Advanced Computer Design

AOS Library User's Manual

AOS LIBRARY USER'S MANUAL

VERSION 1.0

June 1982

Advanced Computer Design

Author: Barry Demchak

Document:

TABLE OF CONTENTS

I. INTRODUCTION

1.0 Scope of this Manual

This manual describes both the standard and the optional library modules available from Advanced Computer Design for use with the the UCSD Pascal Advanced Operating System (AOS) version 1.0.

Users are assumed to have a working knowledge of UCSD Pascal. Of particular importance are the library and separate compilation facilities. This manual does not describe these facilities; they are fully described in the following manuals:

AOS System User's Manual - Describes the UCSD Pascal system software (including the library system and the operation of the Pascal compiler).

AOS Programmer's Manual - Describes the UCSD Pascal language (including the separate compilation facilities).

Other documents of interest include:

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

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

NOTE - This manual describes the library modules available from Advanced Computer Design at the time of printing. Addenda will be provided as new modules are developed.

NOTE - The modules described herein are predominantly system-oriented; a host of application-oriented library modules compatible with the Advanced Operating System are available from other sources.

1.1 Overview

The UCSD Pascal System was designed for use as a simple program development system rather than a general purpose operating system; consequently, much useful system information is inaccessible to user programs. The most common solution to this problem is to structure user programs so they can directly access the operating system's data structures. A better solution is to construct UCSD Pascal units which provide routines for accessing system information in a controlled manner. The interfaces provided by well-designed units make system information access a much safer and easier operation to perform.

Program development using units is faster and more reliable than traditional methods. Large and complex collections of routines may be organized into unit libraries for subsequent incorporation into user programs. Valuable programmer time is conserved by eliminating the re-creation of pre-existing routines.

Units available from Advanced Computer Design are organized into five classifications:

System Functions - Includes program execution from a host program, program chaining, I/O redirection control, system variable access, and miscellaneous system functions.

Screen Formatting - Includes screen and keyboard primitives for multiterminal and multiprocessing use. All screen commands are supported within user-defined text windows.

Data Manipulation - Includes wildcard pattern matching, integer and real format conversions, and miscellaneous numerical operators.

File System Manipulation - Includes extended access to both external and internal files. Routines are provided that read directories, remove files, and change file names, dates, and lengths.

I/O Routines - Includes an asynchronous printer spooler.

Chapter 2 describes the system functions, including the Prog_Ops, Excep_Info, Command_IO, Sys_Info, and Sys_Util units. Chapter 3 describes the screen formatting unit, SC_Cntrl. Chapter 4 describes the data manipulation units, including the Pattern_Match, Num_Con, and Real_Con units. Chapter 5 describes the file system manipulation units, including the Dir_Info and File_Info units. Chapter 6 describes the print spooler unit, Spool_Unit.

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 identifiers are printed beginning with Capital Letters, while standard terms are underlined.

The following terms are used in the description of the UCSD Pascal System: intrinsics library, system library, user library, system support library, and drivers library. These refer to the library system described in the System User's Manual.

II. SYSTEM FUNCTIONS

This chapter describes units which perform functions related to program execution, operating system variables, and I/O redirection.

The program operators unit (section 2.0) and the exception information unit (section 2.1) are normally used by the operating system, but may be also used by user programs. The program operators unit allows the execution of a program as a procedure of the host. The exception information unit translates execution error and I/O error numbers into error text. The command I/O unit (section 2.2) provides routines related to program chaining and I/O redirection. The system information unit (section 2.3) allows access to operating system workfile, file system, and date variables. The system utility unit (section 2.4) provides miscellaneous routines.

2.0 The Program Operators Unit

The program operators unit (called Prog_Ops) provides program invocation and termination routines normally available only to the operating system. Using the facilities of the program operators unit, a program may execute other programs as if they were procedures.

The program operators unit provides the following capabilities:

Program Invocation - Setup and execute a program.

Raise Exception - Invoke the system execution error processor.

I/O Redirection Status - Indicate whether the standard I/O stream for the current program has been modified.

The Prog_Ops routines are described in section 2.0.0. Section 2.0.1 contains a copy of the Prog_Ops unit interface section. A programming example using the Prog_Ops unit appears in section 2.0.2.

NOTE - Use of the Prog_Ops unit requires familiarity with the system I/O redirection facilities. See the System User's Manual and the Programmer's Manual for details.

2.0.0 Using the Program Operators Unit

The program operators unit is provided with all AOS releases. It is imported into a program by the "USES Prog_Ops;" statement (see the Programmer's Manual for details). Prog_Ops must be installed in the intrinsics library. All imported identifiers begin with "Prog_" to prevent conflicts with program identifiers. The routines provided by this unit are described below.

The program operators unit maintains no global variables and uses no other units.

2.0.0.0 Program Invocation

The program operators unit contains four routines used for program invocation. The Prog_Call routine may be called to setup and execute a given program. The Prog_Setup routine prepares a program for subsequent execution. The Prog_Start and Prog_Execute routines execute a program that has already been prepared by the Prog_Setup routine. These routines may be called from anywhere within a user program. Programs invoked by calls to the program operators execute as if they were procedures of the calling program.

NOTE - Code segments and global data spaces for units used by programs are not shared among nested program invocations unless the units are installed in the intrinsics library.

WARNING - The program invocation routines are not designed for use in tasks or in multitasking programs. These routines may not be used to implement a multiuser or multiprogramming system.

2.0.0.0.0 Program Specifiers

When calling the Prog_Call or Prog_Setup routines, a program is specified by an execution option list. It may contain either the name of a code file containing a program, a list of I/O redirection options, or both. The execution option list is formatted and interpreted in the same way as an input to the system eX(ecute command (see the System User's Manual). Examples of execution option lists include:

```
Backup
Format I="4,ssyy"
System.Compiler.  I="Test,$," O=Bucket:
Prose I=File.Names.Text,",,82/05/22" O=List.Text TO=#1:
```

When calling the Prog_Start and Prog_Execute routines, a program is specified by a program descriptor record. This record is initialized by the Prog_Setup routine and contains all information necessary to process an execution option list passed to Prog_Setup. The declaration for a program descriptor is presented below:

```
Prog_Rec = Record
             Heap_Base    : ^Integer;
             User_Vect    : ^Integer;
             Unit_List    : ^Integer;
             Redir_Rec    : ^Prog_Red_Rec;
           End {of Prog_Rec};
```

A program descriptor refers to an area on the system heap containing the structures necessary to execute a program. A pointer to the base of these structures is contained in Heap_Base. The User_Vect and Unit_List fields point to the program runtime structures. The Redir_Rec field points to a list of I/O redirection options to be processed before program execution. If no I/O redirection options were specified in the original execution option list, this field contains Nil.

NOTE - I/O redirection options are effective only when applied to programs that perform I/O through the predeclared INPUT and OUTPUT files. Unit I/O and I/O performed on local file variables are not affected by redirection options (i.e. since the system editor performs unit I/O, editor input must come from the keyboard rather than from an alternate input file). See the System User's Manual and the Programmer's Manual for further details.

2.0.0.0.1 Prog_Call

The Prog_Call function attempts to set up and execute a program.

It returns a boolean function result indicating whether or not the program was executed. Prog_Call accepts two variable strings and one variable integer as parameters. The first string contains an execution option list. If the code file named in the execution option list does not exist or not all of the units used by the program are available, a textual error message is returned in the second string parameter and the function result is returned FALSE. Otherwise, the program is executed and the function result is returned TRUE. Any execution error causing program termination is returned in the integer parameter; 0 is returned for normal program termination.

NOTE - The string variable used to pass the execution option list may also be used to return the textual error message. It is not necessary to declare two string variables for use as parameters to Prog_Call.

2.0.0.0.2 Prog_Setup, Prog_Execute, and Prog_Start

The Prog_Setup segment function is called to set up the runtime structures necessary to execute a program. It accepts a string value, a pointer value, a string variable, and a program descriptor variable as parameters. The first string contains an execution option list. If the program named in the execution option list does not exist or not all of its used units are available, a textual error message is returned in the second string parameter and the function result is returned FALSE. If there is no code file name in the execution list, any I/O redirection options are processed, the second string parameter is returned empty, and the function result is returned FALSE. Otherwise, the program's runtime structures are created on the system heap, the program descriptor is returned, and the function result is returned TRUE.

The program runtime structures include the global variable space for each unit used by the program, and 20 words for each segment or unit declared. No program or unit code segments are loaded. If the value of the pointer parameter is Nil, the structures are created on the top of the current heap. Otherwise, the RELEASE intrinsic is called using the value of the po[illegible] argument, then the structures are created on the top of the resulting heap. This is useful in recyling the heap space occupied by the runtime structures of programs previously processed by Prog_Setup.

Either the Prog_Start function or the Prog_Execute segment function may be used to execute a program already setup by the Prog_Setup routine. Both functions accept a program descriptor as a parameter. They execute the program associated with it. The Prog_Execute function processes any I/O redirection options before program execution; the Prog_Start function does not. The function result is returned containing the number of any execution error that terminates the program; 0 is returned if no execution error is encountered.

The Prog_Start function may be used when the Redir_Rec field of the program descriptor contains Nil. The Prog_Execute segment function

should be used when this field is non-Nil. Using Prog_Start whenever possible avoids the unnecessary allocation of memory space to code that processes I/O redirection directives when no I/O redirection options are specified.

The Prog_Setup function may used in conjunction with Prog_Start or Prog_Execute to minimize the time required to repeatedly call a program. Calling the Prog_Call function for each execution results in significantly more setup overhead than calling Prog_Setup once and calling Prog_Execute or Prog_Start for each execution. In cases where the Prog_Execute is called repeatedly, execution overhead may be reduced by using the $R compile option or the MEMLOCK intrinsic (see the Programmer's Manual) to make the Prog_Execute segment memory-resident.

NOTE - The P=, PP=, L= and PL= I/O redirection options are executed by both the Prog_Setup and the Prog_Execute functions. These functions set the prefix volume according to any P= or PP= options (PP= takes precedence over P=) and then process the rest of the execution option list. Before the functions terminate, the prefix volume is set according to any P= option that may occur in the execution option list; it is restored to its original value of no P= option exists. The L= and PL= options are processed after the prefix options, and follow the same rules. The additional variable space required to implement the O=, PO=, I=, PI=, TO=, PTO=, TI=, and PTI= options is not allocated until the Prog_Execute function is called. See the System User's Manual for further details.

WARNING - Program descriptors referencing a deallocated heap space must not be used. Fatal system crashes may result.

2.0.0.1 Program Termination

Abnormal program termination occurs as the result of an execution error. Execution errors are normally detected by either the operating system or the processor. The Prog_Exception and Prog_IO_Set procedures are used to flag execution errors in user-defined circumstances.

The Prog_Exception procedure accepts an integer value as a parameter. It uses the integer parameter as the execution error number in a call to the system exception handler. The system then behaves as if an execution error of that type had occurred.

The Prog_IO_Set procedure accepts an integer value as a parameter. It sets the value of the IORESULT function (see the Programmer's Manual) to the value of the integer parameter. An I/O execution error is programmed by calling the Prog_IO_Set procedure immediately before a Prog_Exception (10 {I/O error}) call.

The program termination routines may be used i n conjunction with modifications to the exception information unit (section 2.1) to define and implement user-defined execution and I/O errors.

2.0.0.2 I/O Redirection

The Prog_Redir function indicates whether I/O redirection options are in effect for either the OUTPUT or INPUT files. The function accepts a boolean parameter whose value is TRUE for information on output redirection, and FALSE for information on input redirection. The value of the function is returned TRUE if I/O redirection is in effect for the specified channel; otherwise, the function value is returned FALSE.

This routine is useful in implementing the system GOTOXY intrinsic (see the System User's Manual). Normally, a cursor positioning sequence is written to the system console through the FASTCON: device (unit 21). If Prog_Redir(TRUE) reveals that console output has been redirected, the cursor positioning sequence should be written to the STANOUT: device (unit 22) instead.

2.0.1 The Program Operators Unit Interface

This section displays the text of the interface section belonging to the program operators unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "Prog_" to prevent conflicts with host program identifiers.

NOTE - This unit is normally used by the operating system to perform system functions. Not all parts of the unit interface are explained in this document. Users may experiment with these parts at their own risk.

```
Unit Prog_Ops;
Interface

Type Prog_String  = String[255];
     Prog_Ptr     = ^Integer;
     Prog_Str_P   = ^Prog_Str_Rec;
     Prog_Str_Rec = Record
                      Next_Str   : Prog_Str_P;
                      Is_T_File,
                      Is_Literal : Boolean;
                      Name       : String;
                    End {of Prog_Str_Rec};
     Prog_Red_P   = ^Prog_Red_Rec;
     Prog_Red_Rec = Record
                      In_File_Name,
                      Out_File_Name,
                      Prog_Px_Name,
                      Px_Name,
                      Prog_Lib_Name,
                      Lib_Name        : Prog_Str_P;
                    End {of Prog_Red_Rec};
     Prog_Rec     = Record
                      Heap_Base    : Prog_Ptr;
                      User_Vect    : Prog_Ptr;
                      Unit_List    : Prog_Ptr;
                      Redir_Rec    : Prog_Red_P;
                    End {of Prog_Rec};

Segment Function Prog_Setup (File_Name : Prog_String;
                             Heap_Dest : Prog_Ptr;
                         Var Error     : String;
                         Var Prog_Desc : Prog_Rec) : Boolean;
  {Create runtime structures for the program File_Name}

Segment Function Prog_Execute (Prog_Desc : Prog_Rec)
                                                  : Integer;
  {Execute the program described by Prog_Desc}

Function Prog_Start (Prog_Desc : Prog_Rec) : Integer;
  {Execute the program described by Prog_Desc.  Ignore
   I/O redirection}
```

```
Function Prog_Call (Var File_Name : Prog_String;
                    Var Error     : String;
                    Var Err_Num   : Integer) : Boolean;
  {Call the program File_Name}

Function Prog_Redir (Out_Direction : Boolean) : Boolean;
  {Return the status of I/O redirection on a given channel}

Procedure Prog_Exception (Err_Num : Integer);
  {Cause a specified execution error}

Procedure Prog_IO_Set (IO_Err_Num : Integer);
  {Set the value of the system IORESULT intrinsic}
```

2.0.2 Programming Example

The following program demonstrates the capabilities of the program operators unit; it implements a shell (described in the Programmer's Manual). Another example appears in chapter 7 of the Programmer's Manual.

```
Program Prog_Test;
Uses Prog_Ops;
Const FF = 12;
Var Ch : Char;
    File_Name,
    Error : String;
    Trash : Integer;
    Desc_Array : Array ['A'..'Z'] Of ^Prog_Rec;

  Procedure Setup_Progs;
  Var Ch : Char;

    Procedure Do_One (Ch : Char; Name : String);
    Var Temp_Desc : Prog_Rec;
    Begin
      If Prog_Setup (Name, Nil, Error, Temp_Desc) Then
        Begin
          New (Desc_Array[Ch]);
          Desc_Array[Ch]^ := Temp_Desc;
        End {of If};
    End {of Do_One};

  Begin {$R Prog_Setup}
    For Ch := 'A' To 'Z' Do
      Desc_Array[Ch] := Nil;
    Do_One ('C', '*System.Compiler.');
    Do_One ('F', '*System.Filer.');
    Do_One ('S', '*X');
    Do_One ('E', '*System.Editor.');
  End {of Setup_Progs};

  Procedure Prompt (Var Ch : Char);
  Begin
    Write (Chr(FF),'Command: C(ompile, F(iler, S(ubmit, ',
           'E(dit, eX(ecute, H(alt ');
    Repeat
      Read (Keyboard, Ch);
      If Ch >= 'a' Then
        Ch := Chr (Ord (Ch) - Ord ('a') + Ord ('A'));
    Until Ch in ['C', 'F', 'S', 'E', 'X', 'H'];
    Write (Chr(FF));
    If Ch <> 'X' Then
      Writeln;
  End {of Prompt};
```

```
Begin {of Prog_Test}
  Setup_Progs;
  Prompt (Ch);
  While Ch <> 'H' Do
    Begin
      Case Ch Of
        'C',
        'F',
        'S',
        'E' : Trash := Prog_Start (Desc_Array[Ch]^);
        'X' : Begin
                Write ('Execute what file? ');
                Readln (File_Name);
                If Not Prog_Call (File_Name,
                                  Error, Trash) Then
                  Begin
                    Write (Error, '; type <return>');
                    Readln;
                  End {of If};
              End {of 'X'};
      End {of Case};
      Prompt (Ch);
    End {of While};
End.
```

2.1 The Exception Information Unit

The exception information unit (called Excep_Info) allows user programs access to system exception handling information normally available only to the system exception handler.

