a variable declared in the program. Parameter type checking often prevents extensive use of a driver by restricting the type of variables allowed as a source or destination address; programs often need to read or write data from a variety of (differently typed) variables.

One method of overriding type-checking constraints is the use of variant records; this allows the source or destination parameter to accept arguments having different types. The following example assumes that the programmer wishes to read data from a device into two different types of variables. The driver parameter is declared with a type allowing both kinds of variables.

Example of multitype parameter:

```
type  VariantStructure = record
                           case integer of
                             0 : (FirstStructure : Type1);
                             1 : (SecondStructure : Type2);
                         end {of VariantStructure};

procedure Driver (Var Buffer : VariantStructure);
```

The variant part may be extended in the following manner to accommodate byte-oriented drivers:

Example of byte-oriented address parameter:

```
type ByteArray = packed array [0..0] of 0..255;
     VariantStructure = record case integer of
                          0 : (FirstStructure : Type1);
                          1 : (SecondStructure : Type2);
                          2 : (MemoryImage : ByteArray);
                        end {of VariantStructure};
```

The driver may access any byte in the buffer by indexing through the MemoryImage variant. Note that range checks (section 5.0.5) must be suppressed in order to access arbitrary bytes without causing an execution error.

7.6.1 Device Access

Devices are accessed through their device registers. The driver views these registers as the contents of specific memory addresses; accessing these memory addresses causes a device register to be accessed. Reading from a device register obtains device status information; writing to a device register issues device commands.

Device register access is accomplished in UCSD Pascal by assigning the memory address of the device register to a pointer variable, and then by accessing the register through the pointer. Since most devices have several device registers located in contiguous addresses, the pointer is usually declared to point to a multiword record describing the device registers. The record fields are declared so that they coincide with the various bit fields in the device registers (see sections 7.0 and 7.1 for details).

NOTE - The process of storing into a packed record field involves reading the entire word containing the record field, updating the

record field, and then writing the modified word back to the record. Beware of side effects caused by reading and/or writing of packed fields adjacent to the field being modified.

The following example presents a simple device driver for the DLV-11 (a bidirectional serial line whose device registers start at FFB8 hex). A pointer is initialized with the address of the device register block. The program reads characters from the receiver and echoes them to the transmitter.

Example of simple DLV11 device driver:

```
program DLV11Demo;
const DLV11Address = -72;        {FFB8 hex}
type
DLV11Rec = record
             RCsr : packed record  {receiver status}
                      Unused : 0..31;   {unused bits}
                      IntEnab: boolean; {interrupt enable}
                      Ready : boolean;  {char received}
                    end {of RCsr};
             RBuf : char;           {input data}
             XCsr : packed record  {xmitter status}
                      Unused : 0..31;   {unused bits}
                      IntEnab: boolean; {interrupt enable}
                      Ready : boolean;  {xmitter empty}
                    end {of XCsr};
             XBuf : char;           {output data}
           end {of DLV11Rec};
var DLV11 : record
              case integer of
                0 : (Ptr : ^DLV11Rec);
                1 : (Value : integer);
            end {of DLV11};

begin
  DLV11.Value := DLV11Address;
  with DLV11.Ptr^ do
    begin
      RCsr.IntEnab := False;
      XCsr.IntEnab := False;
      repeat
        repeat    {wait for a char to arrive}
        until RCsr.Ready;
        repeat    {wait for xmitter to become available}
        until XCsr.Ready;
        XBuf := RBuf;     {send character received}
      until false;
    end;
end.
```

Specific operational details of Q-Bus I/O devices can be found in the hardware documentation provided by the device's manufacturer. This information contains device register descriptions, operational assumptions, and device register addresses. Due to architectural

differences between the PDQ-3 and other Q-Bus-based computers (i.e. others are byte addressed - the PDQ-3 is word addressed), the device register addresses specified in the hardware documentation must be converted to hex and then divided by two to obtain the proper PDQ-3 address. For example, the DLV11 address is specified as 377560 octal in most hardware documentation. This is equivalent to 1FF70 hex; dividing by 2 obtains FFB8 hex, which is the proper PDQ-3 address. Note that this does not apply to interrupt vectors (section 7.6.3).

NOTE - Device drivers may require protection (i.e. semaphores) from task contention.

7.6.2 DMA Operations

Some Q-Bus devices are capable of performing direct memory access (DMA) operations. These devices provide a device register which contains the memory address of the next buffer element on which an I/O operation is to take place. The device driver must determine the starting buffer address and supply it to the device before a DMA operation is initiated. The address is obtained with the PMACHINE intrinsic (see section 3.10 for details), which returns the address in a temporary variable.

Due to architectural differences between the PDQ-3 and other Q-Bus processors, most I/O devices require that the PDQ-3 buffer address be shifted left one bit before being used. The low order 16 bits of the shifted result is written to the buffer address device register. The high order bit (bit 17) is written to a second device register, part of which is dedicated to address extension bits.

One method of shifting the address involves the use of a temporary record. The DMA address register in the device register declaration is assumed to be declared as type DeviceAddress.

Example of DMA buffer address shifting:

```
type DeviceAddress = packed record
                       LoBit : 0..1;
                       Hi15Bits : 0..32767;
                     end {of DeviceAddress};
var RawAddress : packed record
                   Lo15Bits : 0..32767;
                   HiBit : 0..1;
                 end {of RawAddress};

pmachine (^RawAddress, ^DMAbuffer, STO);
{ RawAddress := memory address of DMA buffer }
<DMA address register>.LoBit := 0;       {perform the shift}
<DMA address register>.Hi15Bits := RawAddress.Lo15Bits;
<DMA address extension reg> := RawAddress.HiBit;
```

7.6.3 Interrupts

Interrupt-driven device drivers are implemented with the concurrency intrinsics WAIT, SIGNAL, and ATTACH (see section 3.0). The interrupt vector used by a device can be found in the hardware documentation describing the device; it is specified in octal, and need only be converted to hexadecimal to obtain the correct vector address for the PDQ-3.

NOTE - The interrupt system becomes disabled when an interrupt occurs. The driver is responsible for re-enabling the interrupt system so that subsequent interrupts are handled. Re-enabling is done by setting the Int Enab bit of the System Status Register (see the Hardware User's Manual for details).

WARNING - A device's interrupt enable bit should never be cleared while the PDQ-3 interrupt system is enabled. The interrupt system is disabled by clearing the Int Enab bit of the System Status Register (see the Hardware User's Manual for details). Note that it is necessary to repeatedly reset the Int Enab bit until it stays at 0. The interrupt system should be disabled as infrequently as possible since the act of disabling the interrupt system may lead to system crashes.

NOTE - Because no method currently exists for deallocating attached semaphore variables, programs must turn off interrupt-causing devices before terminating in order to prevent possible system crashes. See section 3.0.2 for details.

This program uses a clock to beep every 3 seconds:

```
program clock;
const ClockVector = 26 {60 Hz timer interrupt};
var   tick: semaphore;
      pid: processid;

  process timer;
  const bell = 7;
        ThreeSeconds = 180 {ticks};
  var   ticks: integer;
  begin
    ticks := 0;
    repeat
      wait(tick);
      pmachine ((-988), (70), 196);   {reenables interrupts}
      ticks := ticks + 1;
      if ticks >= ThreeSeconds then
        begin ticks := 0; write(chr(bell)) end;
    until false;
  end {timer};

begin
  seminit(tick,0);
  attach(tick,ClockVector);
  start(timer,pid,500,180);
end.  {clock}
```

7.7 Multiterminal Applications

This section describes the role of UCSD Pascal's concurrency features and the PDQ-3 asynchronous I/O system in the development of multiterminal programs. A multiterminal hardware configuration consists of a PDQ-3 communicating with two or more terminals over serial lines. A multiterminal program assigns a task to each of the terminals; each task is responsible for receiving input from its assigned terminal, initiating actions specified by the terminal user, and sending output back to the terminal. When a task initiates an I/O operation, it is suspended until the I/O operation is completed; this allows other tasks to proceed with their execution.