The exception information unit provides the following capabilities:

Error Translations - Translation of execution error and I/O error numbers into textual format.

Segment Decoding - Translation of segment numbers and instruction counters into segment names and procedure offsets.

User-defined execution and I/O errors may be added to the Pascal system by modifying the Excep_Info unit. The Excep_Info unit source is available from ACD.

The Excep_Info routines are described in section 2.1.0. Section 2.1.1 contains a copy of the Excep_Info unit interface section. A programming example using the Excep_Info unit appears in section 2.1.2.

2.1.0 Using the Exception Information Unit

The exception information unit is imported into a program by the "USES Excep_Info;" statement (see the Programmer's Manual for details). All imported identifiers begin with "Ex_" to prevent conflicts with program identifiers. The routines provided by this unit are described below.

This unit normally exists without an interface section in the system support library. A copy of the unit with an interface section is available in the EXCEP.INFO.CODE file provided on the AOS release disk. The copy resident in the system support library must be removed, and the copy resident in the EXCEP.INFO.CODE file must be transferred to the intrinsics library (using the Library utility, see the System User's Manual) before it may be used by a user program.

The system utility unit maintains no global variables and uses no other units.

2.1.0.0 Segment Decoding

The Ex_Stats procedure translates runtime segment execution information into a form that can be used in conjunction with program listings. It accepts two integer value parameters containing a segment number and a byte-offset into the segment. It translates the segment number into a segment name, and the segment-relative byte-offset into a procedure number and a procedure-relative byte-offset. The segment name, procedure number, and byte-offset are returned in the remaining three parameters.

This procedure is useful in printing status and diagnostic reports relating to the progress of program execution. Segment numbers and byte-offsets are available in P-Machine mark stack records. See the Architecture Guide for details.

2.1.0.1 I/O Error Name

The Ex_IO_Err_Name segment procedure accepts an integer value and a string variable as parameters. It translates an I/O error number contained in the integer parameter to a text string describing the error. The textual form is returned in the string parameter.

This procedure is useful in providing a meaningful user interface in programs that perform their own I/O checking. Such programs may use this procedure to translate a bad IORESULT into text before reporting an I/O error to the user.

NOTE - The system exception handler calls Ex_IO_Err_Name when constructing the excecution error message for an I/O error. User-defined I/O errors may be added to the Pascal system by modifying this procedure to return text for currently unallocated error numbers. These I/O errors may be generated by user-provided system drivers and by use of the Prog_IO_Set procedure of the program operators unit (see section 2.0).

2.1.0.2 Execution Error Name

The Ex_Err_Name segment procedure accepts two integer values and a string variable as parameters. It translates an I/O error number and an execution error number contained in the integer parameters to a text string describing the error. The textual form is returned in the string parameter. The I/O error number parameter is meaningful only when the execution error number is 10 (User I/O Error); otherwise, it is ignored.

NOTE - The system exception handler calls Ex_Err_Name when constructing the excecution error message for an execution error. User-defined execution errors may be added to the Pascal system by modifying this procedure to return text for currently unallocated error numbers. These execution errors may be generated by use of the Prog_Exception procedure of the program operators unit (see section 2.0).

2.1.1 The Exception Information Unit Interface

This section displays the text of the interface section belonging to the exception information unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "Ex_" to prevent conflicts with host program identifiers.

```
Unit Excep_Info;
Interface

  Type Ex_Alpha = Packed Array [1..8] Of Char;

  Procedure Ex_Stats (Seg_Num,
                      Seg_IPC  : Integer;
                  Var Seg_Name : Ex_Alpha;
                  Var Proc_Num,
                      IPC      : Integer);
    {Translate runtime information into compiled listing form}

  Segment Procedure Ex_IO_Err_Name (IO_Err : Integer;
                                    Var S      : String);
    {Return text describing an I/O error number}

  Segment Procedure Ex_Err_Name (Xeq_Err,
                                 IO_Err : Integer;
                             Var S      : String);
    {Return text describing an execution error number}
```

2.1.2 Programming Example

The following program demonstrates the capabilities of the exception information unit.

```
Program Excep_Test;
Uses Excep_Info;
Var I : Integer;
    S : String;
Begin
  For I := 0 To 255 Do
    Begin
      Unitclear (I);
      If IO_Result <> 0 Then
        Ex_IO_Err_Name (IO_Result, S)
      Else
        S := 'no error';
      Writeln ('Unit #', I, '  Unitclear result: ', S);
    End;
End {of Excep_Test}.
```

2.2 The Command I/O Unit

The command I/O unit (called Command_IO) provides routines that control I/O redirection and maintain a queue of programs for sequential execution.

The command I/O unit provides the following capabilities:

Program Chaining - Maintain a queue of programs to be executed one after another.

I/O Redirection Control - Suspend and resume input and output I/O redirection.

The Command_IO routines are described in section 2.2.0. Section 2.2.1 contains a copy of the Command_IO unit interface section. A programming example using the Command_IO unit appears in section 2.2.2.

2.2.0 Using the Command I/O Unit

The command I/O unit is provided with all AOS releases. It is imported into a program by the "USES Command_IO;" statement (see the Programmer's Manual for details). The command I/O unit may reside in the intrinsics library, the system library, or a user library. The routines provided by this unit are described below.

The command I/O unit maintains no global variables and uses no other units.

2.2.0.0 Program Chaining

The command I/O unit provides two routines used for program chaining. Program chaining occurs when the system automatically executes a sequence of programs in succession. The Chain and Chain_Expedite procedures are used to specify the program sequence, which is maintained in the chain queue. Both procedures accept a string value parameter containing an execution option list. It may contain either the name of a code file containing a program, a list of I/O redirection options, or both. The execution option list is formatted and interpreted in the same way as an input to the system X(ecute command (see the System User's Manual). Examples of execution option lists are:

```
Backup
Format I="4,ssyy"
System.Compiler.  I="Test,$," O=Bucket:
Prose I=File.Names.Text,",,82/05/22" O=List.Text TO=#1:
```

The Chain procedure places an execution option list at the end of the chain queue. The Chain_Expedite procedure places an execution option list at the beginning of the chain queue. A call to either the Chain or the Chain_Expedite procedure with an empty execution option list clears the chain queue.

The execution option list at the beginning of the chain queue is processed after the termination of the currently executing program. It is processed without redisplaying the system prompt; program chaining appears transparent to the user. An attempt to process an excecution option list that references a nonexecutable program causes the chain queue to be cleared.

NOTE - The chain queue is allocated at system bootstrap time. The chain queue size may be configured using the Setup utility described in the System User's Manual.

NOTE - When the chain queue is full, calls to the Chain and Chain_Expedite procedures are ignored.

2.2.0.1 I/O Redirection

The command I/O unit provides five procedures to control I/O redirection. The Exception procedure accepts a boolean value parameter. It suspends all redirection and T-file processing for both the INPUT and OUTPUT files. If the boolean parameter is true, the chain queue is cleared. The Suspend_T and Resume_T procedures stop and restart T-file processing for both the INPUT and OUTPUT files. The Suspend_Redir and Resume_Redir procedures stop and resume redirection for the INPUT and OUTPUT files.

NOTE - Suspension of I/O redirection and T-file processing applies only to the redirection options processed in the invocation of the current program. Redirection options in effect before the invocation of the current program are not affected. For example, consider an executing program whose output is redirected to the printer. It may execute a program (using the Prog_Call procedure provided by the program operators unit -- see section 2.0) and specify output redirection to a remote port, superseding output to the printer. If the second program suspends its redirection using the Suspend_Redir procedure, output reverts to the printer.

2.2.1 The Command I/O Unit Interface

This section displays the text of the interface section belonging to the Command I/O unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details).

```
Unit Command_IO;
Interface

  Procedure Chain (Prog_Name : String);
    {Append Prog_Name to the end of the chain queue}

  Procedure Chain_Expedite (Prog_Name : String);
    {Insert Prog_Name at the beginning of the chain queue}

  Procedure Exception (Kill_Chain : Boolean);
    {Suspend all I/O redirection and T-file processing}

  Procedure Suspend_T;
    {Suspend all T-file processing}

  Procedure Resume_T;
     {Resume T-file processing}

  Procedure Suspend_Redir;
     {Suspend all redirection processing}

  Procedure Resume_Redir;
     {Resume redirection processing}
```

2.2.2 Programming Example

The following program demonstrates the capabilities of the command I/O unit.

```
Program Chain_Test;
Uses Command_IO;
Var Name : String;

  Procedure Get_Name (Var Name : String);
  Begin
    Write ('Chain what file (<return> to exit) ? ');
    Readln (Name);
  End {of Get_Name};

Begin
  Get_Name (Name);
  While Length (Name) <> 0 Do
    Begin
      Chain (Name);
      Get_Name (Name);
    End {of While};
End {of Chain_Test}.
```

2.3 The System Information Unit

The system information unit (called Sys_Info) allows user programs to obtain (and in some cases modify) system information previously accessible only to system programs.

The system information unit provides the following capabilities:

Work File Names - Obtain the volume names and file titles of the work text and work code files.

System Unit - Obtain the volume name and unit number of the system volume.

Prefixed Volume - Access and modify the prefixed volume name.

System Date - Access and modify the system date.

The Sys_Info routines are described in section 2.3.0. Section 2.3.1 contains a copy of the Sys_Info unit interface section. A programming example using the Sys_Info unit appears in section 2.3.2.

NOTE - Sys_Info accesses global file system information. See the System User's Manual for a description of the file system. This document uses the term title to specify the file title and suffix combined; this differs from the terminology used in the System User's Manual.

2.3.0 Using the System Information Unit

The system information unit is available as an option with the AOS. It is imported into a program by the "USES Sys_Info;" statement (see the Programmer's Manual for details). It may be installed in the intrinsics library, the system library, or the user library. All imported identifiers begin with "SI_" to prevent conflicts with program identifiers. The routines provided by this unit are described below.

The system information unit maintains no global variables and uses no other units.

2.3.0.0 Work Text File Name

The SI_Text_Vid and SI_Text_Tid procedures return the volume name and file title of the work text file.

SI_Text_Vid accepts a string variable as a parameter; the volume name of the work text file is returned in the string parameter. If a work text file does not exist, SI_Text_Vid returns an empty string.

SI_Text_Tid accepts a string variable as a parameter; the file title of the work text file is returned in the string parameter. If a work text file does not exist, SI_Text_Tid returns an empty string.

2.3.0.1 Work Code File Name

The SI_Code_Vid and SI_Code_Tid procedures return the volume name and file title of the work code file.

SI_Code_Vid accepts a string variable as a parameter; the volume name of the work code file is returned in the string parameter. If a work code file does not exist, SI_Code_Vid returns an empty string.

SI_Code_Tid accepts a string variable as a parameter; the file title of the work code file is returned in the string parameter. If a work code file does not exist, SI_Code_Tid returns an empty string.

2.3.0.2 System Unit Number

The SI_Sys_Unit function returns an integer function result. The unit number of the device containing the system volume is returned as the function value.

2.3.0.3 System Volume Name

The SI_Get_Sys_Vol procedure accepts a string variable as a parameter; the volume name of the system volume is returned in the string parameter.

2.3.0.4 Prefixed Volume Name

The SI_Get_Pref_Vol and SI_Set_Pref_Vol procedures access and modify the system's prefixed volume name. The prefixed volume name is normally set with either the filer's P(refix command or the P= redirection option. The default value for the prefixed volume name is the system volume name.

SI_Get_Pref_Vol accepts a string variable as a parameter; the prefixed volume name is returned in the string parameter.

SI_Set_Pref_Vol accepts a string expression as a parameter; the system's prefixed volume name is set to the string parameter value.

2.3.0.5 System Date

The SI_Get_Date and SI_Set_Date procedures access and modify the system date. The date record is declared in the interface section as follows:

```
Type SI_Date_Rec = Packed Record
                     Month : 0..12;
                     Day   : 0..31;
                     Year  : 0..99;
                   End; { SI_Date_Rec }
```

SI_Get_Date accepts a date record variable as a parameter; the system date is returned in the record parameter.

SI_Set_Date accepts a date record variable as a parameter; the system date is set to the value passed in the record parameter.

2.3.1 The System Information Unit Interface

This section displays the text of the interface section belonging to the system information unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "SI_" to prevent conflicts with host program identifiers.

```
Unit Sys_Info;
Interface

  Type SI_Date_Rec = Packed Record
                       Month : 0..12;
                       Day : 0..31;
                       Year : 0..99;
                     End; { SI_Date_Rec }

  Procedure SI_Code_Vid (Var SI_Vol : String);
    { Returns name of volume holding current workfile code }

  Procedure SI_Code_Tid (Var SI_Title : String);
    { Returns title of current workfile code }

  Procedure SI_Text_Vid (Var SI_Vol : String);
    { Returns name of volume holding current workfile text }

  Procedure SI_Text_Tid (Var SI_Title : String);
    { Returns title of current workfile text }

  Function SI_Sys_Unit : Integer;
    { Returns number of boot unit }

  Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
    { Returns system volume name }

  Procedure SI_Get_Pref_Vol (Var SI_Vol : String);
    { Returns prefix volume name }

  Procedure SI_Set_Pref_Vol (SI_Vol : String);
    { Sets prefix volume name }

  Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);
    { Returns current system date }

  Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
    { Sets current system date }
```

2.3.2 Programming Example

The following program demonstrates the capabilities of the system information unit.

```
Program Sys_Test;
Uses Sys_Info;
Var
  Ch    : Char;
  Date  : SI_Date_Rec;
  Vol,
  Title : String;

Begin
  SI_Code_Vid (Vol);
  SI_Code_Tid (Title);
  If Length (Title) <> 0 Then
    Writeln ('The Work Codefile is ', Vol, ':', Title)
  Else
    Writeln ('There is no Work Codefile.');
  SI_Text_Vid (Vol);
  SI_Text_Tid (Title);
  If Length (Title) <> 0 Then
    Writeln ('The Work Textfile is ', Vol, ':', Title)
  Else
    Writeln ('There is no Work Textfile.');

  Writeln;
  SI_Get_Sys_Vol (Vol);
  Writeln ('The System was booted on volume ', Vol,
           ': on unit ', SI_Sys_Unit);
  SI_Get_Pref_Vol (Vol);

  Writeln;
  Writeln ('The Prefix volume is ', Vol, ':');
  Write ('New Prefix: ');
  Readln (Vol);
  Delete (Vol, Pos (':', Vol), 1);
  If Length (Vol) In [1..7] Then
    Begin
      SI_Set_Pref_Vol (Vol);
      SI_Get_Pref_Vol (Vol);
      Writeln ('The Prefix volume is ', Vol, ':');
    End {of If}
  Else
    Writeln ('No change made');
```

```
  Writeln;
  SI_Get_Date (Date);
  Writeln ('The current date is ',
           Date.Month, -Date.Day, -Date.Year);
  Repeat
    Write ('Set for tomorrow''s date ? ');
    Read (Ch);
  Until Ch In ['y', 'Y', 'n', 'N'];
  Writeln;
  If Ch In ['y', 'Y'] Then
    With Date Do
      Begin
        Day := Day + 1;
        If (Month In [1,3,5,7,8,10,12]) And (Day = 32) Or
           (Month In [4,6,9,11]) And (Day = 31) Or
           (Month = 2) And (Day = 29) Then
          Begin
            Day := 1;
            If Month = 12 Then
              Begin
                Year := Year + 1;
                Month := 1;
              End {of If =12}
            Else
              Month := Month + 1;
          End {of If Month};
      SI_Set_Date (Date);
      SI_Get_Date (Date);
      Writeln ('The new date is ',
               Date.Month, -Date.Day, -Date.Year);
    End {of If Ch}
  Else
    Writeln ('No change made');
End {of Sys_Test}.
```

2.4 The System Utility Unit

The system utility unit (called Sys_Util) allows user programs access to miscellaneous system variables and functions. This unit may appear in the intrinsic library, the system library, or a user library.

The system utility unit provides the following capabilities:

System Serial Number - Obtain the serial number of the current system.

Time Delay - Suspend the current task for a given time period.

Maximum I/O Unit - Indicate the highest I/O unit number available.

The Sys_Util routines are described in section 2.4.0. Section 2.4.1 contains a copy of the Sys_Util unit interface section. A programming example using the Sys_Util unit appears in section 2.4.2.

2.4.0 Using the System Utility Unit

The system utility unit is provided with each AOS release. It is imported into a program by the "USES Sys_Util;" statement (see the Programmer's Manual for details). It may be installed in the intrinsics library, the system library, or a user library. All imported identifiers begin with "SU_" to prevent conflicts with program identifiers. The routines provided by this unit are described below.

The system utility unit maintains no global variables and uses no other units.

2.4.0.0 System Serial Number

Each PDQ-3 computer is assigned a unique system serial number. The SU_Ser_Num function returns an integer function result whose value is the serial number of the current system. An application program may check this serial number to guarantee that the program cannot be transported to any PDQ-3 other than the one for which it is sold. See the Hardware User's Manual for further details.

2.4.0.1 Time Delays

The SU_Delay procedure accepts an integer parameter specifying the number of 60ths of a second to delay further execution of the current task. The delay count may range between 1 and 32767; other values cause a delay of 1/60th of a second. Any number of tasks may be delayed at any time.

NOTE - If the system clock driver is not installed, use of the SU_Delay procedure causes no delay and sets the value of the IORESULT intrinsic (described in the Programmer's Manual) to 2; otherwise the IORESULT is set to 0.

2.4.0.2 Maximum I/O Unit Number

The SU_Max_Unit function returns the physical unit number of the highest-numbered system I/O device in the current system I/O configuration.

2.4.1 The System Utility Unit Interface

This section displays the text of the interface section belonging to the system utility unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "SU_" to prevent conflicts with host program identifiers.

```
Unit Sys_Util;
Interface

  Function SU_Ser_Num : Integer;
    { Returns system serial number }

  Procedure SU_Delay (Delay_Count : Integer);
    { Delays current task for Delay_Count 60ths of a sec }

  Procedure SU_Max_Unit : Integer;
    { Returns the highest I/O unit number }
```

2.4.2 Programming Example

The following program demonstrates the capabilities of the system information unit.

```
Program Util_Test;
Uses Sys_Util;
Var  I : Integer;
Begin
  Writeln ('The last I/O unit is ', SU_Max_Unit);
  For I := 0 To 9 Do
    Begin
      Writeln ('Delaying ', I, ' seconds; ',
               'System serial number is ', SU_Ser_Num);
      SU_Delay (I * 60);
    End;
End {of Util_Test}.
```

III. SCREEN FORMATTING

This chapter describes units which perform functions related to CRT screen formatting. The SC_Cntrl unit (section 3.0) provides routines that perform screen control operations on a number of terminals simultaneously.

3.0 The Screen Control Unit

The screen control unit (named SC_Cntrl) provides user programs access to terminal-handling information normally accessible only to system programs. The screen control unit's interface consists of a complete set of terminal-independent screen control operations (along with other useful terminal operations).

The screen control unit provides the following capabilities:

Cursor Positioning - Cursor movement operations including cursor home, X-Y cursor positioning, and cursor up/down/left/right.

Screen Clearing - Screen erase operations including clear line, erase to end of line, clear screen, and erase to end of screen.

Keyboard Input - Keyboard input may be mapped into terminal-independent screen commands.

Text Ports - A single terminal screen may be divided into a group of logically independent screens known as text ports. Text ports are functionally equivalent to entire terminal screens with respect to screen control operations.

Prompt Line Support - SC_Cntrl provides a routine for displaying user-defined prompt lines on text ports. Long prompt lines may be automatically divided into a number of sections (similar to UCSD Pascal system prompts).

Multitasking Support - SC_Cntrl routines are designed to support multitasking and multiterminal environments. Text ports and keyboards can be treated as critical resources to protect them from task contention.

Section 3.0.0 describes the fundamental concepts used in the screen control unit. Section 3.0.1 describes how to use the screen control facilities. Section 3.0.2 presents the unit's interface declarations. Section 3.0.3 gives programming examples demonstrating the use of the screen control unit.

3.0.0 Terminals

A terminal is a serial I/O device. It is accessed through two serial files: the input file and the output file. The output file is known as the terminal screen; the input file is known as the keyboard. The input file is assumed to be non-echoing. Here are some examples of terminals:

Terminal	Input file	Output file
System Console	SYSTERM:	CONSOLE:
Standard I/O	STANIN:	STANOUT:
Remote Console	REMIN:	REMOUT:

A terminal is characterized by its MISCINFO file and its terminal type. The MISCINFO file is created by the SETUP utility (see the System User's Manual for details); it describes the fixed functions of the terminal hardware. The file contains a description of the command sequences necessary to perform various terminal functions (e.g. home, clear screen). It also contains a description of the key sequences emitted by certain keys on the terminal keyboard (e.g. cursor up, cursor down). Note that some terminals may not provide command sequences for all the terminal functions defined in the MISCINFO file.

The terminal type indicates the command sequence necessary to position the screen cursor at given screen coordinates (this is called "random cursor addressing"). A screen has a height (in lines) and a width (in columns). The column number is called the X coordinate, and the line number is called the Y coordinate. A screen position is specified by the coordinate [x, y] (i.e. [<column>, <line>]). Note that all coordinates and dimensions are zero-based; thus, a screen containing 24 lines and 80 columns is said to have a height of 23 and a width of 79. Its upper left corner is [0, 0] and its lower right corner is [79, 23].

3.0.0.0 Text Ports

All screen control operations are performed on text ports. A text port is a rectangular subset of a screen, and is defined by three attributes:

-- The terminal on which it appears.
-- The screen coordinates of its upper left corner.
-- The dimensions of the text port.

Text ports simulate terminal screens. The upper left corner of a text port is addressed by [0, 0]; the lower right corner is addressed by [<width>, <height>]. Each text port maintains its own text port cursor. It is possible to declare any number of text ports on a single screen. Text ports may occupy all or any part of a terminal screen, and may overlap other text ports on the screen.

3.0.0.0.0 Text Port Records

Text port information is stored in a text port record. Text port records specify a text port's attributes and current state during screen control operations. The type declaration for a text port record is defined in the interface section of the screen control unit; it contains information describing a text port and its host terminal.

```
SC_Port     = Record
                        {Text Port Attributes}
                Height, Width,
                Row, Col,
                Cur_X, Cur_Y,

                        {Terminal Attributes}
                Max_Height, Max_Width : Integer;
                Slow, XY_Crt          : Boolean;
                Term_Type             : SC_Term_Type;
                Term_ID               : SC_Term_ID;

                In_File,
                Out_File,

                        {Overhead Information}
                In_Use_Count,
                Out_Use_Count         : Integer;
              End {of SC_Port};
```

The text port attributes define a particular text port. The Height and Width are the zero-based dimensions of the text port (e.g. a text port containing 24 lines has a Height of 23). The Row and Col specify the terminal screen coordinates of the upper left corner of the text port. Cur_X and Cur_Y are the text port-relative coordinates of the text port's cursor.

The terminal attributes describe the terminal on which the text port is declared. Max_Height and Max_Width contain the zero-based dimensions of the terminal screen. Slow is set to TRUE if the transmission speed to the terminal is slow enough that some prompts should be abbreviated. XY_Crt is set to TRUE if the terminal supports random cursor addressing. The screen control unit uses Term_ID to identify the proper command and key sequences for the terminal. Term_Type indicates the terminal type. In_File and Out_File specify the I/O unit numbers of the text port's keyboard and screen devices, respectively.

The overhead information describes the state of the text port. It is used internally by the screen control unit and provides no direct function to the program.

3.0.1 Using the Screen Control Unit

The screen control unit is available as an option with the AOS. It is imported into a program with the "USES SC_Cntrl;" statement (see the Programmer's Manual for details). The screen control unit may be installed in the intrinsics library, the system library, or the user library. All imported identifiers begin with "SC_" to prevent conflicts with program identifiers.

The screen control unit maintains 2 words of global variables and uses no other units. 88 words are allocated on the system heap each time the SC_Init function (section 3.0.1.0) is called successfully.

All screen control operations require a text port record as a parameter. (The destination terminal is implicit in this specification.) A text port record must be initialized for each terminal before the screen control unit can operate on the terminal.

NOTE - Two copies of the screen control unit exist: one compatible with hosts using the II.0 heap intrinsics and one compatible with hosts using the IV.0 heap intrinsics. The II.0-compatible copy is provided in the ACD.LIBRARY file; the IV.0-compatible copy is provided in the ACD.H+.LIBRARY file. Both the II.0-compatible copy and the IV.0-compatible copy are called SC_Cntrl. The heap usage of a given copy of the screen control unit may be determined using the Libmap utility documented in the System User's Manual.

3.0.1.0 Terminal Initialization

The SC_Init function initializes a terminal for use by the screen control unit. The screen file name, the keyboard file name, the MISCINFO file name, and the terminal type are passed as parameters. The SC_Init function creates a text port containing the entire terminal screen and places the text port cursor in the upper left corner ([0, 0]). If any of the file parameters cannot be opened, the function returns FALSE; otherwise the text port record is returned and the function value returns TRUE. Note that SC_Init should only be called once for a given terminal.

NOTE - The terminal type parameter indicates the random cursor positioning sequence appropriate for the terminal screen; it may have the value SC_System, SC_Soroc, SC_VT52, or SC_Datamedia. SC_System causes the screen control unit to use the system GOTOXY intrinsic to position the cursor within a text port. SC_System should only be used in conjunction with text ports declared on the system console. SC_Soroc may be used with any terminal compatible with Soroc or Lear Seigler ADM series terminals. SC_VT52 may be used with any terminal compatible with the DEC VT-52 terminal, including the Zenith Z-19 and the DEC VT-100. SC_Datamedia may be used with terminals compatible with the Datamedia 1500 and DT-80 series terminals.

WARNING - As a result of a successful call to the SC_Init function, screen control information is stored on the system heap. The

RELEASE intrinsic should not be used to deallocate a heap containing this information.

3.0.1.1 Terminal Operations

Terminal operations are screen control functions that do not alter the location of the text port cursor. A program may query the screen control unit as to the availability of certain screen control functions and keyboard functions. It may also read input from a terminal's keyboard or lock a terminal so no other task can access it.

3.0.1.1.0 Availability of Terminal Functions

The screen control unit provides the SC_Scrn_Has function to allow the program to determine whether the terminal is capable of performing a certain screen operation. (Some terminals do not support all of the screen control operations required by the screen control unit. The screen control unit simulates unsupported operations by using the supported operations where possible.) The function accepts a text port record and the desired operation (identified by a scalar of type SC_Scrn_Command) as parameters. The function returns TRUE if the terminal supports the desired command; otherwise, it returns FALSE.

The SC_Has_Key function is provided to allow a program to determine whether the keyboard is capable of generating a certain key sequence. The function accepts a text port record and the desired key sequence (identified by a scalar of type SC_Key_Command) as parameters. The function returns TRUE if the keyboard can generate the desired sequence; otherwise, it returns FALSE.

3.0.1.1.1 Multitasking Support

The SC_Out_Lock and SC_Out_Release, SC_In_Lock, and SC_In_Release procedures are used in multitasking environments to protect text ports from contention between tasks. These procedures accept a text port record as a parameter. The SC_Out_Lock procedure guarantees that only one task can write to the terminal screen associated with the specified text port. If another task has locked the screen, the calling task must wait until the other task releases the terminal. The SC_Out_Release procedure releases a terminal screen so that tasks waiting for access to the screen can proceed. The SC_In_Lock and SC_In_Release procedures provide analagous functions for the terminal keyboard.

Note that when more than one task accesses text ports on a single terminal, all tasks must use these procedures to guarantee mutually exclusive terminal access.

All text port operations contain internal locks to ensure their own mutually exclusive access to text ports. Calls to the SC_Out_Lock and SC_In_Lock procedures are necessary only when executing se-

quences of text port operations that cannot be interrupted by other tasks.

3.0.1.1.2 Keyboard Input

The SC_Map_CRT_Command function reads a key sequence from the keyboard associated with a specified text port and returns a scalar (of type SC_Key_Command) describing the key sequence. A character containing the actual key sequence is also returned; if the key generated an escape sequence, the character returned is the character value of the second character of the key sequence biased by 128. The key is not echoed to the screen.

The SC_Map_CRT_Command procedure protects the keyboard from access by other tasks until the input operation is complete. This guarantees that keys generating multicharacter sequences are processed before other tasks are allowed to read from the keyboard.

3.0.1.2 Text Port Operations

Text port operations fall into three categories: cursor positioning, screen clearing, and prompt line support. All operations are limited to the screen area specified by the text port. Coordinates passed to the procedures performing text port operations are relative to the upper left corner of the text port.

3.0.1.2.0 Cursor Positioning

The cursor positioning procedures are:

SC_Left	move cursor left
SC_Right	move cursor right
SC_Up	move cursor up
SC_Down	move cursor down
SC_Home	move cursor to [0, 0]
SC_Goto_XY	move cursor to [X, Y]

All procedures require a text port record as a parameter. The procedures will not move the cursor outside of the text port; attempts to do so cause the cursor to "stick" at the edge of the text port.

The SC_Goto_XY procedure moves the cursor to a specific position in the text port. It requires the coordinates of the new cursor position as a parameter.

3.0.1.2.1 Screen Clearing

The screen clearing operations are:

SC_Clr_Line	Clears line from [0, Y]
SC_Erase_To_Eol	Clears line from [X, Y]
SC_Clr_Screen	Clears screen from [0, 0]
SC_Eras_Eos	Clears screen from [X, Y]

All procedures require a text port record as a parameter. Some procedures require a coordinate as a parameter; with others, the coordinate is implied. The cursor is placed at the specified text port coordinates. The screen control unit attempts to perform the operation with the screen functions available on the terminal. If the terminal does not support the screen operation, the operation is simulated by writing blank characters and then re-positioning the cursor at the specified text port coordinates.

3.0.1.2.2 Prompt Lines

The SC_Prompt function provides prompt line support when using text ports. The function accepts a prompt line of the format:

<header>: <command> <command> <command> ... <version>

Here is an example of a prompt line:

Command: A(ssem, F(ile, E(dit, R(un, H(alt [1.0]

The prompt is placed at a specified position in the text port. If the prompt is long enough that it would extend past the end of the text port, the SC_Prompt function breaks the prompt up (at some specified break character, usually ',') and displays the <header> and <version> parts along with as many commands as can fit on the line. After the prompt is written, the cursor may be placed either at the end of the prompt line or at another location in the text port.

The SC_Prompt function returns a key sequence read from the keyboard in response to the prompt. Two sets are passed to SC_Prompt as parameters. If the primary acceptance set (of type SC_Key_Command) is empty, the prompt line is displayed and SC_Prompt returns an undefined character value; otherwise, SC_Prompt continues to read keys from the keyboard until a key sequence is read whose corresponding scalar is found in the primary acceptance set. If the primary acceptance set contains the SC_Unknown scalar and the key sequence read from the keyboard does not correspond to any of the keys enumerated in the SC_Key_Command type, then the key must be found in the secondary acceptance set (of type CHAR). Note that escape sequences are specified in the secondary acceptance set by biasing the second character of the sequence by 128.

If the prompt line is broken up because it is too long for the text port, a '?' is appended to the list of commands. A '?' input causes a new subset of commands to be displayed in place of the

prompt line. This continues until all commands have been displayed, at which time the initial prompt line is redisplayed.

3.0.1.3 Multiple Text Ports on a Single Terminal

Further text ports may be declared on a terminal screen by calling the SC_New_Port procedure. This procedure accepts the Row, Col, Height, and Width of the new text port as parameters. It also requires a text port record belonging to a text port previously created on the terminal screen. It returns a text port record describing the new text port. Note that if any part of the new text port is outside of the terminal screen, the new text port is truncated so that it lies entirely on the terminal screen.

Most terminal screens have a single cursor. When multiple text ports exist on a terminal screen, multiple cursors may be simulated by ca[illegible] SC_Goto_XY procedure before accessing the text port; this is necessary when writing to the text port or calling SC_Right, SC_Left, SC_Down and SC_Up. While the cursor position for a text port is maintained in the text port record, the terminal screen's cursor may reside in another text port on the screen. Thus, the terminal screen cursor is guaranteed to coincide with the text port cursor after a call to SC_Goto_XY. (Note that this is also true after calling SC_Home, SC_Clr_Line, SC_Erase_To_Eol, SC_Clr_Screen, SC_Eras_Eos and SC_Prompt).