The variables declared in a multiterminal program may be divided into two classes: private variables, and shared variables. Private variables define the state of each user. They are declared in the terminal process; thus, each task is assigned its own set of private variables. A typical example of a private variable is the file variable which each task uses for reading and writing to its own terminal. Shared variables define the resources being shared by all tasks. They are declared in the main program; because they are subject to task contention, they must be guarded by semaphores. A typical example of a shared variable is a disk file which is accessible to every task.

The example presented below illustrates the adaptation of a single-user program to a multiterminal environment. Most of the conversion effort involves the determination of which global variables in the program should be shared, and which should be private. (Note that nonglobal variables are unaffected by the conversion.) Semaphores are assigned to each shared variable; references to shared variables must be guarded by semaphores to ensure mutual exclusion of the terminal tasks. A terminal process must be written; it contains the declarations of all private variables, and does nothing more than initialize its private variables and execute the main program routine. The main program routine must be modified so that references to (formerly) global variables become references to private task variables.

NOTE - Each task must have a task stack space large enough to execute the main program (see section 3.0.0.2 for details).

NOTE - See section 3.1.0 for restrictions imposed by interaction between tasks and segments.

NOTE - In addition to the changes mentioned above, it is necessary to locate all critical sections (see section 3.0.1.0) and protect them with semaphores. Failure to identify and protect all potential critical sections can cause multiterminal programs to fail intermittently.

Example of single-terminal program:

```
program StraightTalk;
const EscapeChar = ':';
var   Ch : char;
      InFile : interactive;
      OutFile, LogFile : text;

  procedure LogChar (Ch : char);
  begin
    writeln (LogFile, 'Received ', Ch);
  end;

begin
  reset (InFile, 'REMIN1:');          { init global vars }
  rewrite (OutFile, 'REMOUT1:');
  rewrite (LogFile, 'CONSOLE:');
  repeat                              { main routine }
    read (InFile, Ch);
    write (OutFile, Ch);
    LogChar (Ch);
  until Ch = EscapeChar;
end {StraightTalk}.
```

Example of multiterminal program:

```
program CrossTalk;
var LogLock : semaphore;
    LogFile: text;                { shared }
    pid: processid;

  procedure LogChar (Ch : char; Receiver : string);
  begin
    wait (LogLock);
    writeln (LogFile, 'Received ', Ch, ' from ', Receiver);
    signal (LogLock);
  end;

  process ATerminal (InName, OutName : string);
  const EscapeChar = ':';
  var Ch : char;                  { private }
      InFile : interactive;
      OutFile : text;
  begin
    reset (InFile, InName);                { init private vars }
    rewrite (OutFile, OutName);
    repeat                                 { main routine }
      read (InFile, Ch);
      write (OutFile, Ch);
      LogChar (Ch, InName);
    until Ch = EscapeChar;
  end {ATerminal};

begin
  { init tasks & shared variables }
  seminit (LogLock, 1);
  rewrite (LogFile, 'CONSOLE:');
  start (ATerminal ('REMIN1:', 'REMOUT2:'), pid, 500);
  start (ATerminal ('REMIN2:', 'REMOUT1:'), pid, 500);
end.
```

7.8 Locating Execution Errors

This section describes how to locate the source of an execution error in a UCSD Pascal program. When the operating system detects an execution error, it halts the program and displays an error message on the screen (see section 2.0.0 in the System User's Manual for more information). The error message includes a field such as:

```
Segment PROSE    Proc 90, Offset 101
```

This message field specifies the error location in terms of the code file structure. The "Segment" value indicates the name of the current code segment. The "Proc" value indicates the current procedure within the segment. The "Offset" value indicates the procedure-relative byte offset of the instruction which caused the error.

NOTE - It is possible to obtain a list of procedure calls leading up to an execution error by setting the Error List Length greater than 1. This field is maintained using the Setup utility described in the System User's Manual.

7.8.0 Using Compiled Listings

A compiled listing displays the segment number, procedure number, and code offset of each line in the program (see section 5.0.0 for details on compiled listings). Finding the source of an execution error consists of finding the Pascal statement in the listing for the segment whose list procedure and offset numbers match those of the error message. Note that while the procedure numbers can be matched exactly, the code offset displayed in the error message usually falls between the code offsets displayed in the listing. The error location can be narrowed down to the line whose displayed offset value is the closest value less than or equal to the error offset.

NOTE - In some situations, the execution error displays segment or offset numbers which don't appear in a compiled listing of the program. If an execution error occurs in an unrecognized segment, something is seriously wrong with the system! (Call ACD for assistance.) If an execution error is traced to a used unit, a compiled listing of the unit must be obtained before the error can be traced any further. When strings or long integers are passed as parameters, execution errors can occur in the vicinity of the associated procedure call. See sections 3.4 and 3.6 for more information.

Having located the suspected source line, the execution error message should be sufficient to determine the cause of the error. "Value range error" indicates that the program tried to assign a value outside of the declared bounds of an array or subrange variable. "Integer overflow" is only generated by long integer operations; it cannot result from integer operations (as no overflow checks are performed on integers). "Divide by zero" is

detectable in integer, long integer, and real division. User I/O errors are generated either by an invalid input or by a file system error.

7.8.1 Without Using Compiled Listings

It is possible to trace execution errors to the procedure level without the use of compiled listings; all that is required is knowledge of the program's overall structure (i.e. declaration order of procedures) and an understanding of the compiler's rules for assigning procedure numbers in a compiled program.

Procedures in a program or segment are assigned procedure numbers in the order in which their headings appear, starting at procedure number one. (Note that forward declarations count as headings.) Procedure number one in a segment or program is the outermost block of the segment or program; in both cases, the first local procedure declaration is assigned procedure number two, the next three, and so on. Note that procedure numbers are assigned independent of the lexical nesting of procedures within a segment.

The procedure assignment rules presented in this section may be partially verified by examining the compiled listing printed in section 5.0.0.

7.8.2 Further Investigations

Locating the source of an execution error is often only the first step in finding program errors; it is often necessary to begin printing debug information (by inserting WRITELN statements into the incorrect program) in order to investigate values of suspected variables prior to the execution error.

7.9 Programming with Units

This section demonstrates the value of the UCSD Pascal UNIT in the economical and reliable production of applications software. The major benefits of unit usage are derived from their ability to act as a foundation for the development of increasingly complex facilities, and their ability to be separately and independently compiled.

The following two sections demonstrate unit usage by showing how to develope a unit and how to take advantage of pre-existing units. Unit syntax and semantics are discussed in section 3.2. The unit library system is described in section 2.2 of the System User's Manual. The integration of user units into the operating system is described in section 6.0.

7.9.0 Unit Development

Program development using units is faster and more reliable than traditional methods. Large sections of code normally included in a program may be separated into units where they are available simply by reference rather than by in-line compilation. Facilities provided by such units are also available to any other programs requiring them. This approach saves time during program compilation and allows a unit to be tested and maintained independently of the program. Since a single copy of a unit is shared among all client programs, bug fixes and performance optimizations applied to a unit are automatically available to all client programs.

The first step in the development of a unit is the specification of its interface section. When possible, it is prudent to structure interface variables and procedures to provide generalized functionality rather than facilities specific to an individual program.

Once the unit interface has been specified, it is wise to consider how each component of the interface is to be tested. This results in a greater understanding of all details of the interface functions, and provides a foundation for the construction of test and validation suites.

Implementation of the unit is performed by coding the interface functions and writing initialization and termination code for the unit's global variables. Note that pre-existing units may provide functions valuable in implementing this unit (see section 7.9.1).

Once the unit has been compiled, it may be installed in the library system and tested. Testing and validation suites should be developed to exercise each component of the unit interface. These suites may be used during initial unit debugging and as a debugging aid during unit maintenance. Note that the unit may be programmed and maintained as an in-line unit of the validation suite program. This arrangement facilitates the validation of the unit after updates since the validation suite is always recompiled with the unit.

NOTE - It is most expedient to install the unit in the user library (see section 2.2.3 of the System User's Manual) until debugging is complete.

7.9.1 Using Pre-existing Units

A number of units have been developed for use with the Advanced Operating System, including units which perform complex system, programming, and applications functions. They afford access to routines that might be impossible for most programmers to write (i.e. routines that require intimate knowledge of the system architecture) or routines that might be merely inconvenient to rewrite each time they are required. A partial listing of pre-existing units available for use with the AOS appears in Appendix K.

WARNING - Since the use of a unit causes the entire unit to be memory-resident during program execution, excessive use of pre-existing units in the construction of new units can lead to exorbitant runtime memory requirements.

The example below demonstrates the use of several of the units listed in Appendix K. The program performs the essential functions of the Printer utility documented in the System User's Manual. Note that the majority of the code serves to link the complex functions of the units together. Identifiers provided by used units are underlined.

```
program DumpFiles;
uses SpoolUnit, PatternMatch, DirInfo, NumCon;
const LinesInPage  = 66;
      PrintPerPage = 60;
      Esc          = 27;
var Lines,
    OutUnit  : integer;
    S        : string;

  procedure GetFileName (var Name: string; Lines: integer);
  begin
    writeln;
    write ('File to print ? ');
    readln (Name);
    if length (Name) <> 0 then
      if Name[1] = '\' then
        begin
          delete (Name, 1, 1);
          Lines := LinesInPage;
        end {of if}
      else
        Lines := PrintPerPage;
    if length (Name) <> 0 then
      if Name[length (Name)] = '.' then
        delete (Name, length (Name), 1)
      else
        Name := concat (Name, '.Text');
  end {GetFileName};

  procedure GetOutUnit (var OutUnit: integer);
  var Temp: string;
  begin
    repeat
      writeln;
      write ('What is the output unit (1, 6, 8) ? ');
      readln (Temp);
      if NStrToInt (Temp, 1, OutUnit) = 1 then
        exit (DumpFiles);
    until OutUnit In [1, 6, 8];
  end {GetOutUnit};
```

```
  procedure PrintFiles (Name: string; Lines: integer);
  var List : DListP;
      Ch   : char;
      Heap : ^integer;
  begin
    mark (Heap);
    if DDirList (Name, [DText], List, False) = DOkay then
      while List <> Nil Do
        begin
          Name := concat (List^.DVolume, ':', List^.DTitle);
          repeat
            write ('Print ', Name, ' ? ');
            read (Ch);
            if not eoln then writeln;
          until Ch in ['Y', 'y', 'N', 'n', ' ', Chr (Esc)];
          if Ch in ['Y', 'y'] then
            case SpoolFile (Name, Lines,
                            LinesInPage, OutUnit) of
              SpNotFound : writeln (Name, ' not found');
              SpFull     : begin
                             writeln ('Queue is full');
                             List^.DNextEntry := Nil;
                           end {of SPFull};
              SpQueued   : writeln (Name, ' queued');
            end {of case};
          if Ch = Chr (Esc) then List := Nil
          else List := List^.DNextEntry;
        end {of while}
    else writeln ('No files found');
    release (Heap);
  end {PrintFiles};

begin
  GetOutUnit (OutUnit);
  SpoolStop;
  repeat
    GetFileName (S, Lines);
    if length (S) <> 0 then
      PrintFiles (S, Lines);
  until length (S) = 0;
  SpoolRestart;
end.
```

7.10 Programs as Procedures

Applications occasionally require the execution of certain functions already performed by existing programs. Combination of the system program call and I/O redirection facilities allows application to perform these operations by invoking existing programs either interactively or secretly.

The PROGCALL function provided by the PROGOPS unit (described in the Library User's Manual) allows a program to be called as a procedure from anywhere in a program or unit. I/O redirection options and the name of a program are passed to PROGCALL as parameters. Using the I/O redirection facilities to provide preprogrammed input and redirected output allows silent invokation of the program; the user need never be aware of the program call. Interactive calls may be performed by using no I/O redirection options. I/O redirection options are documented in section 2.4.4 of the System User's Manual.

NOTE - The time required by the PROGCALL intrinsic to set up a program for execution may result in unacceptable delays in cases where the program is repetitively invoked. Combining the PROGSETUP and either the PROGSTART or PROGEXECUTE intrinsics results in the elimination of the program setup overhead in these cases. For further details, refer to the Library User's Manual.

WARNING - The PROGOPS intrinsics should be called from only one concurrent task at a time. Attempts to execute these intrinsics from more than one task simultaneously may yield bizarre and unpredictable results.

The following example demonstrates the PROGCALL function; it compiles each ".TEXT" file on a given volume (assuming each text file contains a program or unit) to its corresponding ".CODE" file, then calls the Filer for a directory listing of all ".CODE" files. Keyboard input is accepted after the listing is started, and the program regains control when the Filer is terminated. Note that there are no facilities in this program for handling exceptional conditions. Note also that the DIRINFO unit is used to obtain the names of all unit text files.

```
program CompAndList;
uses Progops, PatternMatch, DirInfo;
var Name,
    Prefix : string;
    List   : DListP;

  procedure Call (ProgName: string);
  var Error   : string;
      ErrNum  : integer;
  begin
    if not ProgCall (ProgName, Error, ErrNum) then
      begin
        writeln (Error);
        exit (CompAndList);
      end;
  end {Call};

begin
  write ('Volume id: ');
  readln (Prefix);
  if length (Prefix) <> 0 then
    if DDirList (concat (Prefix, ':=.Text'), [DText],
                 List, True) = DOkay then
      begin
        while List <> Nil Do
          begin
            with List^ do
              Name := Copy (DTitle, ., DFPat^.CompLen)
            writeln ('Compiling ', Name);
            Call (Concat ('*System.Compiler.  pp=', Prefix,
                          ' i="', Name, ',$," o=bucket:'));
            List := List^.DNextEntry;
          end {of while};
        Call (Concat ('*System.filer.  i="e',
                      Prefix, ':=.code,'));
        writeln;
        writeln ('Units, ProgCalls, and I/O redirection!');
      end
    else writeln ('No files found')
  else writeln ('Aborted');
end.
```

7.11 Programming for I/O Redirection

Programs may be written to take advantage of the system I/O redirection facilities. The system provides routines that suspend, restart, and provide status information on the I/O redirection facilities. Special system devices are also provided to access the standard input and standard output.

I/O redirection processing may be suspended or resumed using procedures provided in the COMMANDIO unit (described in the Library User's Manual). The SuspendRedir procedure temporarily suspends the most recent input and output redirection options. I/O is performed using the input and output streams existing prior to the creation of the current input and output streams. The ResumeRedir procedure resumes I/O to the current input and output streams. These procedures may be used to generate alarm messages when a program encounters an unexpected condition. The SuspendT procedure temporarily suspends the most recent t-file redirection options. The ResumeT procedure restores t-file I/O. These procedures may be used to guarantee that certain output reaches only the device attached to the standard output rather all t-files, too.