Note that when more than one task accesses text ports on a single terminal screen, the SC_Out_Lock or SC_In_Lock procedures may be necessary in order to ensure mutually exclusive access to the terminal.

WARNING - Passing text port records by value is NOT recommended since this results in the creation of text port records that have not been initialized by the SC_New_Port procedure.

3.0.2 The Screen Control Unit Interface

The interface section of the screen control unit is presented here for reference. Note that all identifiers begin with "SC_".

```
Unit SC_Cntrl;
Interface

Type
    SC_Term_Id       = ^Integer;

    SC_CHSet         = Set Of Char;

    SC_Long_String   = String[255];

    SC_Term_Type     = (SC_System, SC_Soroc, SC_VT52,
                        SC_Datamedia);

    SC_Scrn_Command  = (SC_WHome, SC_Eras_S, SC_Erase_Eol,
                        SC_Clear_Lne, SC_Clear_Scn,
                        SC_Up_Cursor, SC_Down_Cursor,
                        SC_Left_Cursor, SC_Right_Cursor);

    SC_Key_Command   = (SC_Backspace_Key, SC_Eof_Key,
                        SC_Etx_Key, SC_Escape_Key, SC_Del_Key,
                        SC_Up_Key, SC_Down_Key, SC_Left_Key,
                        SC_Right_Key, SC_Unknown);

    SC_Key_Set       = Set Of SC_Key_Command;

    SC_Port          = Record
                              {Text Port Attributes}
                         Height, Width,
                         Row, Col,
                         Cur_X, Cur_Y,

                              {Terminal Attributes}
                         Max_Height, Max_Width : Integer;
                         Slow, XY_Crt          : Boolean;
                         Term_Type             : SC_Term_Type;
                         Term_ID               : SC_Term_ID;

                         In_File,
                         Out_File,

                         In_Use_Count,
                         Out_Use_Count         : Integer;
                       End;
```

```
Function SC_Init (Info_Name,
                  In_Name,
                  Out_Name  : String;
                  Term_Type : SC_Term_Type;
                  Var Port  : SC_Port)      : Boolean;
    {This function initializes a text port on a terminal.
     Info_Name contains the name of the MISCINFO file contain-
     ing the terminal characteristics.  In_Name names the
     keyboard file for the terminal.  Out_Name names the
     screen file for the terminal.  Term_Type indicates the
     type of random cursor addressing necessary for the
     terminal.  SC_System specifies the current system cursor
     addressing routine, GOTOXY.  Port is returned containing
     the text port record.  If any files cannot be opened,
     Port is undefined and the function result is FALSE.}


Procedure SC_New_Port (Var New_Port : SC_Port;
                       Col,
                       Row,
                       Width,
                       Height       : Integer;
                       Sample_Port  : SC_Port);
    {This procedure declares a new text port whose upper left
     corner is at the coordinates [Col, Row] on the terminal
     screen.  It has the the dimensions Width by Height.  The
     New_Port is attached to the same terminal as is associat-
     ed with the Sample_Port.}


Procedure SC_Out_Lock (Var Port : SC_Port);
    {This procedure is called when more than one task might
     access text ports on a given terminal screen.  It secures
     the screen for the caller until an SC_Out_release oc-
     curs.}


Procedure SC_Out_Release (Var Port : SC_Port);
    {This procedure is called to release a terminal screen
     secured by a SC_Out_Lock call.}


Procedure SC_In_Lock (Var Port : SC_Port);
    {This procedure is called when more than one task might
     access text ports on a given terminal keyboard.  It
     secures the keyboard for the caller until an SC_In_Re-
     lease occurs.}


Procedure SC_In_Release (Var Port : SC_Port);
    {This procedure is called to release a terminal secured by
     a SC_In_Lock call.}
```

```
Function SC_Scrn_Has (Var Port : SC_Port;
                      What     : SC_Scrn_Command) : Boolean;
    {This function returns TRUE if the terminal associated
     with the text port described by Port has the What
     function.}


Procedure SC_Left (Var Port : SC_Port);
    {The cursor is placed one position to the left in the text
     port described by Port.}


Procedure SC_Right (Var Port : SC_Port);
    {The cursor is placed one position to the right in the
     text port described by Port.}


Procedure SC_Up (Var Port : SC_Port);
    {The cursor is placed one position up in the text port
     described by Port.}


Procedure SC_Down (Var Port : SC_Port);
    {The cursor is placed one position down in the text port
     described by Port.}


Procedure SC_Home (Var Port : SC_Port);
    {The cursor is placed at [0, 0] in the text port described
     by Port.}


Procedure SC_Goto_XY (Var Port : SC_Port;
                      X,
                      Line     : Integer);
    {The cursor is placed at [X, Line] in the text port
     described by Port.}


Procedure SC_Clr_Line (Var Port : SC_Port;
                       Y        : Integer);
    {The text port described by Port is erased from [0, Y] to
     the end of the text port line and the cursor is
     repositioned at [0, Y].}


Procedure SC_Erase_To_Eol (Var Port : SC_Port;
                           X,
                           Line     : Integer);
    {The cursor is placed at [X, Line] in the text port
     described by Port. The line is erased from this position
     to the end of the text port line and the cursor is
     repositioned at [X, Line].}
```

```
Procedure SC_Clr_Screen (Var Port : SC_Port);
    {The cursor is placed at [0, 0] in the text port described
     by Port.  The text port is erased from this position to
     the end of the text port and the cursor is repositioned
     at [0, 0].}


Procedure SC_Eras_Eos (Var Port : SC_Port;
                       X,
                       Line     : Integer);
    {The cursor is placed at [X, Line] in the text port
     described by Port. The text port is erased from this
     position to the end of the text port and the cursor is
     repositioned at [X, Line].}


Function SC_Has_Key (Var Port : SC_Port;
                     What     : SC_Key_Command) : Boolean;
    {Returns TRUE if the terminal keyboard associated with the
     text port described by Port has the What function.}


Function SC_Map_Crt_Command (Var Port : SC_Port;
                             Var K_Ch : Char)
                            : SC_Key_Command;
    {Reads K_Ch from the keyboard associated with the text
     port described in Port and returns the character and a
     scalar describing the character. If the key sequence is
     prefixed, it is returned biased by 128. Note that the
     second character of an escape sequence is returned in
     K_Ch.}


Function SC_Prompt (Var Port     : SC_Port;
                    Line         : SC_Long_String;
                    X_Cursor,
                    Y_Cursor,
                    X_Pos,
                    Where        : Integer;
                    Match_Chars  : SC_CHSet;
                    Match_Keys   : SC_Key_Set;
                    Break_Char   : Char)      : Char;
    {This function displays the prompt line in the text port
     described by Port at [X_Pos, Where]. If the prompt line
     is too long for the text port, it is broken up into
     several chunks. It can only be broken where the Break_
     Char occurs. After the prompt is displayed, the text
     port cursor is placed at [X_Cursor, Y_Cursor]. Note that
     if X_Cursor < 0, the cursor is placed after the last
     character in the prompt Line. If the set Match_Keys <>
     [], characters are read from the keyboard associated with
     the text port until a key sequence is read that is in
     Match_Keys. If the match is SC_Unknown, the character
     must occur in the set Match_Chars.}
```

3.0.3 Programming Example

The following program demonstrates most of the functions of the screen control unit. It starts one task for each of two terminals. Each task starts by exercising keyboard sequences. Each task then demonstrates the text port operations on each of several text ports.

A text port is created on a region of the screen. The entire screen is filled with "." characters, providing a backdrop to make text port operations more easily visible. The text port is first filled with characters. The various text port operations are then carried out within the text port.

Note that the mutual exclusion procedures, SC_Out_Lock and SC_In_Lock, are not used since no more than one task ever writes to a given terminal.

```
Program Tester;
Uses SC_Cntrl;
Var P : Processid;


  Process SC_Test (In_Name,
                   Out_Name : String);
      {This process declares text ports on the terminal
       described by In_Name and Out_Name.  It then proceeds
       to test the key sequences and then the text ports.}
  Var Original,
      Port     : SC_Port;
      Junk     : Boolean;
      Out_Fyle : Text;
      In_Fyle  : Interactive;


    Procedure Test_Keyboard (Port : SC_Port);
        {This procedure tests each of the pre-defined
         key sequences.}
    Var Current_Key : SC_Key_Command;


      Procedure Verify (Name : String);
      Var Ch : Char;
      Begin
        Write (Out_Fyle, 'Type the ', Name, 'key -- ');
        If Current_Key = SC_Map_Crt_Command (Port,Ch) Then
          Writeln (Out_Fyle, 'Correct')
        Else
          Writeln (Out_Fyle,
                   'Incorrect -- key typed : ', Ord (Ch));
        Current_Key := Succ (Current_Key);
      End {of Verify};
```

```
Begin {of Test_Keyboard}
  Current_Key := SC_Backspace_Key;
  Verify ('Backspace ');
  Verify ('Eof ');
  Verify ('Etx ');
  Verify ('Escape ');
  Verify ('Delete ');
  Verify ('Up ');
  Verify ('Down ');
  Verify ('Left ');
  Verify ('Right ');
End {of Test_Keyboard};


Procedure Test_Port (Columns,
                     Rows,
                     Width,
                     Height       : Integer);
Const Eol = 13;
Var I,
    J    : Integer;
    Ch   : Char;
    Port : SC_Port;
    Line : Packed Array [0..79] Of Char;
    Dots : Packed Array [0..1919] Of Char;


  Procedure Delay;
  Begin
    Readln (In_Fyle);
  End {of Delay};


Begin {of Test_Port}
  SC_New_Port (Port, Columns, Rows,
               Width, Height, Original);
  SC_Clr_Screen (Original);
  With Port Do
    Writeln (Out_Fyle,
             'Left Corner at ', ' [', Col, ',', Row, '];',
             ' Width = ', Width + 1,
             ' Height = ', Height + 1);
  Delay;

                              {Clear screen to '.'s}
  {This assumes wraparound on 80'th column}
  FillChar (Dots, Sizeof (Dots), '.');
  Write (Out_Fyle, Dots : (Port.Max_Height + 1) *
                          (Port.Max_Width + 1) - 1);
```

```
                                {Fill Port with alphas}
SC_Clr_Screen (Port);
For I := 0 To 79 Do
  Line[I] := Chr (33 + I);
For I := 0 To Port.Height - 1 Do
  Begin
    SC_Goto_XY (Port, 0, I);
    Write (Out_Fyle, Line : Port.Width + 1);
  End;
SC_Goto_XY (Port, 0, Port.Height);
Write (Out_Fyle, Line : Port.Width);
Delay;

                              {Ring around the text port}
{We send one too many arrows in each direction to make
 sure we bump into the edge of the text port.}
SC_Home (Port);
For I := 0 To Port.Height Do
  SC_Down (Port);
For I := 0 To Port.Width Do
  Sc_Right (Port);
For I := 0 To Port.Height Do
  SC_Up (Port);
For I := 0 To Port.Width Do
  Sc_Left (Port);
Delay;

                            {Criss cross port with spaces}
For J := 0 To Port.Height Do
  Begin
    SC_Goto_XY (Port, J, J);
    If (Port.Width + Port.Col <> Port.Max_Width) Or
       (J <> Port.Height) Then
      Write (Out_Fyle, ' ');
    SC_Goto_XY (Port, Port.Width - J, J);
    Write (Out_Fyle, ' ');
  End;
Delay;

                                {Wipe out right half}
For J := 0 To Port.Height Do
  SC_Erase_To_Eol (Port, Port.Width - J, J);
Delay;

                                {Wipe out lower half}
SC_Eras_Eos (Port, Port.Width Div 2, Port.Height Div 2);
Delay;

                                {Wipe out whole thing}
For J := Port.Height Div 2 Downto 0 Do
  SC_Clr_Line (Port, J);
Delay;
```

```
                                    {Check out prompts}
      Ch := SC_Prompt (Port,Concat('Command: A(ssem, C(omp, ',
                                   'F(ile, G(omp, H(alt, ',
                                   'X(ecute [A]'),
                       -1, -1, 0, 0,
                       ['A', 'C', 'F', 'G', 'H', 'X'],
                       [SC_Unknown], ',');
      SC_Goto_XY (Port, Port.Width Div 2, Port.Height Div 2);
      Write (Out_Fyle, Ch);
      Delay;
    End {of Test_Port};


  Begin {of SC_Test}
    Rewrite (Out_Fyle, Out_Name);
    Reset (In_Fyle, In_Name);
    Junk := SC_Init ('*SYSTEM.MISCINFO', In_Name, Out_Name,
                     SC_VT52, Original);
    Test_Keyboard (Original);
    Test_Port ( 0,  0, 79, 23);
    Test_Port (10, 10, 10, 10);
    Test_Port ( 0,  0, 11, 11);
    Test_Port (68,  0, 11, 11);
    Test_Port ( 0, 12, 11, 11);
    Test_Port (68, 12, 11, 11);
    Test_Port ( 9,  0, 37, 11);
    Test_Port ( 9, 12, 37, 11);
    Test_Port ( 0,  0, 79, 11);
    Test_Port ( 0, 12, 79, 11);
  End {of SC_Test};


Begin
  Start (SC_Test ('SYSTERM:', 'CONSOLE:'), P, 4000);
  Start (SC_Test ('REMIN1:', 'REMOUT1:'), P, 4000);
End.
```

IV. DATA MANIPULATION

This chapter describes units which perform functions involving the manipulation of integer, real, and string data.

The integer conversion unit (section 4.0) provides type transfer functions for variables of types string and integer. The real conversion unit (section 4.1) provides type transfer functions for variables of types string and real. The pattern matching unit (section 4.2) provides routines that compare strings containing wildcards to strings containing text.

4.0 The Integer Conversion Unit

The integer conversion unit (named Num_Con) contains routines capable of translating integers between numerical and character string representations. Commonly used numerical functions are also provided.

The integer conversion unit provides the following capabilities:

-- String-to-integer and integer-to-string conversion.

-- Integer min and max functions.

-- Unsigned integer comparison operators.

Section 4.0.0 presents concepts needed for using the integer conversion unit. Section 4.0.1 provides a detailed description of each function. Section 4.0.2 displays the text comprising the integer conversion unit's interface section. Section 4.0.3 contains a programming example.

4.0.0 Integers

An integer is defined as a whole number in the range -32768..32767. Integers are stored as two's complement binary values in variables of type integer. Values stored in this fashion are referred to as numerical integers. Integers can also be stored as character strings in variables of type string. Values stored in this fashion are referred to as character string integers. Character string integers have the following form:

```
<integer> ::= <sign> <digit> {<digit>}

<sign>    ::= [ + | - ]

<digit>   ::= 0 | 1 | 2 | 3 | 4 |
              5 | 6 | 7 | 8 | 9
```

Examples of character string integers:

```
-32768
+1
1
0
16
```

4.0.0.0 Unsigned Integers

Unsigned integers are defined to contain values in the range 0..65535. They are represented as 16-bit binary values and are stored in variables of type integer. They differ from signed integers only in use and interpretation. They are identical in the

range 0..32767; however, unsigned values in the range 32768..65535 are treated by integer operations as signed values in the range -32767..-1. Since the div, mod, >, >=, <, and <= operators do not work correctly with large unsigned integers, some useful unsigned integer functions are provided in the number conversion unit.

4.0.1 Using the Integer Conversion Unit

The integer conversion unit is available as an option with the AOS. It is imported into a program by the "USES Num_Con;" statement (see the Programmer's Manual for details). The Num_Con unit may be installed in either the intrinsics library, the system library, or a user library. All imported identifiers begin with "N_" to prevent conflicts with program identifiers.

The integer conversion unit maintains no global variables and uses no other units.

4.0.1.0 Integer Conversion

The N_Int_To_Str procedure accepts a signed integer expression and a variable of type string as parameters. The character string integer corresponding to the value of the integer expression is returned in the string variable.

The N_Uns_To_Str procedure accepts an unsigned integer expression and a variable of type string as parameters. The character string integer corresponding to the value of the unsigned integer expression is returned in the string variable.

The N_Str_To_Int function accepts three parameters and returns an integer result. The first parameter is a string variable containing a character string integer. The second parameter is an integer expression specifying the starting position of the character string integer in the string parameter. If N_Str_To_Int finds a valid character string integer within the string parameter, it returns the corresponding numerical integer in the third parameter (which is an integer variable). The function returns an index into the string parameter which points to the character immediately following the character string integer. If the string does not contain a valid character string integer, the function returns the value of the second parameter.