I/O redirection status may be obtained using the ProgRedir function provided by the PROGOPS unit (described in the Library User's Manual). This function returns true if redirection options have been applied to either the standard input or the standard output.

A program may access the standard input or standard output through the unit I/O intrinsics or local file variables instead of through the predeclared INPUT and OUTPUT file identifiers. I/O performed to the STANIN: (unit 21) and STANOUT: (unit 22) volumes is directed to the standard input and standard output, respectively.

The following program takes advantage of I/O redirection using the facilities described above. It outputs a report either to a file or simultaneously to the console and a printer. Assume that the program resides in the file PRINT.CODE and that it may be invoked by specifying "Print TO=Printer:" as the execution option list.

```
program PrintReport;
uses CommandIO;
var OutFileName : string;

    procedure PrintReport (Name : string);
    var OutFile : text;
    begin
{$I-} rewrite (OutFile, Name);
{$I^} if IORESULT = 0 then
        begin
          ResumeT;       {Activate write to t-file}
{$I-}     writeln (OutFile, 'Sample report:');
          <...>
{$I^}     if IORESULT <> 0 then
            begin
              SuspendRedir;     {in case of O= redir option}
              writeln ('Error writing to ', Name, '!!!');
              ResumeRedir;
            end;
          SuspendT;      {Deactivate write to t-file}
        end
      else
        writeln ('Can''t open file ', Name);
    end {of PrintReport};

begin {of PrintReport}
  SuspendT;      {Don't write prompts to t-files}
  write ('Print to what file (<return> for STANOUT:) ? ');
  readln (OutFileName);
  if length (OutFileName) = 0 then
    OutFileName := 'Stanout:'
  else
    if OutFileName[length (OutFileName)] = '.' then
      delete (OutFileName, length (OutFileName), 1)
    else
      OutFileName := Concat (OutFileName, '.Text');
  PrintReport (OutFileName);
  <...>
end {of PrintReport}.
```

APPENDIX A: STANDARD I/O RESULTS

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Unit Number
3	Bad Mode, Illegal operation
4	Undefined hardware error
5	Lost unit, Unit is no longer on-line
6	Lost file, File is no longer in directory
7	Bad Title, Illegal file name
8	No room, insufficient space
9	No unit, No such volume on line
10	No file, No such file on volume
11	Duplicate file
12	Not closed, attempt to open an open file
13	Not open, attempt to access a closed file
14	Bad format, error in reading real or integer
15	Ring buffer overflow
16	Write Protect; attempted write to protected disk
17	Illegal block number
18	Illegal buffer address

APPENDIX B: STANDARD EXECUTION ERRORS

0	No error
1	Invalid index, value out of range
2	No segment, bad code file
3	Exit from uncalled procedure
4	Stack overflow
5	Integer overflow
6	Divide by zero
7	Invalid memory reference <bus timed out>
8	User Break
9	System I/O error
10	User I/O error
11	Unimplemented instruction
12	Floating Point math error
13	String too long
14	Illegal heap operation

APPENDIX C: CONDITIONS CAUSING I/O ERRORS

1	CRC Error	Returned whenever a CRC (cyclic redundancy check) or Parity error occurs.
2	Bad Unit Number	Returned for accesses to a device for which there is no driver declared.
3	Bad Mode	Returned for attempts to read on a write-only device or write on a read-only device.
4	Undefined Error	Returned when an error of indeterminable type occurs.
5	Lost Unit	Returned by the file system only; it indicates that a disk has gone off-line during an I/O operation.
6	Lost File	Returned by the file system only; it indicates that a file expected to be in a disk directory is not present.
7	Bad Title	Returned by the file system only; it indicates an attempt to open a file with an invalid file name.
8	No Room	Returned by the file system only; it indicates either an attempt to open or extend a disk file when disk space is unavailable, or an attempt to open a new file on a disk with a full directory.
9	No Unit/Volume	Returned either after an attempt to access an off-line unit or after an error occurs during UNITCLEAR. Also returned by the file system to indicate an attempt to access a volume which is not on-line.

10	No File	Returned by the file system only; it indicates an attempt to open a nonexistent disk file.
11	Duplicate File	Returned by the file system only; it indicates an attempt to create more than one temporary file with the same file name on a single disk volume.
12	Not Closed	Returned by the file system only; it indicates an attempt to open a file variable which is already connected to an external file.
13	Not Open	Returned by the file system only; it indicates an attempt to access a file variable which is not connected to an external file.
14	Bad Format	Returned by the file system only; it indicates an attempt to read a real value or integer value with incorrect input format.
15	Ring Buffer Overflow	Returned during a read from a serial device after its input buffer has overflowed. (Not currently implemented)
16	Write Protected Disk	Returned when attempting to write to a write-protected disk.
17	Illegal Block Number	Returned when attempting to access a nonexistent block on a block-structured device, or when a seek error occurs.
18	Illegal Buffer Address	Returned when attempting to initiate an I/O operation with a non-word-aligned starting buffer address. (Applies only to block-structured devices)

APPENDIX D: STANDARD I/O UNIT ATTRIBUTES

This section describes the operations defined for the PDQ-3 system I/O units in their standard configuration. Since system device drivers and the I/O device configuration itself may be modified in the field (see section 6.2 for details), this section may not apply to a custom I/O configuration. All operations are performed with the unit I/O intrinsics described in section 3.9. See the Hardware User's Manual for more information on I/O devices.

I/O units can be divided into two classes according to their attributes: serial units, and block-structured units. A unit's class determines the kinds of operations performed on the unit and the available I/O options. I/O options are specified by setting various bits in the control word parameter of the UNITREAD and UNITWRITE intrinsics.

NOTE - An option is enabled if its bit is set to 1; otherwise, it is disabled. The low order bit in a control word is bit 0. Unused bits in control words should always be set to 0. For example, a control word value of 6 sets bits 1 & 2 to 1 (and all other bits to 0).

D.0 Serial Unit Attributes

Serial units read and/or write sequences of characters to a serial device. Each serial input unit maintains its own input queue. Certain characters are treated as control characters rather than data. (See section 1.4 in the System User's Manual for more information on control characters.)

D.0.0 Serial Input Attributes

Characters recognized as control characters by serial input operations are:

-- Control-S and control-Q suspend and resume device output.

-- <eof> is treated as the end-of-file marker; the end-of-file marker is placed in the buffer, and the input operation is terminated immediately.

-- Control-D is the floppy disk type key.

-- Control-F discards ("flushes") subsequent device output.

-- Control-P invokes the system monitor.

-- Control-X discards the contents of the console type-ahead queue.

-- <null> is treated specially in some cases (see section 6.1.3 in the System User's Manual and section 3.3.2 in this manual).

Serial input options are defined as follows:

Bit 1 Raw input mode; disable all control character processing. Suppress character echoing when reading from unit 1. Note that the input unit retains this mode after completing a read operation, thus affecting the handling of subsequent asynchronously received input. This mode can be disabled by performing another read operation with bit 1 reset; a read of 0 bytes is sufficient to enable or disable the mode.

Bit 2 Suppress recognition of the end-of-file character.

Bit 3 Suppress CR/LF generation in character echoing when reading from unit 1.

D.0.1 Serial Output Attributes

Characters treated specially by serial output operations are:

-- The ASCII DLE character is treated as the escape character of a blank compression character sequence; the next character is defined to contain a byte value which is 32 greater than the number of blank characters to be written to the device. Note that DLE processing applies only to text files; it must be suppressed when writing code or data files to a serial device.

-- The ASCII CR character is defined as a "new-line" character in text files. Whenever CR is written to a serial device, the I/O system automatically follows it with the ASCII LF character (line feed). Note that CR/LF processing applies only to text files; it must be suppressed when writing code or data files to a serial device.

-- The ASCII FF character is defined as a "Clear Screen" character in text files. Whenever FF is written to units 1 or 2, the I/O system automatically substitutes the console Clear Screen sequence defined with the Setup utility (see section 8.3 in the System User's Manual). Note that FF processing applies only to text files; it must be suppressed when writing code or data files to a serial device.

Serial output options are defined as follows:

Bit 2 Suppress DLE expansion.

Bit 3 Suppress automatic LF after CR. Suppress FF substitution.

NOTE - I/O is somewhat faster if bits 2 & 3 are set.

D.0.2 Handshaking Protocols

All serial units support the RS-232 DTR ("Data Terminal Ready") handshaking protocol. Serial unit drivers use DTR (when supported by the serial device) to simulate the control-S / control-Q handshaking described in section D.0.0. Note that raw input mode disables normal control-S / control-Q handshaking by swallowing the control characters, while DTR protocol is unaffected by raw input mode.

D.1 Block-structured Unit Attributes

Block-structured I/O options differ between floppy disk units and hard disk units. The UNITSTATUS intrinsic (section 3.9.2) indicates whether a block-structured unit is a floppy or a hard disk.

D.1.0 Floppy Unit Attributes

Floppy disk units perform automatic switching between single and double density disks. Double-sided disks are handled by manual switches via the disk type key (see section 1.4.3.4 in the System User's Manual).

I/O options for floppy disks units are defined as follows:

Bit 1 Physical sector I/O. Allows access to any physical sector on the disk. Disk sectors are addressed by logical sector number; the first sector on the disk is sector 0. Note that physical sector mode allows normally inaccessible disk sectors to be accessed (e.g. sectors on track 0). The starting block parameter is redefined to denote the starting logical sector number. If the byte count parameter is 0, the I/O operation transfers one physical sector of data; the size of a physical record is determined by the type of the current disk (128 for single density disks and 256 for double density disks) and may be obtained by using the UNITSTATUS intrinsic. If the byte count parameter is nonzero, it is treated as a normal byte count.

D.1.1 Hard Disk Unit Attributes

I/O options for hard disks specify addressing relative to one of three areas on the hard disk: the data space, the bootstrap space, and the configuration space. Since the mapping between I/O units and physical drives is hidden at the driver level, the physical drive number may also be specified.

The bootstrap space contains a copy of the system bootstrap. It is usually up to 3328 bytes (6.5 blocks) long.

The configuration space contains a 1024 byte table specifying how the data space is partitioned into virtual floppies. The table is

maintained by the Drive.Con utility described in the System User's Manual. It is declared as follows:

```
HardConTbl: record
              MaxEntry      : integer; {last ConEntry}
              BlocksSegment : integer; {blocks in a segment}
              SegmentsDrive : integer; {segments in a drive}
              ConEntry      : array [0..MaxConLen] of
                                packed record
                                  StartArea  : integer;
                                  IsMounted  : boolean;
                                  LastBlock  : 0..32767;
                                  Description : string[19];
                                end {of ConEntry};
            end {HardConTbl};
```

The MaxEntry field is a zero-based index specifying the last valid entry in the ConEntry table. BlocksSegment indicates the number of 512-byte blocks contained in a disk track. SegmentsDrive contains the number of tracks per physical disk drive. The ConEntry table lists each virtual floppy contained on the disk drive. StartArea contains the zero-based starting physical track number for the floppy (note that track 0 is reserved for the bootstrap and configuration spaces). IsMounted indicates whether the virtual floppy is available for system access. LastBlock is a zero-based value containing the number of the last accessable block on the floppy; it is a multiple of BlocksSegment and usually coincides with the floppy size contained in the floppy volume directory. Description contains up to 19 user-supplied characters describing the contents of the virtual floppy.

Bits 1 and 2 I/O addressing environment. "0" specifies addressing relative to the data space allocated for the unit. Offsets from the beginning of the data space are expressed in blocks. "1" and "2" specify addressing relative to the bootstrap and configuration spaces, respectively. Offsets from the beginning of these areas are expressed in logical disk sectors; the size of a sector is determined by the type of the hard drive and may be obtained by using the UNITSTATUS intrinsic. The first sector on the disk is sector 0. Note that the bootstrap and configuration spaces are normally inaccessible to the file system. If the byte count parameter is 0, the I/O operation transfers one sector of data; otherwise the specified byte count is used.

Bits 3, 4, 5 Physical drive number. This field is used to specify a physical drive number in the range 0..7. It is used in conjunction with I/O to the bootstrap and configuration spaces; it is ignored during accesses to the data space.

D.3 I/O Unit Specification

This section describes the standard system I/O units. The unit attribute determines the options available for use with the UNITREAD and UNITWRITE intrinsics. (see sections D.1 and D.2 for details; see sections 4.42 and 4.45 for parameter information). Unit-specific features are described next to the operations affected. The UNITSTATUS record format depends on the type of unit being polled. See section 3.9.2 for details.

NOTE - For reasons of compatibility with other implementations of UCSD Pascal, references to unit 128 are mapped into unit 3, and references to unit 129 are mapped into unit 21. The REMIN: (unit 7) and REMOUT: (unit 8) map to the same device as REMIN4: (unit 19) and REMOUT4: (unit 20).

#0:	- Device	- System Clock
	- Volume Name	- CLOCK:
	- Attribute	- Serial
	- UnitClear	- No action
	- UnitRead	- If BLOCKNUM >= 0, current task is suspended for the number of clock ticks specified by value in BLOCKNUM; next, if BYTES >= 4, the system time is read into the first two words of BUFF (least significant word first).
	- UnitWrite	- Stores first two words in BUFF into system time variable (least significant word first).
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-

#1:	- Device	- System console
	- Volume Name	- CONSOLE:
	- Attribute	- serial
	- UnitClear	- Clears type-ahead and UART buffers.
	- UnitRead	- Masks off high order bit, echoes input character, zero-fills remainder of BUFF instead of returning end-of-file marker.
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-

#2:	- Device	- System console
	- Volume Name	- SYSTERM:
	- Attribute	- Serial
	- UnitClear	- Clears type-ahead and UART buffers
	- UnitRead	- Masks off high order bit.
	- UnitWrite	-
	- UnitBusy	- Returns TRUE if no input character is available
	- UnitWait	- No action
	- UnitStatus	-
#3:	- Device	- System Console Type-ahead buffer
	- Volume Name	- KEYBUFR:
	- Attribute	- Serial
	- UnitClear	- Clears keyboard characters inserted by UNITWRITE(3).
	- UnitRead	- Bad mode
	- UnitWrite	- Writes characters into console type-ahead buffer in front of keyboard-queued characters.
	- UnitBusy	- Returns TRUE if type-ahead buffer is full.
	- UnitWait	- No action
	- UnitStatus	-
#4:	- Device	- Floppy drive 0
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Seek to track 0.
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#5:	- Device	- Floppy drive 1
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Seek to track 0.
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-

#6:
- Device - Parallel printer output (FFA4 hex)
- Volume Name - PRINTER:
- Attribute - Serial