The syntax for character string integers parsed by N_Str_To_Int is a superset of the format described in section 4.0.0 - leading blank characters are ignored. Character string integers are defined to terminate either at the first nonconforming character or at the end of the string expression.

4.0.1.1 Numerical Functions

This section describes the miscellaneous numerical functions provided by the integer conversion unit.

4.0.1.1.0 Signed Integer Min and Max

The N_Min and N_Max functions accept two signed integer parameters and return an integer result. N_Min returns the lesser of the two parameters. N_Max returns the greater of the two parameters.

4.0.1.1.1 Unsigned Integer Min and Max

The N_Min_U and N_Max_U functions accept two unsigned integer parameters and return an integer result. N_Min_U returns the lesser of the two parameters. N_Max_U returns the greater of the two parameters.

4.0.1.1.2 Unsigned Integer Comparison

The N_Leq_U and N_Geq_U functions accept two unsigned integer parameters and return a Boolean result. N_Leq_U returns TRUE if the first parameter is less than or equal to the second parameter; otherwise, it returns FALSE. N_Geq_U returns TRUE if the first parameter is greater than or equal to the second parameter; otherwise, it returns FALSE.

4.0.2 The Integer Conversion Unit Interface

This section displays the text of the interface section belonging to the integer conversion unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "N_".

```
Unit Num_Con;
Interface

Function N_Str_To_Int (Str      : String;
                       Index    : Integer;
                   Var Num      : Integer) : Integer;
   {Converts the string Str to an integer Num beginning at the
    character  Str[Index].  If there is a valid integer in the
    string, the value of the function  is  the  index  in  the
    string  after  the  last  character in the integer format;
    otherwise, the function value is Index.}

Procedure N_Int_To_Str (I      : Integer;
                    Var S      : String);
   {Converts the integer I to a string S.}

Function N_Min (X, Y : Integer) : Integer;
   {Returns the smaller of the two integers X and Y.}

Function N_Max (X, Y : Integer) : Integer;
   {Returns the larger of the two integers X and Y.}

Function N_Leq_U (X, Y : Integer) : Boolean;
   {Returns TRUE if unsigned integer X <= unsigned integer Y.}

Function N_Geq_U (X, Y : Integer) : Boolean;
   {Returns TRUE if unsigned integer X >= unsigned integer Y.}

Function N_Min_U (X, Y : Integer) : Integer;
   {Returns the smaller of the two unsigned integers X and Y.}

Function N_Max_U (X, Y : Integer) : Integer;
   {Returns the larger of the two unsigned integers X and Y.}

Procedure N_Uns_To_Str (I : Integer;
                    Var S : String);
   {Converts the unsigned integer I to a string S.}
```

4.0.3 Programming Example

The following program demonstrates the capabilities of the integer conversion unit.

```
Program Num_Demo;
Uses Num_Con;

      { This program accepts a string containing integers,
        parses them, finds the minimum and maximum values,
        and prints the numbers separated by ','s.}

Var Convert_Num,
    In_String,
    Out_String     : String;
    Index,
    Largest,
    Next_Index,
    Number,
    Smallest       : Integer;

Begin
  Writeln;
  Write ('Type your integers separated by <blank>s : ');
  Readln (In_String);
  Largest := -Maxint;
  Smallest := Maxint;
  Out_String := '';
  Index := 1;
  Next_Index := N_Str_To_Int (In_String, Index, Number);
  While Next_Index <> Index Do
    Begin
      N_Int_To_Str (Number, Convert_Num);
      Out_String := Concat (Out_String, Convert_Num, ', ');
      Largest := N_Max (Largest, Number);
      Smallest := N_Min (Smallest, Number);
      Index := Next_Index;
      Next_Index := N_Str_To_Int (In_String, Index, Number);
    End {of If};
  If Next_Index <> 1 Then
    Begin
      Delete (Out_String, Length (Out_String) - 1, 2);
      Writeln ('Reconstructed numbers: ', Out_String);
      Writeln ('Largest signed number was ', Largest,
               '; Smallest signed number was ', Smallest);
    End {of If}
  Else
    Writeln ('No valid integers were found.');
End {of Num_Demo}.
```

4.1 The Real Conversion Unit

The real conversion unit (named Real_Con) contains routines capable of translating reals between numerical and character string representations. Commonly used numerical functions are also provided.

The real conversion unit provides the following capabilities:

-- String-to-real and real-to-string conversion.

-- Real min and max functions.

Section 4.1.0 presents concepts needed for using the real conversion unit. Section 4.1.1 provides a detailed description of each function. Section 4.1.2 displays the text comprising the real conversion unit's interface section. Section 4.1.3 contains a programming example.

4.1.0 Reals

A real number is defined as a rational number which lies in the range -3.0E38 .. 3.0E38 and is accurate to seven digits. Real numbers are stored as signed binary values in variables of type real. Values stored in this fashion are referred to as numerical reals. Real numbers may also be stored as character strings in variables of type string. Values stored in this fashion are referred to as character string reals. There are two kinds of character string representations for real numbers: fixed point format, and floating point format.

A real number displayed in floating point format consists of a signed fraction (with absolute value less than 9.99...) and a signed integer specifying an integral power of ten. Character string reals displayed in floating point format have the following form:

```
<floating point format> ::= <mantissa part> <exponent part>

<mantissa part>         ::= <sign> <mantissa>

<mantissa>              ::= <whole part> [<decimal part>]

<whole part>            ::= <digit string>

<decimal part>          ::= .<digit string>

<exponent part>         ::= e<exponent> | E<exponent>

<exponent>              ::= <sign> <digit> [<digit>]

<digit string>          ::= {<digit>}

<sign>                  ::= [ - | + ]

<digit>                 ::= 0 | 1 | 2 | 3 | 4 |
                            5 | 6 | 7 | 8 | 9
```

NOTE - One restriction on this grammar is that <whole part> and <decimal part> cannot both be empty.

A real number displayed in fixed point format consists of a signed fraction. Character string reals displayed in fixed point format have the following form:

```
<fixed point format> ::= <mantissa part>
```

NOTE - The syntax specification presented in this document (and thus recognized by the real conversion unit) is a superset of the standard Pascal syntax for real numbers.

Examples of character string reals displayed in both fixed and floating point format:

Fixed point format	Floating point format
123	1.23e2
-123.456	-1.23456E+02
+1234567.	+1.234567E5
.5432	-5.432e-1

4.1.1 Using the Real Conversion Unit

The real conversion unit is available as an option with the AOS. It is imported into a program by the "USES Real_Con;" statement (see the Programmer's Manual for details). The Real_Con unit may be installed in either the intrinsics library, the system library, or a user library. All imported identifiers begin with "R_" to prevent conflicts with program identifiers.

The real conversion unit maintains no global variables, but does use the number conversion unit (Num_Con, section 4.0) in its implementation section.

4.1.1.0 Real Conversion

The R_Real_To_Str and R_Str_To_Real routines perform type conversion between real and string values.

4.1.1.0.0 Real to String Conversion

The R_Real_To_Str procedure accepts four parameters. The first parameter is a real variable containing the numerical real to be converted. The second parameter is an integer variable whose value determines the format of the resulting character string real; it is known as the format specifier. The third parameter is an integer variable whose value determines the number of significant digits in the character string real; it is known as the precision specifier. The fourth parameter is a string variable used to return the character string real.

If the value in the format specifier is less than one, the character string real is returned in floating point format; otherwise, fixed point format is returned, and the value of the format specifier determines the number of digits to the right of the decimal point.

The precision specifier determines the number of significant digits contained in the character string real. If the format specifier specifies more digits after the decimal point than the total number of significant digits (as specified by the precision specifier), the end of the character string real is padded with blank characters to make up the difference.

The conversion of numerical reals into character string reals is subject to a few restrictions (violation of which may cause system crashes or execution errors):

-- When the value of a numerical real is less than 1 and greater than -1, the resulting fixed format real can temporarily occupy a string of size (|exponent| + precision specifier + 3). The string variable passed to the R_Real_To_Str must be at least this size; otherwise, unpredictable results may occur.

-- Attempts to convert character string reals into numerical reals whose values lie outside the range of the floating point

implementation may cause execution errors. This problem occurs when the exponent value of a real number is positive and <exponent value> + <format specifier> >= 38.

4.1.1.0.1 String to Real Conversion

The R_Str_To_Real function accepts three parameters and returns an integer result. The first parameter is a string variable containing a character string real. The second parameter is an integer expression specifying the starting position of the character string real in the string parameter. If R_Str_To_Real finds a valid character string real within the string parameter, it returns the corresponding numerical real in the third parameter (which is a real variable). The function returns an index into the string parameter which points to the character immediately following the character string real. If the string does not contain a valid character string real, the function returns the value of the second parameter.

The syntax for character string reals parsed by R_Str_To_Real is a superset of the format described in section 4.1.0; leading blank characters are ignored. Character string reals are terminated either by the first nonconforming character or by the end of the string expression.

Note that the conversion of a character string real into a numerical real may cause an execution error if the value of the numerical real exceeds the range of real variables.

4.1.1.1 Numerical Functions

The R_Min and R_Max functions accept two real parameters and return a real result. R_Min returns the lesser of the two parameters. R_Max returns the greater of the two parameters.

4.1.2 The Real Conversion Unit Interface

This section displays the text of the interface section belonging to the real conversion unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "R_".

```
Unit Real_Con;
Interface


Function R_Str_To_Real (Str      : String;
                        Index    : Integer;
                    Var Num      : Real) : Integer;
   {Converts the string Str to a real Num starting at the
    S[Index]. If there is a valid real in the string, the
    value of the function is the index in the string after the
    last character in the real format; otherwise, the function
    value is Index.}


Procedure R_Real_To_Str (R           : Real;
                         D,
                         P           : Integer;
                     Var Return      : String);
   {Converts the real number R to a string Return with P
    significant digits. If D < 1, the floating point format
    is used. If D > 0, the fixed point format is used and D
    is the number of decimal digits.}


Function R_Min (X, Y : Real) : Real;
   {Returns the smaller of the two reals X and Y.}


Function R_Max (X, Y : Real) : Real;
   {Returns the larger of the two reals X and Y.}
```

4.1.3 Programming Example

The following program demonstrates the capabilities of the real conversion unit.

```
Program Real_Demo;
Uses Real_Con;

    { This program accepts a string containing reals,
      parses them, finds the minimum and maximum values,
      and prints the numbers separated by ','s.}

Var Convert_Num,
    In_String,
    Out_String     : String[255];
    Index,
    Next_Index     : Integer;
    Largest,
    Number,
    Smallest       : Real;

Begin
  Writeln;
  Write ('Type your reals separated by <blank>s : ');
  Readln (In_String);
  Largest := -1.0E38;
  Smallest := 1.0E38;
  Out_String := '';
  Index := 1;
  Next_Index := R_Str_To_Real (In_String, Index, Number);
  While Next_Index <> Index Do
    Begin
      R_Real_To_Str (Number, 0, 6, Convert_Num);
      Out_String := Concat (Out_String, Convert_Num, ', ');
      Largest := R_Max (Largest, Number);
      Smallest := R_Min (Smallest, Number);
      Index := Next_Index;
      Next_Index := R_Str_To_Real (In_String, Index, Number);
    End {of If};
  If Next_Index <> 1 Then
    Begin
      Delete (Out_String, Length (Out_String) - 1, 2);
      Writeln ('Reconstructed numbers: ', Out_String);
      Writeln ('Largest number was ', Largest,
               '; Smallest number was ', Smallest);
    End {of If}
  Else
    Writeln ('No valid reals were found.');
End {of Real_Demo}.
```

4.2 The Pattern Matching Unit

The pattern matching unit (named Pattern_Match) contains routines capable of comparing strings containing wildcards to strings containing text. The wildcards available are similar to those provided on the UNIX(tm) operating system from Bell Laboratories.

Section 4.2.0 presents the concepts needed for using the pattern matching unit. Section 4.2.1 provides a detailed description of the pattern matching function. Section 4.2.2 displays the text comprising the pattern matching unit's interface section. Section 4.2.3 contains a programming example.

4.2.0 Wildcards

Wildcards are character sequences which are treated specially when encountered in a character string; they are named wildcards because of their ability to match whole classes of character sequences rather than a single character sequence. For instance, the string "a=" matches all character strings starting with the letter "a" because "=" is defined as a wildcard which matches any character sequence.

This section describes the wildcard conventions recognized by the pattern matching unit. Characters treated specially are described in section 4.2.0.0. The wildcard character "?" matches any single character; it is described in section 4.2.0.1. The subrange wildcard is similar to the "?" wildcard, but restricts the set of matching characters to the range specified in the wildcard itself. Subrange wildcards are described in section 4.2.0.2. The wildcard character "=" matches any sequence of characters (including the empty sequence); it is described in section 4.2.0.3.

4.2.0.0 Special Characters

The Pattern_Match wildcard convention defines the following characters as special characters:

```
-- question mark "?"
-- equals sign "="
-- curly brackets "{" and "}"
-- comma ","
-- hyphen "-"
-- tilde "~"
-- backslash "\"
```

Special characters may only be used as parts of wildcards; however, a literal occurrence of a special character can be represented by a two character sequence consisting of a backslash followed by the special character. A backslash indicates that the following character is to appear literally in the character string; for instance, "xx\=yy" is treated as the literal character string "xx=yy" rather than a wildcard string.

NOTE - Literal occurrences of backslashes (i.e. "\") are specified by pairs of backslashes (i.e. "\\").

Examples of backslash in wildcards:

"ab\?def"	matches	"ab?def"
"ab{a-z, \=}de\\f"	matches	"ab=de\f"
"ab\-def"	matches	"ab-def"

4.2.0.1 Question Mark Wildcard

A question mark matches any single character.

Examples of "?" wildcard:

Pattern:	"ab?def"
Matches:	"abbdef" "abrdef"
Nonmatches:	"abdef" "abjkdef" "abef"

4.2.0.2 Equals Sign Wildcard

An equals sign matches any sequence of characters, including the empty sequence.

Examples of "=" wildcard:

Pattern:	"ab=def="
Matches:	"abcdefg" "abdef" "abcccdef"
Nonmatches:	"abcef"

4.2.0.3 Subrange Wildcard

The subrange wildcard matches a single character from the character set specified in the subrange. The following special characters are used to construct subrange wildcards: comma, hyphen, tilde, and curly brackets.

A subrange wildcard consists of a character set delimited by curly brackets. A character set consists of a list of character items separated by commas. A character item is either a character or a character range (two characters separated by a hyphen). A character range implicitly specifies all characters lying between the two

characters. (Consult the ASCII table in Appendix C to determine the ordering of characters.) Character items preceded by tildes are called negated items.

Examples of subrange wildcards:

```
{a,b,c}
{a-d,j,w-z}
{a-z,~j,~x-y}
```

Syntax for subrange wildcard:

```
<wild card>   ::=   "{"<item list>"}"
<item list>   ::=   <item>{,<item>}
<item>        ::=   [~] <char item>
<char item>   ::=   <char> | <range>
<range>       ::=   <char>-<char>
<char>        ::=   a  printable ASCII character
```

The initial value of the character set depends on the first character item in the list. If the first item is negated, the set initially contains all characters; otherwise, the set is initially empty.

The list of character items is evaluated left-to-right. Characters specified by nonnegated items are included into the set; characters specified by negated items are excluded from the set. Thus, a character matches the subrange wildcard if it matches one of the nonnegated items, but does not match any of the negated choices. For example, the subrange "{a-z,~r}" represents the set of characters from "a" to "z", excluding "r".

NOTE - Subrange wildcards must contain nonempty character sets; otherwise, a file name error occurs. Blank characters within subrange wildcards are ignored.

NOTE - Wildcard characters can be specified in character sets with the backslash notation described in section 4.2.0.0.

Examples of subrange wildcard:

```
Pattern:        "ab{a-r, ~j, ~k}def"

Matches:        "abbdef"
                "abrdef"

Nonmatches:     "abjdef"
                "abkdef"
                "abzdef"
```

4.2.1 Using the Pattern Matching Unit

The pattern matching unit is available as an option with the AOS. It is imported into a program by the "USES Pattern_Match;" statement (see the Programmer's Manual for details). The Pattern_Match unit may be installed in either the intrinsics library, the system library, or a user library. All imported identifiers begin with "P_" to prevent conflicts with program identifiers.

The pattern matching unit maintains no global variables and uses no other units.

NOTE - Some pattern matching information returned by the Pattern_Match unit is stored in a linked list on the system heap. The dynamic variable management intrinsics MARK, RELEASE, and DISPOSE can be used to deallocate these data structures. Note that careless use of these intrinsics may result in undefined pointer values.