- UnitClear -
- UnitRead - Bad mode
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#7:
- Device - DLV-11J port 3 input (FFB8 hex)
- Volume Name - REMIN:
- Attribute - Serial

- UnitClear - Clears REMIN: type-ahead queue
- UnitRead -
- UnitWrite - Bad mode
- UnitBusy - Returns TRUE if no input character is available.
- UnitWait - No action
- UnitStatus -

#8:
- Device - DLV-11J port 3 output (FFB8 hex)
- Volume Name - REMOUT:
- Attribute - Serial

- UnitClear - Clears REMIN: type-ahead queue
- UnitRead - Bad mode
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#9:
- Device - Optional hard disk virtual floppy 0
- Volume Name - user defined
- Attribute - Block-structured

- UnitClear - Indicate status
- UnitRead -
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#10:	- Device	- Optional hard disk virtual floppy 1
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#11:	- Device	- Optional hard disk virtual floppy 2
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#12:	- Device	- Optional hard disk virtual floppy 3
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#13:	- Device	- DLV-11J port 0 input (FEA0 hex)
	- Volume Name	- REMIN1:
	- Attribute	- Serial
	- UnitClear	- Clears REMIN1: type-ahead queue
	- UnitRead	-
	- UnitWrite	- Bad mode
	- UnitBusy	- Returns TRUE if no input character is available.
	- UnitWait	- No action.
	- UnitStatus	-

#14:
- Device - DLV-11J port 0 output (FEA0 hex)
- Volume Name - REMOUT1:
- Attribute - Serial

- UnitClear - Clears REMIN1: type-ahead queue
- UnitRead - Bad mode
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#15:
- Device - DLV-11J port 1 input (FEA4 hex)
- Volume Name - REMIN2:
- Attribute - Serial

- UnitClear - Clears REMIN2: type-ahead queue
- UnitRead -
- UnitWrite - Bad mode
- UnitBusy - Returns TRUE if no input character is available.
- UnitWait - No action
- UnitStatus -

#16:
- Device - DLV-11J port 1 output (FEA4 hex)
- Volume Name - REMOUT2:
- Attribute - Serial

- UnitClear - Clears REMIN2: type-ahead queue
- UnitRead - Bad mode
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#17:
- Device - DLV-11J port 2 input (FEA8 hex)
- Volume Name - REMIN3:
- Attribute - Serial

- UnitClear - Clears REMIN3: type-ahead queue
- UnitRead -
- UnitWrite - Bad mode
- UnitBusy - Returns TRUE if no input character is available.
- UnitWait - No action
- UnitStatus -

#18:	- Device	- DLV-11J port 2 output (FEA8 hex)
	- Volume Name	- REMOUT3:
	- Attribute	- Serial
	- UnitClear	- Clears REMIN3: type-ahead queue
	- UnitRead	- Bad mode
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#19:	- Device	- DLV-11J port 3 input (FFB8 hex)
	- Volume Name	- REMIN4:
	- Attribute	- Serial
	- UnitClear	- Clears REMIN4: type-ahead queue
	- UnitRead	-
	- UnitWrite	- Bad mode
	- UnitBusy	- Returns TRUE if no input character is available.
	- UnitWait	- No action
	- UnitStatus	-
#20:	- Device	- DLV-11J port 3 output (FFB8 hex)
	- Volume Name	- REMOUT4:
	- Attribute	- Serial
	- UnitClear	- Clears REMIN4: type-ahead queue
	- UnitRead	- Bad mode
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#21:	- Device	- System console
	- Volume Name	- FASTCON:
	- Attribute	- Serial
	- UnitClear	- Clear type-ahead and UART buffers
	- UnitRead	- Bad mode
	- UnitWrite	- Fast console output. No CR, FF, or DLE expansion.
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-

#22:
- Device - Standard input
- Volume Name - STANIN:
- Attribute - Serial

- UnitClear - No action
- UnitRead - Read from KEYBOARD file
- UnitWrite - Write to OUTPUT file
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#23:
- Device - Standard output
- Volume Name - STANOUT:
- Attribute - Serial

- UnitClear - No action
- UnitRead - Read from KEYBOARD file
- UnitWrite - Write to OUTPUT file
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#24:
- Device - Bit bucket
- Volume Name - BUCKET:
- Attribute - Serial

- UnitClear - No action
- UnitRead - Supply CHR(0) (e.g. EOF)
- UnitWrite - Ignore output
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#25:
- Device - Optional hard disk virtual floppy 4
- Volume Name - user defined
- Attribute - Block-structured

- UnitClear - Indicate status
- UnitRead -
- UnitWrite -
- UnitBusy - Returns FALSE
- UnitWait - No action
- UnitStatus -

#26:	- Device	- Optional hard disk virtual floppy 5
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#27:	- Device	- Optional hard disk virtual floppy 6
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-
#28:	- Device	- Optional hard disk virtual floppy 7
	- Volume Name	- user defined
	- Attribute	- Block-structured
	- UnitClear	- Indicate status
	- UnitRead	-
	- UnitWrite	-
	- UnitBusy	- Returns FALSE
	- UnitWait	- No action
	- UnitStatus	-

APPENDIX E: RESERVED WORDS

Standard Pascal Reserved Words

and	end	not	then
array	file	of	to
begin	for	or	type
case	function	packed	until
const	goto	procedure	var
div	if	program	while
do	in	record	with
downto	label	repeat	
else	mod	set	

NOTE - NIL is a predefined identifier in UCSD Pascal.

UCSD Pascal Reserved Words

forward
interface
implementation
process
segment
separate
unit
uses

APPENDIX F: PREDECLARED IDENTIFIERS

Standard Pascal Predeclared Identifiers

abs	false	page	sqr
arctan	get	pred	sqrt
boolean	input	put	succ
char	integer	read	text
chr	ln	readln	true
cos	maxint	real	trunc
dispose	new	reset	write
eof	odd	rewrite	writeln
eoln	ord	round	
exp	output	sin	

UCSD Pascal Predeclared Identifiers

atan	ioresult	release	unitread
attach	keyboard	rmemavail	unitstatus
blockread	length	scan	unitwait
blockwrite	mark	seek	unitwrite
close	memavail	semaphore	varavail
concat	memlock	seminit	vardispose
copy	memswap	signal	varnew
delete	moveleft	sizeof	wait
exit	moveright	start	
fillchar	nil	str	
gotoxy	opennew	string	
halt	openold	time	
idsearch	pos	treesearch	
insert	processid	unitbusy	
interactive	pwroften	unitclear	

NOTE - NIL is a reserved word in standard Pascal.

APPENDIX G: IMPLEMENTATION LIMITS

Maximum number of segments in a program: 128

Maximum number of procedures in a segment: 255

Maximum level of nested procedures: 8

Maximum level of nested statements: 12

Maximum size of a procedure: 1200 bytes

Maximum size of variables in a procedure: 32766 words

Maximum size of a record or array: 32766 words

Maximum size of a set: 4080 elements

Maximum size of a string: 255 characters

Integer range: -32768 .. 32767 (no overflow checking)

Long integer accuracy: up to 36 digits

Real range: -3.0E38 .. 3.0E38 (approximate)

Real accuracy: up to 6 significant digits

Sets

An integer subrange type encompassing negative integer values may not be used as the base type of a set in UCSD Pascal. "Negative" sets compile successfully, but cause execution error 1 ("Value range error") when they are assigned negative values.

Example of an invalid set:

```
program revelation;
var nuclear: set of -66..6;
    solar: set of 3..33;
begin
  solar := [5];
  nuclear := [-30];  { program crashes here }
end.
```

Mixed Expression Evaluation

The lack of integer overflow checking can affect expressions mixing integers with long integers or reals. The compiler evaluates mixed expressions left-to-right; the expression is evaluated with integer operations until either an operand of the final type (long integer or real) is encountered or the end of the expression is reached. Only at this point does the compiler convert the expression (sub)result to the final type; however, the integer-valued expression may have already overflowed.

Example of mixed expression misevaluation:

```
program mal;
var I: integer;
    R: real;
begin
  I := 20000;
  R := 3.0;
  writeln(I + 20000 + R);
```

In this example, the compiler emits code to perform an integer addition of the integer variable I and the integer constant 20000. The integer result is then converted to type real and added to the real variable to obtain the expression result. Unfortunately, the integer addition overflows, resulting in an incorrect integer subresult; the error is merely propogated by the subsequent real operations.

This problem can be avoided by reordering expressions so that real or long integer operands precede the integer operands; this forces the compiler to convert integer operands to the final type as they are encountered.

NIL Pointer References

UCSD Pascal does not detect dynamic variable references through pointer variables containing the value NIL (these should be flagged with execution error 7, but are not).

Record Variant Accesses

UCSD Pascal provides no checks for the detection of invalid record variant references (i.e. accessing a record variant which doesn't correspond with the tag field value).

FOR Statements

FOR statements with a final value of MAXINT become infinite loops. Avoid using MAXINT (and -MAXINT) as the initial and final values.

CASE Statements

CASE statements in UCSD Pascal are implemented with a jump table; the table size is dependent on the range of values spanned by the case labels. Consider the following CASE statement:

```
case int of
 -1000:  kind := loss;
  1000:  kind := profit;
end {case};
```

The jump table generated by this statement would be 2000 words long (!!); however, the compiler's limit on procedure size prevents large CASE statements such as this one from compiling successfully. When confronted with situations of this type, it is more efficient (with respect to code size) to use IF statements for detecting extreme values, and save CASE statements for relatively small ranges of values (e.g. CHAR, enumerated types, and modest integer subranges).

Special Symbols

Some of the special symbols in UCSD Pascal are internally equivalent; they may be substituted for each other without affecting the compilability of a program.

SEGMENT is equivalent to PROGRAM

: is equivalent to ..

MOD and DIV with Negative Arguments

The result of a MOD or DIV operation involving negative arguments differs between implementations of UCSD Pascal. The result of a DIV operation with positive arguments is always truncated. When using negative operands, some processors round the result of a DIV towards the larger integer (less negative); some processors round towards the smaller (more negative). Since "a MOD b" is defined to be "a - (a DIV b) * b", the values returned by MOD are affected by the result of DIV.

The PDQ-3 rounds the result of a DIV using a negative operand towards the larger integer. For example, -3 DIV 2 results in -1.

NOTE - On the PDQ-3, if the second operand of a MOD calculation is negative, a non-suppressable range check execution error occurs.

APPENDIX H: COMPILER SYNTAX ERRORS

```
  1:  Error in simple type
  2:  Identifier expected
  3:  'PROGRAM' expected
  4:  ')' expected
  5:  ':' expected
  6:  Illegal symbol (maybe missing ';' on the line above)
  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 expected
 16:  '=' expected
 17:  'BEGIN' expected
 18:  Error in declaration part
 19:  error in <field-list>
 20:  ',' expected
 21:  '.' expected
 22:  'INTERFACE' expected
 23:  'IMPLEMENTATION' expected
 24:  'UNIT' expected

 50:  Error in constant
 51:  ':=' expected
 52:  'THEN' expected
 53:  'UNTIL' expected
 54:  'DO' expected
 55:  'TO' or 'DOWNTO' expected in for statement
 56:  'IF' expected
 57:  'FILE' expected
 58:  Error in <factor> (bad expression)
 59:  Error in variable
 60:  Must be semaphore
 61:  Must be processid

101:  Identifier declared twice
102:  Low bound exceeds high bound
103:  Identifier is not of the appropriate class
104:  Undeclared identifier
105:  sign not allowed
106:  Number expected
107:  Incompatible subrange types
108:  File not allowed here
109:  Type must not be real
110:  <tagfield> type must be scalar or subrange
111:  Incompatible with <tagfield> part
112:  Index type must not be real
113:  Index type must be a scalar or a subrange
114:  Base type must not be real
```

115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisified forward reference
118: Forward reference type identifier in variable declaration
119: Re-specified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: Forward declared function result type can't be re-specified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substition of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected

171: Standard file was re-declared
172: Undeclared external file
173: Fortran procedure or function expected!
174: Pascal function or procedure expected
175: Semaphore value parameter not allowed
182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration
189: 'USES' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'USES' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable
196: 'USES LONGINT' required

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number

250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range

398: Implementation restriction
399: Implementation restriction
400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: Call not allowed in separate procedure
406: Include file not legal
407: disk error
408: compiler error

APPENDIX I: ASCII CHARACTER SET

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	040	21	!	65	101	41	A	97	141	64	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	78	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	064	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	89	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	307	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

APPENDIX J: DIFFERENCES BETWEEN UCSD VERSIONS

This section describes differences between versions II, III, IV and AOS 1.0 of the UCSD Pascal system. Executable code files are not transportable across versions; however, UCSD Pascal programs are generally source compatible across versions (i.e. they may be recompiled without changes to run on a different version). Source incompatibilities result from changes in some of the UCSD Pascal extensions; programs written in the UCSD variant of standard Pascal are completely source compatible across versions.

Concurrency

Concurrency is not included in version II UCSD Pascal; it exists only in versions III and IV. Concurrency in versions III and IV is identical. Time delays are unique to the versions III.1 and the AOS on the PDQ-3.

NOT

The NOT operator in versions II and IV returns the full-word logical negation of its operand. NOT in version III (releases H.0 and beyond) returns the Boolean negation of its operand (i.e. the low order bit of the operand is negated, but the high order 15 bits of the result are zeroed). The AOS default evaluation matches versions II and IV. Boolean negation may be selected using the $J compile option described in section 5.0.13.

Long Integers

In version III, programs containing long integers must use the system unit named LONGINT. This is unnecessary in versions II, IV and the AOS.

Transcendental Functions

In some version II systems, programs calling the transcendental functions must use the system unit named TRANSCEND. This is not necessary in versions III, IV and the AOS.

Segments

Some versions of II allow a program to contain up to 8 segments. Version IV allows up to 255 segments. The AOS allows up to 128 segments (not including intrinsic units).

Units

Separate units are unique to version II. Data units (i.e. units containing only an interface section of types and variables) are allowed in versions II.1, III.1 and the AOS. Version IV and AOS units may contain segment procedures, files, and termination sections. Versions IV, II.1, the AOS, and some releases of version II allow intialization sections.

Treesearch

The ordering of trees built by TREESEARCH is implementation dependent and varies across machines. TREESEARCH itself works correctly on all systems; only manual tree traversals are affected by this property.

Intrinsics

The intrinsics PMACHINE, ATTACH, SEMINIT, SIGNAL, START, and WAIT are present in versions III, IV and the AOS. Version IV and the AOS allow the memory management intrinsics MEMLOCK, MEMSWAP, DISPOSE, VARNEW, VARDISPOSE, and VARAVAIL. Note that the file intrinsics OPENOLD and OPENNEW are not present in version IV. Version III and the AOS allow the memory management intrinsic RMEMAVAIL. Versions IV, the AOS, and some versions of II.0 allow the Unit I/O UNITSTATUS intrinsic. The I/O redirection intrinsic REDIRECT is provided only in version IV.