NOTE - Two copies of the pattern matching unit exist: one compatible with hosts using the II.0 heap intrinsics and one compatible with hosts using the IV.0 heap intrinsics. The II.0-compatible copy is provided in the ACD.LIBRARY file; the IV.0-compatible copy is provided in the ACD.H+.LIBRARY file. Both the II.0-compatible copy and the IV.0-compatible copy are called Pattern_Match. The heap usage of a given copy of the pattern matching unit may be determined using the Libmap utility documented in the System User's Manual.

4.2.1.0 Wildcard String Matching

The P_Match function serves as a general purpose pattern matcher for string variables. The two main parameters are a wildcard string and a literal string. A wildcard string is a character string which may contain the wildcards defined in section 4.2.0. The characters in a literal string are treated literally. P_Match determines whether the literal string matches the wildcard string. If the strings match, P_Match returns TRUE; otherwise, it returns FALSE.

NOTE - P_Match cannot match two wildcard strings; the characters in the test string are treated literally.

P_Match optionally provides information describing how the strings were matched (i.e. which character strings in the test string matched the wildcards in the wildcard string). This information is returned as a linked list on the system heap.

4.2.2 The Pattern Matching Unit Interface

This section displays and discusses the text of the interface section belonging to the pattern matching unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "P_".

```
Unit Pattern_Match;
Interface

Type
  P_Pat_Rec_P = ^P_Pat_Rec;
  P_Pat_Rec = Record
                Comp_Pos,
                Comp_Len,
                Wild_Pos,
                Wild_Len : Integer;
                Next     : P_Pat_Rec_P;
              End {of P_Pat_Rec};

Function P_Match (Wild,
                  Comp     : String;
              Var P_Ptr    : P_Pat_Rec_P;
                  P_Info   : Boolean) : Boolean;
  { Compares the Wild string (possibly containing wildcards) to
    the Comp literal string and returns TRUE if they match. If
    P_Info is True, P_Ptr is returned pointing to a list of
    records containing information on how Wild and Comp were
    matched; otherwise, P_Ptr is returned Nil.}
```

4.2.2.0 Pattern Matching Information

If P_Info is passed TRUE, P_Match returns pattern matching information. P_Ptr is a pointer to a linked list of records of type P_Pat_Rec containing the starting positions and lengths of corresponding character patterns in Wild and Comp.

The Comp_Pos and Wild_Pos fields are the starting positions of corresponding character patterns in Comp and Wild, respectively. Comp_Len and Wild_Len are the pattern lengths. Next points to the next pattern record in the list; its value is NIL in the last pattern record. The patterns occur in the list in the order in which they were matched in the strings.

If the strings do not match, or the list was not requested (i.e. P_Info is passed FALSE), P_Ptr is returned NIL.

Example of pattern record list:

```
Wild contains:      '=ab{a-m}=f?'
Comp contains:      'abcdefg'
```

If P_Info is passed TRUE, the pattern record list returned is:

```
1.  Wild_Pos = 1, Wild_Len = 1
    Comp_Pos = 1, Comp_Len = 0
        ('=' matches the empty string)
2.  Wild_Pos = 2, Wild_Len = 2
    Comp_Pos = 1, Comp_Len = 2
        ('ab' matches 'ab')
3.  Wild_Pos = 4, Wild_Len = 5
    Comp_Pos = 3, Comp_Len = 1
        ('{a-m}' matches 'c')
4.  Wild_Pos = 9, Wild_Len = 1
    Comp_Pos = 4, Comp_Len = 2
        ('=' matches 'de')
5.  Wild_Pos = 10, Wild_Len = 1
    Comp_Pos = 6, Comp_Len = 1
        ('f' matches 'f')
6.  Wild_Pos = 11, Wild_Len = 1
    Comp_Pos = 7, Comp_Len = 1
        ('?' matches 'g')
```

NOTE - When the "=" wildcard in Wild matches an empty string in Comp, Comp_Len is set to 0 and Comp_Pos is set to the position of the next pattern in Comp (i.e. the position where a nonempty pattern would have occurred). Be sure to check the validity of Comp_Pos indices before using them to reference characters in Comp; otherwise, range errors may occur.

4.2.3 Programming Example

The following program is an example of a string comparison routine which uses P_Match. The program reads two strings and prints the result of the comparison; if requested, it also prints information describing how the patterns matched.

```
Program Wild_Test;
Uses Pattern_Match;

Var
  W, C          : String;
  Ch            : Char;
  Pat_Ptr       : P_Pat_Rec_P;
  Want_Patterns : Boolean;

  Procedure Print_Patterns (Pat_Ptr : P_Pat_Rec_P;
                            C, W    : String);
  Var Count : Integer;
  Begin
    Writeln ('Type <cr> for patterns');
    Readln; Writeln;
    Count := 1;
    Repeat
      Writeln ('Pattern ', Count, ' :');
      With Pat_Ptr^ Do
        Begin
          Writeln ('  Comp : ', C);
          If Comp_Len <> 0 Then Write ('^':(Comp_Pos + 9));
          If Comp_Len > 1 Then Write ('^':(Comp_Len - 1));
          Writeln;
          Writeln ('  Wild : ', W);
          Write ('^':(Wild_Pos + 9));
          If Wild_Len > 1 Then Write ('^':(Wild_Len - 1));
          Writeln; Writeln;
        End;
      Pat_Ptr := Pat_Ptr^.Next;
      Count := Count + 1;
    Until Pat_Ptr = Nil;
  End {of Print_Patterns};
```

```
Begin {of Wild_Test}
  Repeat
    Writeln ('--WildCard Check--');
    Write ('Wildcard String   : ');
    Readln (W);
    Write ('Comparison String : ');
    Readln (C);
    Write ('Want pattern matching information ? [y/n] ');
    Read (Ch);
    Want_Patterns := Ch In ['y','Y'];
    Writeln; Writeln;
    If P_Match (W, C, Pat_Ptr, Want_Patterns) Then
      Writeln ('A Match')
    Else Writeln ('No Match');
    If Want_Patterns And (Pat_Ptr <> Nil) Then
      Print_Patterns (Pat_Ptr, C, W);
    Write ('Continue ? [y/n] ');
    Read (Ch);
    Writeln; Writeln;
  Until Ch In ['n', 'N'];
End {of Wild_Test}.
```

V. FILE SYSTEM MANIPULATION

This chapter describes units which perform functions involving the manipulation of the UCSD Pascal file system.

The directory information unit (section 5.0) provides routines that operate on directories and external files. The file information unit (section 5.1) provides routines that return information on file variables declared within a program.

5.0 The Directory Information Unit

The directory information unit (named Dir_Info) contains routines which provide user programs with access to file system information previously accessible only to system programs.

The directory information unit provides the following capabilities:

Directory Information Access - For any on-line disk unit, Dir_Info returns the volume name, volume date, number of disk files on volume, amount of unused space, and attributes of individual disk files.

Directory Manipulation - Dir_Info provides routines for changing the date or name of a disk file or volume, removing files from a volume, and taking volumes off-line.

Wildcards - Most Dir_Info routines recognize the wildcard convention established by the pattern matching unit (section 4.2) in their file name arguments.

Error Handling - Dir_Info defines a standard error result (similar to UCSD Pascal I/O results) for routines involved with file names and directory searches.

Multitasking Support - Dir_Info provides routines for protecting file system information from contention between concurrent tasks. These routines ensure that only one task can modify file system information at a time.

Section 5.0.0 presents concepts needed for using the directory information unit. Section 5.0.1 provides an overview of the Dir_Info routines. Section 5.0.2 describes the routines in detail and provides programming examples. Section 5.0.3 displays the text comprising the directory information unit's interface section.

5.0.0 Basic Concepts

This section describes the concepts and features needed to use the directory information unit.

5.0.0.1 Wildcards

Most Dir_Info routines allow wildcards in their file name arguments. Wildcards are character sequences which are treated specially when encountered in a character string; they are named wildcards because of their ability to match whole classes of character sequences rather than a single character sequence. The wildcard convention observed by the Dir_Info routines is described in the documentation for the pattern matching unit (section 4.2).

NOTE - The D_Change_Name, D_Change_End, and D_Scan_Title routines do not recognize wildcards in their file name arguments; wildcard

characters are treated literally.

5.0.0.2 File Name Arguments

Most Dir_Info routines accept file name arguments. The file name specifies the volume and/or file to be accessed by the routine. See the System User's Manual for a complete description of UCSD Pascal files and file names.

File name syntax:

```
<file name>    ::= <volume id><file id>

<volume id>    ::= [ #<unit number>:    |
                     <volume name>:     |
                     : | * | *:         ]

<file id>      ::= [<title><suffix><specifier>]

<specifier>    ::= "["<number>"]" | "["*"]"
```

NOTE - Volume names and file titles may contain wildcards. Unit numbers and colons separating volume id's and file id's must appear literally; they must be independent of any wildcard.

NOTE - All Dir_Info routines except D_Scan_Title ignore file length specifiers.

NOTE - File name conventions in Dir_Info differ slightly from UCSD Pascal file name conventions in cases where the UCSD conventions are inconsistent:

-- Dir_Info considers an empty file name argument to specify the prefix volume (i.e. <file id> is empty (implying a volume reference), and <volume id> is empty (implying the prefixed volume)). An empty string is not a valid file name in UCSD Pascal.

-- Dir_Info interprets wildcard file names of the form "<vol name>:=" to be valid volume specifiers. This is consistent with Dir_Info's definition of the "=" wildcard, but inconsistent with the UCSD Filer's interpretation of the "=" wildcard; the Filer does not accept file names of this form as volume specifiers.

5.0.0.3 File Type Selection

Some Dir_Info routines accept a file type parameter (named D_Select) which is used to specify the file objects to be accessed. (File objects include volumes, unused areas on disk volumes, temporary files, text files, code files, and data files.) The file type parameter is necessary because file names are not sufficient to completely specify all types of file objects (e.g. unused disk

areas). Both the file name argument and the D_Select parameter are used by the routines which generate directory information to determine the file objects on which to return information. See section 5.0.2.1 for details.

Dir_Info defines a scalar type which is used to specify file objects. D_Select is declared as a set of this type; a file object is selected by including its corresponding scalar in D_Select.

File object types:

```
D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_Temp, D_Free);
D_Choice   = Set Of D_NameType;
```

The scalar values are defined as follows:

D_Vol — Select all volumes matching the file name argument. Note that while volume names may contain wildcards, unit numbers must be specified literally.

D_Free — Select all unused areas of disk space on the volumes matching the file name argument.

D_Temp — Select all temporary files matching the file name argument. Files are considered temporary if they have been opened (but not yet locked) by a program.

D_Text — Select all text files matching the file name argument.

D_Code — Select all code files matching the file name argument.

D_Data — Select all data files matching the file name argument.

5.0.0.4 File Dates

Disk files and disk volumes are assigned file dates. File dates are stored in records of type D_Date_Rec and are accessed and modified by the Dir_Info routines D_Dir_List and D_Change_Date.

D_Date_Rec is declared as follows:

```
D_DateRec = Packed Record
              Month : 0..12;
              Day   : 0..31;
              Year  : 0..100;
            End; { D_DateRec }
```

A year value of 100 in a file date record indicates that the object is a temporary disk file. (This is a UCSD Pascal file system convention.)

5.0.0.5 File End

Data may be entered into disk files by using either the standard Pascal procedure PUT or the UCSD Pascal intrinsic BLOCKWRITE. The last block of a file created using PUT may be only partially full of valid data (because a record written using PUT may not occupy an entire block). The last block of a file created using BLOCKWRITE is considered full of valid data. A file end attribute is associated with each disk file; it indicates the number of valid bytes (1..512) of data in the last block of the file. The file end may be determined using the Dir_List routine; it may be modified using the D_Change_End routine.

5.0.0.6 Error Results

All Dir_Info routines which access file system information return a value reflecting the result of the file system operation. This result indicates either that the routine finished without errors or that an error occurred; valid information is not returned when routines return a result value indicating the occurrence of an error.

Conditions causing errors include:

-- The specified files, volumes, or unused spaces can not be found in the disk directory.

-- The specified unit is off-line.

-- The file name argument has improper syntax.

-- The specified file name conflicts with an existing file.

In no cases can an error cause abnormal termination of a function; errors which cannot be identified explicitly by the routine are flagged by returning a result indicating that an unknown error has occurred.

Dir_Info defines a scalar type to describe the possible errors encountered:

```
Type  D_Result = (D_Okay,
                  D_Not_Found,
                  D_Exists,
                  D_Name_Error,
                  D_Off_Line,
                  D_Other);
```

Details on error results and the status of the returned directory information in the presence of an error can be found in section 5.0.2.

5.0.1 Using the Directory Information Unit

This section provides a functional overview of the directory information unit's capabilities. See section 5.0.2 for programming examples and detailed descriptions of Dir_Info routines.

The directory information unit is available as an option with the AOS. It is imported into a program with the "USES Pattern_Match, Dir_Info;" statement (see the Programmer's Manual for details). The directory information unit may be installed in the intrinsics library, the system library, or the user library. All identifiers imported from Dir_Info begin with "D_" to prevent conflicts with program identifiers. Identifiers imported from Pattern_Match begin with "P_".

The directory information unit maintains 3 words of global variables. It also uses the pattern matching unit (Pattern_Match described in section 4.2) in its interface section.

NOTE - The directory information returned by some Dir_Info routines is stored in linked lists on the system heap. The dynamic variable management intrinsics MARK, RELEASE, and DISPOSE can be used to deallocate these data structures. Note that careless use of MARK and RELEASE may result in undefined pointer values.

NOTE - Two copies of the directory information unit exist: one compatible with hosts using the II.0 heap intrinsics and one compatible with hosts using the IV.0 heap intrinsics. The II.0-compatible copy is provided in the ACD.LIBRARY file; the IV.0-compatible copy is provided in the ACD.H+.LIBRARY file. Both the II.0-compatible copy and the IV.0-compatible copy are called Dir_Info. The heap usage of a given copy of the directory information unit may be determined using the Libmap utility documented in the System User's Manual.

5.0.1.0 Unit Initialization

When the Dir_Info unit is used under the Advanced Operating System, unit initialization is performed automatically. When using this unit under other versions of UCSD Pascal, the D_Init procedure must be called before any of the Dir_Info routines are used. See section 5.0.2.6 for more information on D_Init.

5.0.1.1 File Name Parsing

The D_Scan_Title function parses file name arguments according to the syntax rules for UCSD file names, and returns the file name's volume id, title, type, and length specifier. The function result is used to flag invalid file name arguments.

See section 5.0.2.0 for more information on D_Scan_Title.

5.0.1.2 Directory Information

The D_Dir_List function creates a list of records containing directory information on volumes and disk files. This information includes volume names and unit numbers of blocked and unblocked on-line units, the number of files on blocked units, lengths and starting blocks of disk files and unused disk spaces, file names and types, file dates, and file ends. The function result is used to flag invalid file name arguments, off-line volumes, or not-found files.

D_Dir_List optionally provides information describing how the wildcard file name argument matched files and/or volumes.

See section 5.0.2.1 for more information on D_Dir_List.

5.0.1.3 Directory Manipulation

Dir_Info provides four routines for manipulating directory information: D_Change_Name, D_Change_Date, D_Change_End, and D_Rem_Files.

The D_Change_Name function accepts two main parameters: the name of an existing file and a new file name. The existing file is searched for in the specified disk directory; if found, its name is changed to the new file name. (Volume names are changed in similar fashion by passing only volume id's in the file name arguments.) D_Change_Name optionally prevents the deletion of existing files having the same file name as the new file name. The function result is used to flag invalid file names, not-found files, or the success of the name change. See section 5.0.2.2 for more information on D_Change_Name.

NOTE - D_Change_Name does not recognize wildcards in its file name arguments; wildcard characters are treated literally.

The D_Change_Date function changes the file date for all files and/or volumes whose names match the file name argument. The function result is used to flag invalid file name arguments, off-line volumes, or not-found files. See section 5.0.2.3 for more information on D_Change_Date.

The D_Change_End function accepts two parameters: the name of an existing file and a new file end. The existing file is searched for in the specified disk directory; if found, the current file end is changed to the specified file end. The function result is used to flag invalid file names, not-found files, or the success of the end change. D_Change_End is documented further in section 5.0.2.5.

NOTE - D_Change_End does not recognize wildcards in its file name arguments; wildcard characters are treated literally. It operates on disk files exclusively.

The D_Rem_Files function removes files or volumes whose file names match the specified file name argument. A removed disk file is permanently deleted from the directory. A removed volume is taken

off-line so that it no longer appears in the volume table (as displayed in the filer's V(olume command); disk volumes come back on-line when they are referenced, but serial volumes are inaccessible until the system is reinitialized. The function result is used to flag invalid file name arguments, not-found files, or off-line volumes. See section 5.0.2.4 for more information on D_Rem_Files.

5.0.1.4 Multitasking Support

Dir_Info provides three routines for protecting directory information from task contention: D_Init, D_Lock, and D_Release. These routines ensure that directory information is properly treated as a shared resource in multitasking environments.

D_Init initializes the synchronization mechanism used to protect the directory.

D_Lock grants exclusive directory access to the task that executes it; however, a task may be suspended until another task releases the directory lock before it can continue execution past its call to D_Lock.

D_Release releases directory access to whatever task that may desire it. Tasks already waiting for directory access are automatically awakened when the directory becomes available by a call to D_Release.

NOTE - It is the programmer's responsibility to ensure protection of the directory in multitasking environments. Each task must call D_Lock before accessing the directory, and then call D_Release when it is finished accessing the directory.

NOTE - All Dir_Info routines and file system intrinsics (e.g. RESET) contain internal locks to ensure their own mutually exclusive access to directory information. Calls to D_Lock and D_Release are necessary only when executing a series of Dir_Info routines which must not be interrupted by other tasks. Note that tasks must not call standard file system intrinsics after locking the directory; only Dir_Info routines can be used to access directories locked by D_Lock.

NOTE - Tasks which call Dir_Info routines must contain sufficient stack space to execute the routines without causing stack overflows. Calls to D_Scan_Title consume approximately 800 words of stack space. Calls to D_Dir_List, D_Change_Name, D_Change_Date, and D_Rem_Files consume approximately 2000 words of stack space. Calls to D_Init, D_Lock, and D_Release use negligible amounts of stack space.

5.0.2 Directory Information Intrinsics

This section provides detailed descriptions of the Dir_Info routines. See section 5.0.1 for an overview of the routines. Each sub-section contains a programming example demonstrating the use of the routine.

5.0.2.0 D_Scan_Title

Syntax:

```
Function D_Scan_Title (D_Name   : String;
                       Var D_Volume,
                           D_Title  : String;
                       Var D_Type   : D_NameType;
                       Var D_Segs   : Integer) : D_Result;
```

D_Scan_Title parses the UCSD Pascal file name passed in D_Name, and returns the file name's volume id, file title, file type, and file length specifier. The function result indicates the validity of the file name argument. See section 5.0.0.2 for more information on file name arguments.

5.0.2.0.0 D_Scan_Title Parameters and Function Result

D_Scan_Title accepts the following parameters:

D_Name — A string containing a UCSD Pascal file name. Wildcards are ignored.

D_Volume — A string which returns the volume id contained in D_Name. If D_Name contains no volume id or if the volume id is ':', D_Volume is assigned the system's prefix volume name. If the volume id is '*' or '*:', D_Volume is as signed the system's boot volume name. Volume names assigned to D_Volume may contain only upper case characters, and no blank characters.

D_Title — A string which returns the file title contained in D_Name. If D_Name does not contain a file title, D_Title is assigned the empty string. File titles assigned to D_Title contain only upper case characters, and do not contain blank characters.

D_Type — A scalar which returns a value indicating the file type of the file name contained in D_Name.

Definition of D_Type's scalar type:

```
D_NameType = (D_Vol, D_Code, D_Text,
              D_Data, D_Temp, D_Free);
```

D_Type is set to D_Vol if the file title in D_Name is empty. D_Type is set to D_Code if the file title is terminated by ".CODE" or to D_Text if the file title is terminated by ".TEXT" or ".BACK". If none of the above holds true, D_Type is set to D_Data. See section 5.0.0.3 for more information on file types.

D_Segs — An integer which is assigned a value indicating the presence of a file length specifier in D_Name. The value returned in D_Segs is assigned as follows:

Length Specifier	D_Segs Value
[<number>]	<number>
[*]	-1
<not present>	0

D_Scan_Title returns a function result of type D_Result (see section 5.0.0.6 for more information). The only scalar values returned by D_Scan_Title are D_Okay and D_Name Error; they have the following meanings:

D_Okay — No Error. All information returned by D_Scan_Title is valid.

D_Name_Error — Illegal file name syntax in D_Name. The information returned by D_Scan_Title is invalid.

5.0.2.0.1 Programming Example

The following program demonstrates the use of D_Scan_Title.

```
Program Scan_Test;
Uses Pattern_Match, Dir_Info;
Var Name,
    Volume,
    Title    : String;
    Typ      : D_Name_Type;
    Seg_Flag : Integer;
    Result   : D_Result;
    Ch       : Char;
Begin
  Writeln('--D_ScanTitle Test');
  Repeat
    Writeln;
    Write('File name to parse: ');
    Readln(Name);
    Result := D_ScanTitle(Name, Volume, Title,
                          Typ, Seg_Flag);
    Writeln('parsed: ');
    Writeln('  Volume name  -- ', Volume);
    Writeln('  File name    -- ', Title);
    Write  ('  File type    -- ');
    Case Typ Of
      D_Text : Writeln('text file');
      D_Code : Writeln('code file');
      D_Data : Writeln('data file');
    End; { Cases }
    If Seg_Flag <> 0 Then
      Writeln('  Segment flag -- ', Seg_Flag);
    Writeln;
    Write('Continue? ');
    Read(Ch);
    Writeln;
  Until Ch In ['n', 'N'];
End.  { Scan_Test }
```

5.0.2.1 D_Dir_List

Syntax:

```
Function D_Dir_List (D_Name   : String;
                     D_Select : D_Choice;
                 Var D_Ptr    : D_List_P;
                     D_PInfo  : Boolean) : D_Result;
```

D_Dir_List creates a list of records containing directory information on volumes and disk files. This information includes volume names and unit numbers of blocked and unblocked on-line units, number of files on blocked units, lengths and starting blocks of disk files and unused disk spaces, file names and types, file dates, and file ends. The function result value indicates invalid file name arguments, off-line volumes, or not-found files.

D_Dir_List optionally provides information describing how the wildcard file name argument matched files and/or volumes.

5.0.2.1.0 D_Dir_List Parameters and Function Result

D_Dir_List accepts a string containing a file name and a set specifying the file types on which to return information. D_Dir_List returns a pointer to a linked list of directory information records. Each record contains the name of a file or volume which matches the file name argument and also is one of the types specified in the file type set.

5.0.2.1.0.0 D_Name

The D_Name parameter contains a file name which may contain wildcards.

5.0.2.1.0.1 D_Select

The D_Select parameter is a set specifying the directory objects for which information is to be returned by D_Dir_List. See section 5.0.0.3 for more information on directory object selection.

5.0.2.1.0.2 D_Ptr

The D_Ptr parameter is assigned a pointer to a linked list of records containing directory information for all specified file objects. In order to have information returned describing it, a file object must meet the following criteria:

-- It must reside on a volume which matches the volume id in D_Name.

-- If the object is a disk file, it must match the file id in D_Name.

-- It must belong to one of the types included in D_Select.

The linked list contains one record for each file object matched. The records are defined as follows (P_Pat_Rec_P is imported from Pattern_Match):

```
D_List_P = ^D_List;
D_List   =
    Record
      D_Unit          : Integer;
      D_Volume        : String[7];
      D_VPat          : P_Pat_Rec_P;
      D_Next_Entry    : D_List_P;
      Case D_Is_Blkd : Boolean Of
        True : (D_Start,
                D_Length     : Integer;
                Case D_Kind : D_Name_Type Of
                  D_Vol,
                  D_Temp,
                  D_Code,
                  D_Text,
                  D_Data : (D_Title : String[15];
                            D_FPat  : P_Pat_Rec_P;
                            D_Date  : D_Date_Rec;
                            Case D_Name_Type Of
                              D_Vol : (D_Num_Files : Integer);
                              D_Temp,
                              D_Code,
                              D_Text,
                              D_Data: (D_End : Integer)));
    End; { D_List }
```

The fields in the D_List record return the following information for each file object in the D_Ptr list:

-- D_Unit returns the unit number of the unit containing the object.

-- D_Volume returns the name of the volume containing the object.

-- D_VPat is a pointer to pattern matching information collected while comparing volumes to the volume id in D_Name (see section 5.0.0.1 for details on pattern matching info). D_VPat is set to NIL if pattern matching information is not requested (see section 5.0.2.1.0.3 for details).

-- D_Next_Entry is a pointer to the next directory information record in the list. It is set to NIL if the current record is the last record in the list.

-- D_Is_Blkd is set to TRUE if the file object is (or resides on) a block-structured unit. Records describing serial volumes have D_IsBlked set to FALSE; the remaining fields are undefined.

The following fields exist only in records describing file objects stored on disk units (i.e. D_Is_Blkd is TRUE):

-- D_Start contains the starting block number of the file object. If the object is of type D_Vol, this value is interpreted as the block number of the first block on the volume (e.g. 0 for disk volume).

-- D_Length contains the length (in blocks) of the file object. If the object is of type D_Vol, this value is interpreted as the total number of blocks on the volume (e.g. 494 for single density 8" floppy disk).

-- D_Kind indicates the type of the file object described by the current record. (See section 5.0.0.3 for more information.)

The following fields exist only in records describing disk file objects other than unused disk areas (i.e. D_Kind in [D_Vol, D_Temp, D_Code, D_Text, D_Data]):

-- D_Title contains the file title of the object. For objects of type D_Vol, this field contains the empty string.

-- D_FPat is a pointer to pattern matching information collected while comparing file names to the file id in D_Name (see section 5.0.0.1 for details on pattern matching info). D_FPat is set NIL if pattern matching information is not requested or if the file id in D_Name is empty.

-- D_Date contains the file date (section 5.0.0.4) for the current object.

-- D_Num_Files is valid only for objects of type D_Vol; it contains the number of files in the volume's directory.

-- D_End is valid only for objects of type D_Temp, D_Text, D_Code, and D_Data. It is the number of valid data bytes in the last record of the file.

File information is returned (in a linked list accessed by D_Ptr) starting with information on the lowest numbered I/O unit whose volume name matches D_Name. If D_Vol is in D_Select, a volume entry is emitted. File entries and unused entries specified in D_Select are emitted in block-number order. This pattern is repeated for each I/O unit in ascending unit-number order.

5.0.2.1.0.3 D_PInfo

When set to TRUE, the D_PInfo parameter indicates that pattern matching information should be returned in a linked list accessed by D_Ptr. This information is collected by the P_Match function (of the Pattern_Match unit) in the process of comparing volume and file id's, and is useful for determining how the wildcards in D_Name were expanded. Information is returned in two pointers; one for volume names matched (named D_VPat) and one for file id's matched (named D_FPat).

Example of pattern record lists:

D_Name is set to '=:TEST{1-9}='

Two volumes contain files which match D_Name:

```
BOOT contains TEST5.CODE
WORK contains TEST5.TEXT
```

For BOOT:TEST5.CODE, D_Volume is 'BOOT', D_Title is 'TEST5.CODE', and D_VPat returns a pointer to the following information:

```
1. Wild_Pos is 1, Wild_Len is 1
   Comp_Pos is 1, Comp_Len is 4
       ('=' matches 'BOOT')
```

D_FPat returns a pointer to the following information:

```
1. Wild_Pos is 1, Wild_Len is 4
   Comp_Pos is 1, Comp_Len is 4
       ('TEST' matches 'TEST')
2. Wild_Pos is 5, Wild_Len is 5
   Comp_Pos is 5, Comp_Len is 1
       ('{1-9}' matches '5')
3. Wild_Pos is 10, Wild_Len is 1
   Comp_Pos is 6, Comp_Pos is 5
       ('=' matches '.CODE')
```

A similar list is returned for WORK:TEST5.TEXT.

NOTE - If the volume id in D_Name consists of a unit number (e.g. "#5"), the volume assigned to the unit is defined to match the volume id in D_Name. The Pos and Len pointers are set as in the following example:

D_Name is set to "#5:"

A disk volume named "FOON" resides in unit 5.

1. Wild_Pos is 1, Wild_Len is 2
   Comp_Pos is 1, Comp_Pos is 4
   ('#5' matches 'FOON')

NOTE - D_FPat and D_VPat never contain invalid information. If information is unavailable or has not been requested, the pointers are set to NIL.

5.0.2.1.0.4 Function Result

D_Dir_List returns a value of type D_Result. D_Dir_List can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay	No error. All D_Ptr information is valid.
D_Not_Found	No such file/volume found. No match found for D_Name. D_Dir_List sets D_Ptr to NIL.
D_Name_Error	Illegal syntax in D_Name. D_Dir_List sets D_Ptr to NIL.
D_Off_Line	Volume/unit off-line. The volume specified by D_Name was not on-line. This error occurs only when the volume id in D_Name does not contain wildcards (i.e. a single volume is specified, and it is off-line). If the volume name in D_Name contains wildcards but does not match any on-line volumes, D_Dir_List returns D_Not_Found. D_Ptr is set to NIL.
D_Other	Unknown error. D_Dir_List encountered an error it could not identify, but which interrupted normal execution of the function. D_Ptr is set to NIL.

5.0.2.1.1 Programming Example

The following program is a general purpose directory lister; it accepts a string containing wildcards and creates a list of matching files and (if requested) pattern matching information for the files. Note that the program uses the MARK and RELEASE intrinsics to remove D_Dir_List information from the heap after the information has been used.

```
Program D_Test;
Uses Pattern_Match, Dirinfo;
Var Select        : D_Choice;
    Want_Patterns : Boolean;
    Heap_Ptr      : ^Integer;
    Segs          : Integer;
    Typ           : D_Name_Type;
    Volume, Title,
    Match         : String;
    Result        : D_Result;
    Ch            : Char;
    Ptr           : D_List_P;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var Ch : Char;
  Begin
    Write('     ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }

  Procedure Print_Patterns(Pat_Ptr     : D_Pat_Rec_P;
                           Comp, Wild : String);
  Var Count : Integer;
  Begin
    Count := 1;
    Writeln('type <cr> for patterns');
    Readln; Writeln;
    Repeat
      Writeln('Pattern ', Count, ' :');
      With Pat_Ptr^ Do
        Begin
          Writeln('  Comp : ', Comp);
          If Comp_Len <> 0 Then
            Write('^':(Comp_Pos + 9));
          If Comp_Len > 1 Then Write('^':(Comp_Len - 1));
          Writeln;
          Writeln('  Wild : ', Wild);
          Write('^':(Wild_Pos + 9));
          If Wild_Len > 1 Then Write('^':(Wild_Len - 1));
          Writeln; Writeln;
        End;
      Pat_Ptr := Pat_Ptr^.Next;
      Count := Count + 1;
    Until Pat_Ptr = Nil
  End; { Print_Patterns }
```

```
Procedure Print_Info(Ptr : D_List_P);
Begin
  Repeat
    With Ptr^ Do
      Begin
        If D_Is_Blkd Then
          Case D_Kind Of
            D_Free : Write('Free space on ');
            D_Vol  : Write('Volume ');
            D_Temp : Write('Temporary file on ');
            D_Text : Write('Text file on ');
            D_Code : Write('Code file on ');
            D_Data : Write('Data file on ');
          End { Cases }
        Else
          Write('Unblocked volume ');
        Writeln(D_Volume);
        If Want_Patterns And (D_VPat <> Nil) Then
          Begin
            Writeln;
            Writeln('  Volume patterns:');
            Print_Patterns(D_VPat, D_Volume, Volume);
          End;
        Writeln('  Unit number .......  ', D_Unit);
        If D_Is_Blkd Then
          Begin
            If Not (D_Kind In [D_Vol, D_Free]) Then
              Writeln('  File name .........  ', D_Title);
            If D_Kind <> D_Free Then
              Begin
                If Want_Patterns And (D_FPat <> Nil) Then
                  Begin
                    Writeln('  File name patterns:');
                    Print_Patterns(D_FPat, D_Title, Title);
                  End;
                With D_Date Do
                  Writeln('  File date .........  ',
                          Month, '/', Day, '/', Year);
              End; { If D_Kind }
            Writeln('  Starting block ....  ', D_Start);
            Writeln('  File length .......  ', D_Length);
            If D_Kind = D_Vol Then
              Writeln('  Files on volume ...  ',
                      D_Num_Files)
            Else
              Writeln('  Last block size ...  ', D_End);
          End; { If D_Is_Blkd }
        End; { With Ptr^ }
    Writeln;
    Write('Type <cr> for rest of list');
    Readln; Writeln;
    Ptr := Ptr^.D_Next_Entry;
  Until Ptr = Nil;
End; { Print_Info }
```

```
Begin { D_Test }
  Repeat
    Mark(Heap_Ptr);
    Select := [];
    Writeln('Directory Lister --');
    Write('Volume and/or file name to match: ');
    Readln(Match);
    Write('Return pattern matching information? [y/n] ');
    Read(Ch); Writeln;
    Want_Patterns := Ch In ['y', 'Y'];
    If Want_Patterns Then
      Result := D_ScanTitle(Match, Volume, Title, Typ, Segs);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    GiveChoice('Temp Files ', [D_Temp]);
    GiveChoice('Free Space ', [D_Free]);
    Result := D_DirList(Match, Select, Ptr, Want_Patterns);
    Writeln;
    If Ptr <> Nil Then
      Print_Info(Ptr)
    Else
      Case Result Of
        D_Name_Error : Writeln('     Error in file name');
        D_Off_Line : Writeln('     Volume off line');
        D_Not_Found : Writeln('     File not found');
        D_Other : Writeln('     Miscellaneous error');
      End; {cases}
    Writeln;
    Repeat
      Write('Continue ? ');
      Read(Ch); Writeln;
    Until Ch In ['n', 'N', 'y', 'Y'];
    Writeln;
    Release(Heap_Ptr);
  Until Ch In ['n', 'N'];
End.  { D_Test }
```

5.0.2.2 D_Change_Name

Syntax:

```
Function D_Change_Name (D_Old_Name,
                        D_New_Name : String;
                        D_Rem_Old  : Boolean) : D_Result;
```

D_Change_Name searches for the volume or file designated by the file name contained in D_Old_Name and changes its name to the file name contained in D_New_Name.