The value of the UNITBUSY boolean function in version III systems distributed by Western Digital differs with values returned on other UCSD Pascal systems. The UNITBUSY function value should be negated when transferring software between Western Digital version III systems and others.

I/O Redirection

I/O redirection is provided in version IV and the AOS. The PI=, PO=, P=, and L= options are supported in both versions. The PP=, PL=, TO=, PTO=, TI=, and PTI= options are supported only by the AOS. The O= and I= options are supported in version IV. The AOS maps these to the PO= and PI options. The O= and I= options may be simulated in the AOS by re-executing the system shell using the PO= and PI= options.

Pointer Comparison

Extended pointer comparison exists in versions III, IV, and the AOS.

Procedure Size

The restrictions on procedure size are greatly relaxed in version IV. This can affect the transportability of programs developed on version IV.

APPENDIX K: AOS LIBRARY UNITS

This section lists library units available for use by programs running under the AOS. Some units are provided with the AOS; others may be purchased separately. Each unit listing contains a brief description of the unit and a list of its interface routines. Complete documentation is available in the Library User's Manual.

PROGOPS

The PROGOPS unit contains routines useful in the invocation and termination of programs. It is provided in the standard AOS release and resides in the intrinsics library.

The interface routines are:

Prog_Call	Setup and execute program
Prog_Setup	Prepare a program for execution
Prog_Execute	Execute program with I/O redirection
Prog_Start	Execute program (no I/O redirection)
Prog_Redir	Indicate current I/O redirection
Prog_Exception	Cause execution error
Prog_IO_Set	Cause I/O error

COMMANDIO

The COMMANDIO unit contains routines useful in CHAINing and I/O redirection control. It is provided in the standard AOS release and resides in the system library.

The interface routines are:

Chain	Chain another program
Exception	Flag exceptional condition
Suspend_T	Suspend T-file processing
Resume_T	Resume T-file processing
Suspend_Redir	Suspend I/O redirection
Resume_Redir	Resume I/O redirection

EXCEPINFO

The EXCEPINFO unit contains routines useful in translating execution and I/O errors numbers into a text form. It is provided in the standard AOS release and resides in the system support library.

The interface routines are:

Ex_Stats	Convert runtime info to listing info
Ex_IO_Err_Name	Translate an I/O error into text
Ex_Err_Name	Translate an error into text

SPOOLER

The SPOOLER unit contains routines useful in initiating and controlling a printer spooler. It is provided in the standard AOS release and resides in the system library.

The interface routines are:

Spool_File	Spool a file to a serial unit
Spool_Status	Return the spooler status
Spool_Restart	Restart the spooler
Spool_Stop	Stop the spooler

SYSUTIL

The SYSUTIL unit contains miscellaneous routines useful in reading the system serial number, performing time delays, and other system-oriented functions. It is provided in the standard AOS release and resides in the system library.

The interface routines are:

Ser_Num	Get system serial number
Time_Delay	Delay for a specified time

PATTERNMATCH

The PATTERNMATCH unit contains pattern matching routines capable of matching multiple wildcards, character ranges, and literals. It is provided as an optional system library unit.

The interface routine is:

P_Match	Compare a wildcard and source string

DIRINFO

The DIRINFO unit contains routines useful in the manipulation of the file system. Directory list, name change, file deletion, and date access routines are provided. DIRINFO uses the PATTERNMATCH unit for added flexibility in file name specification. It is provided as an optional system library unit.

The interface routines are:

D_Dir_List	Build a list of directory entries
D_Scan_Title	Parse a file name
D_Change_Name	Change a file name
D_Change_Date	Change date on a set of file entries
D_Rem_Files	Remove a set of files
D_Init	Initialize DIRINFO
D_Lock	Lock file system
D_Release	Release file system

SCCNTRL

The SCCNTRL unit contains screen control routines capable of operating in a multi-terminal, multi-processing environment. Text port and prompt line support are also provided. It is provided as an optional system library unit.

The interface routines are:

SC_Init	Initialize a text port
SC_New_Port	Declare a new text port
SC_Out_Lock	Lock a text port output channel
SC_Out_Release	Release a text port output channel
SC_In_Lock	Lock a text port input channel
SC_In_Release	Release a text port input channel
SC_Scrn_Has	Query screen capabilities
SC_Left	Move the cursor left
SC_Right	Move the cursor right
SC_Up	Move the cursor up
SC_Down	Move the cursor down
SC_Home	Move the cursor home
SC_Goto_XY	Move the cursor
SC_Clr_Line	Clear a line
SC_Erase_To_Eol	Erase to the end of a line
SC_Clr_Screen	Clear a screen
SC_Eras_Eos	Erase to the end of a screen
SC_Has_Key	Query keyboard capabilities
SC_Map_Crt_Command	Read a key from a keyboard
SC_Prompt	Display a prompt

NUMCON

The NUMCON unit contains routines useful in integer manipulations, including unsigned comparisons, min, max, and conversions of integers to strings and vice versa. It is provided as an optional system library unit.

The interface routines are:

N_Str_To_Int	Convert a string to an integer
N_Int_To_Str	Convert an integer to a string
N_Uns_To_Str	Convert an unsigned int to a string
N_Min	Return the smaller of two integers
N_Max	Return the larger of two integers
N_Leq_U	Unsigned <= comparison
N_Geq_U	Unsigned >= comparison
N_Min_U	Unsigned Min
N_Max_U	Unsigned Max

REALCON

The REALCON unit contains routines useful in real manipulations, including conversions of reals to strings and visa-versa. It is provided as an optional system library unit.

The interface routines are:

R_Str_To_Real	Convert a string to a real
R_Real_To_Str	Convert a real to a string
R_Min	Return the smaller of two reals
R_Max	Return the larger of two reals

FILEINFO

The FILEINFO unit contains routines providing information on file variables. It is provided as an optional system library unit.

The interface routines are:

F_Length	Get length of a file
F_Unit	Get unit number containing a file
F_Volume	Get volume containing a file
F_File_Title	Get title of a file
F_Start	Get starting block of a file
F_Is_Blocked	Get type of unit containing a file
F_Date	Get last access date of a file

SYSINFO

The SYSINFO unit contains routines providing access to the system workfile, date, and file prefix variables. It is provided as an optional system library unit.

The interface routines are:

SI_Code_Vid	Get name of vol holding workfile code
SI_Code_Tid	Get name of file holding workfile code
SI_Text_Vid	Get name of vol holding workfile text
SI_Text_Tid	Get name of file holding workfile text
SI_Sys_Unit	Get number of bootload unit
SI_Get_Sys_Vol	Get system volume name
SI_Get_Pref_Vol	Get prefix volume name
SI_Set_Pref_Vol	Set prefix volume name
SI_Get_Date	Get current system date
SI_Set_Date	Set current system date

ADDENDA FOR THE PROGRAMMER'S MANUAL

Section 3.0.0 (page 15)

As stated, the system prevents a program from terminating until all of its tasks have terminated. This is implemented by recording the number of tasks active in the system before the program executes. The program is prevented from terminating until the number of active tasks matches the original count. Unfortunately, if a program starts a task then calls another program, which also starts a task. The called program is erroneously allowed to terminate if the calling program's task terminates, but the called program's task does not.

Appendix D (page 217)

When the UnitStatus intrinsic is called for the STANIN: device (unit 22) it returns the CharsQueued and QueueSize fields appropriate for the device currently supplying the input stream. If this device is a disk file or a literal string, both fields are returned zero.

Appendix K (page 238)

The SPOOLER unit is actually called SPOOLUNIT.