D_Change_Name only changes one file name at a time, and thus does not accept file names containing wildcards; however, it can be combined with other Dir_Info routines to create user-defined file name changing routines which accept wildcards (see section 5.0.0.1 for details).

5.0.2.2.0 D_Change_Name Parameters and Function Result

D_Change_Name accepts the following parameters:

D_Old_Name — A string containing the name of the file to be changed. If the file name is invalid, D_Change_Name returns D_Name_Error. Note that wildcard characters are treated literally.

D_New_Name — A string containing the replacement file name. If the file name is invalid, D_Change_Name returns D_Name_Error. Note that wildcard characters are treated literally.

If D_Old_Name contains an empty file title, D_Change_Name changes the name of the volume specified by D_Old_Name to the volume name in D_New_Name; any file title in D_New_Name is ignored. If D_Old_Name contains an nonempty file title, D_Change_Name changes the name of the disk file specified by D_Old_Name to the file title in D_New_Name; any volume name in D_New_Name is ignored. If the file id in D_New_Name is empty, D_Change_Name returns D_Name_Error.

D_Rem_Old — If set to TRUE, D_Rem_Old indicates that an existing file or volume designated by the file name in D_New_Name may be removed in order to change the file name. If set to FALSE, the presence of an existing file or volume with the same name as D_New_Name aborts the name change, and D_Change_Name returns D_Exists as a function result.

D_Change_Name returns a value of type D_Result. D_Change_Name can return all scalar values defined in D_Result; the values have the following meanings:

D_Okay	No error. D_Old_Name was found and its name changed.
D_Not_Found	No such file/volume found. No match found for D_Old_Name. No change made.
D_Exists	The name change was blocked by the presence of an existing file with the same name as D_New_Name. No change made.
D_Name_Error	Illegal file name syntax in D_Old_Name or D_New_Name. No change made.
D_Off_Line	Volume/unit off-line. Volume/unit specified by D_Old_Name was not on-line. No change made.
D_Other	Unknown. D_Change_Name encountered an error it could not identify. No change made.

5.0.2.2.1 Programming Example

The following program demonstrates the use of D_Change_Name.

```
Program Change_Test;
Uses Pattern_Match, Dir_Info;
Var Rem_Old  : Boolean;
    Old, New : String;
    Ch       : Char;
    Rslt     : D_Result;
Begin
  Writeln('D_ChangeName Test --');
  Repeat
    Writeln;
    Write('Name to change : ');
    Readln(Old);
    Write('New name : ');
    Readln(New);
    Write('Remove existing files of that name ? [y/n] ');
    Read(Ch); Writeln;
    Rem_Old := Ch In ['y', 'Y'];
    Case D_ChangeName(Old, New, Rem_Old) Of
      D_Okay : Writeln('     No error');
      D_Off_Line : Writeln('     Volume off line');
      D_Name_Error : Writeln('     Error in file name');
      D_Not_Found : Writeln('     File not found');
      D_Other : Writeln('     Miscellaneous error');
    End; { cases }
    Writeln;
    Write('Continue ? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End.  { Change_Test }
```

5.0.2.2.2 Wildcard File Name Replacement

D_Change_Name does not accept wildcard file name arguments; however, it may be combined with the pattern matching information returned by D_Dir_List to implement a wildcard file name changing routine. Note that this routine must use directory locks in multitasking environments.

For example, assume that the user has the following files:

```
TEST1.TEXT
TEST12.CODE
TEST.DATA
```

... and would like to change them to:

```
OLD1A.TEXT
OLD12A.CODE
OLDA.DATA
```

This can be performed by using D_Dir_List to search for the file

name 'TEST=.='. The pattern matching information returned by D_Dir_List can be used to create new file titles; in this case, 'TEST' is replaced with 'Old', and the first '=' is replaced with the concatenation of the pattern matched by the '=' and the literal string 'A'. The part of each file title matched by the period and the second "=" wildcard is unchanged. D_Change_Name is called with the modified file title for each file matched by D_Dir_List.

The following program demonstrates the use of D_Change_Name and D_Dir_List in the construction of a specialized file name changing utility. The program accepts a file name argument containing two '=' wildcards; for each file which matches the argument, the file title is changed by swapping the string patterns matched by the two "=" wildcards.

```
Program Wild_Change;
Uses Pattern_Match, Dir_Info;
Var Heap_Ptr : ^Integer;
    Typ      : D_Name_Type;
    Segs     : Integer;
    Select   : D_Choice;
    Volume, Name,
    Match    : String;
    Result   : D_Result;
    Ch       : Char;
    Ptr      : D_List_P;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var Ch : Char;
  Begin
    Write('     ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }

  Procedure Print_Patterns(Pat_Ptr    : D_Pat_Rec_P;
                           Comp, Wild : String);
  Var Count : Integer;
  Begin
    Count := 1;
    Writeln('type <cr> for patterns');
    Readln; Writeln;
    Repeat
      Writeln('Pattern ', Count, ' :');
      With Pat_Ptr^ Do
        Begin
          Writeln('  Comp : ', Comp);
          If Comp_Len <> 0 Then
            Write('^':(Comp_Pos + 9));
          If Comp_Len > 1 Then Write('^':(Comp_Len - 1));
          Writeln;
          Writeln('  Wild : ', Wild);
          Write('^':(Wild_Pos + 9));
          If Wild_Len > 1 Then Write('^':(Wild_Len - 1));
          Writeln; Writeln;
        End;
```

```
    Pat_Ptr := Pat_Ptr^.Next;
    Count := Count + 1;
  Until Pat_Ptr = Nil;
End; { Print_Patterns }

Procedure Print_Info(Ptr           : D_List_P;
                     Want_Patterns : Boolean;
                     Volume, Name  : String);
Begin
  Repeat
    Writeln('MATCHED FILE --');
    With Ptr^ Do
      Begin
        Write(D_Volume, ':');
        If D_Is_Blkd Then
          If Length(D_Title) > 0 Then
            Write(D_Title);
        Writeln;
        If Want_Patterns And (D_VPat <> Nil) Then
          Begin
            Writeln;
            Writeln('     Volume patterns:');
            Print_Patterns(D_VPat, D_Volume, Volume);
          End;
        If D_Is_Blkd Then
          If Want_Patterns And (D_FPat <> Nil) Then
            Begin
              Writeln('     File name patterns:');
              Print_Patterns(D_FPat, D_Title, Name);
            End;
      End; { With Ptr^ }
    Writeln;
    Write('Type <cr> for rest of list');
    Readln; Writeln;
    Ptr := Ptr^.D_Next_Entry;
  Until Ptr = Nil;
End; { Print_Info }

Procedure Change(Ptr : D_List_P; Name : String);
Var I, Pos1, Len1, Pos2, Len2, Last_Pos,
    Mid_Pos, Last_Equal    : Integer;
    Pat1, Pat2, Title, New : String;

  Procedure Find_Equal(D_Title, Name : String;
                   Var Pat_Ptr       : D_Pat_Rec_P;
                   Var Pat           : String;
                   Var Pos, Len      : Integer);
  Begin
    While (Name[Pat_Ptr^.Wild_Pos] <> '=') And
      (Pat_Ptr^.Next <> Nil) Do
        Pat_Ptr := Pat_Ptr^.Next;
    With Pat_Ptr^ Do
      Begin
        If Comp_Len = 0 Then Pat := ''
        Else Pat := Copy(D_Title, Comp_Pos, Comp_Len);
        Pos := Comp_Pos;
```

```
        Len := Comp_Len;
      End;
  End; { Find_Equal }

Begin { Change }
  With Ptr^ Do
    Begin
      Find_Equal(D_Title, Name, D_FPat, Pat1, Pos1, Len1);
      If D_FPat <> Nil Then
        Begin
          D_FPat := D_FPat^.Next;
          Find_Equal(D_Title, Name, D_FPat,
                     Pat2, Pos2, Len2);
          New := D_Title;
          Last_Pos := Pos2 + Len2;
          Mid_Pos := Pos1 + Len2;
          Last_Equal := Last_Pos - Len1;
          For I := Pos1 To Mid_Pos - 1 Do
            New[I] := Pat2[I - Pos1 + 1];  { 1st '=' }
          For I := Mid_Pos To Last_Equal - 1 Do
            New[I] := D_Title[I - Len2 + Len1];
          For I := Last_Equal To Last_Pos - 1 Do
            New[I] := Pat1[I - Last_Equal + 1]; { 2nd '=' }
          New := Concat(D_Volume, ':', New);
          Title := Concat(D_Volume, ':', D_Title);
          Result := D_ChangeName(Title, New, True);
          Write(Title, '-->', New);
          Case Result Of
            D_Name_Error : Write('  Error in file name');
            D_Off_Line : Write('  Volume off line');
            D_Not_Found : Write('  File not found');
            D_Other : Write('  Miscellaneous error');
          End; {cases}
          Writeln;
        End; { if D_FPat }
    End; { with }
End; { Change }

Function Display(S, Match,
                 Volume, Name : String;
                 Select       : D_Choice) : D_List_P;
Var Ch             : Char;
    Ptr            : D_List_P;
    Want_Patterns  : Boolean;
    Result         : D_Result;
Begin { Display }
  Writeln; Writeln(S);
  Write('     Display pattern matching information ? ');
  Read(Ch); Writeln;
  Want_Patterns := Ch In ['y', 'Y'];
  Result := D_DirList(Match, Select, Ptr, True);
  If Ptr <> Nil Then
    Print_Info(Ptr, Want_Patterns, Volume, Name)
  Else
    Case Result Of
      D_Name_Error : Writeln('     Error in file name');
```

```
        D_Off_Line : Writeln('     Volume off line');
        D_Not_Found : Writeln('     File not found');
        D_Other : Writeln('     Miscellaneous error');
      End; {cases}
    Display := Ptr;
  End; { Display }

Begin { Wild_Change }
  Writeln;
  Repeat
    Mark(Heap_Ptr);
    Select := [];
    Write('File title to match (must contain two ''='' ): ');
    Readln(Match);
    Result := D_ScanTitle(Match, Volume, Name, Typ, Segs);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    Ptr := Display('Old Files :', Match,
                   Volume, Name, Select);
    If Ptr <> Nil Then
      Begin
        Repeat
          Change(Ptr, Name);
          Ptr := Ptr^.D_Next_Entry;
        Until Ptr = Nil;
        Write('Redisplay files? ');
        Read(Ch); Writeln;
        If Ch In ['y', 'Y'] Then
          Ptr := Display('New Files :', Match,
                         Volume, Name, Select);
      End;
    Writeln;
    Repeat
      Write('Continue ? ');
      Read(Ch); Writeln;
    Until Ch In ['n','N','y','Y'];
    Writeln;
    Release(Heap_Ptr);
  Until Ch In ['n', 'N'];
End.  { Wild_Change }
```

5.0.2.3 D_Change_Date

Syntax:

```
Function D_Change_Date (D_Name     : String;
                        D_New_Date : D_Date_Rec;
                        D_Select   : D_Choice) : D_Result;
```

D_Change_Date changes the file date of volumes and files whose names match the file name argument contained in D_Name. D_Change_Date accepts wildcards in its file name argument.

5.0.2.3.0 D_Change_Date Parameters and Function Result

D_Change_Date accepts the following parameters:

D_Name	A string which contains a valid file name. The file name may contain wildcards.
D_New_Date	A record of type D_Date_Rec which contains the new date. A value of 100 is not accepted by D_Change_Date in a new date. See section 5.0.0.4 for more information.
D_Select	A set of file and/or volume types as described in section 5.0.0.3. All scalar types except D_Free and D_Temp apply to D_Change_Date. Disk free spaces identified by the D_Free scalar do not contain file dates. Temporary status for files is specified by a special value in the file date field. Thus, D_Free and D_Temp are ignored if they are included in D_Select.

D_Change_Date returns a value of type D_Result. D_Change_Date can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay	No error. D_Name was found, and D_New_Date was written to the directory for the specified file or disk volume.
D_Not_Found	No such file/volume found. No match found for D_Name. No change made.
D_Name_Error	Illegal syntax in D_Name. No change made.
D_Off_Line	Volume/unit off-line. Volume/unit specified by D_Name was not on-line. No change made. This error occurs only if the volume id in D_Name specifies a single volume which is off-line. If the volume name in D_Name contains wildcards and does not match any on-line volumes, D_Change_Date returns D_Not_Found.
D_Other	Unknown error. No change made. D_Change_Date encountered an unidentified error which prevented successful completion of the operation.

5.0.2.3.1 Programming Example

The following program demonstrates the use of D_Change_Date.

```
Program Date_Test;
Uses Pattern_Match, Dir_Info;
Var Result    : D_Result;
    Ch        : Char;
    M, D, Y   : Integer;
    New_Date  : D_Date_Rec;
    Select    : D_Choice;
    File_Name : String;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var Ch : Char;
  Begin
    Write('     ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }
```

```
Begin { Date_Test }
  Select := [];
  Writeln('D_ChangeDate Test --');
  Repeat
    Writeln;
    Write('File to change : '); Readln(File_Name);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    Writeln('New date : ');
    Write('Month [1 - 12] : '); Readln(M);
    Write('Day   [1 - 31] : '); Readln(D);
    Write('Year  [0 - 99] : '); Readln(Y);
    With New_Date Do
      Begin
        Month := M;
        Day := D;
        Year := Y;
      End; { With New_Date }
    Writeln;
    Result := D_ChangeDate(File_Name, New_Date, Select);
    Case Result Of
      D_Okay : Writeln('date changed');
      D_Name_Error : Writeln('error in file name');
      D_Off_Line : Writeln('volume off line');
      D_Not_Found : Writeln('file not found');
      D_Other : Writeln('miscellaneous error');
    End; { cases }
    Writeln;
    Write('Continue ? ');
    Read(Ch); Writeln;
  Until Ch In ['n','N'];
End.  { Date_Test }
```

5.0.2.4 D_Rem_Files

Syntax:

```
Function D_Rem_Files (D_Name     : String;
                      D_Select   : D_Choice) : D_Result;
```

The D_Rem_Files function removes file objects whose names match the file name argument contained in D_Name and types match the elements included in D_Select. The file name argument may contain wild-cards. Disk files are permanently deleted from their directories. Volumes are taken off-line, but not altered in any way; off-line disk volumes may be brought back on-line merely by referencing them, while off-line serial volumes remain inaccessible until the system is reinitialized.

5.0.2.4.0 D_Rem_Files Parameters and Function Result

D_Rem_Files accepts the following parameters:

D_Name — A string containing the name of the file(s) or volume(s) to be removed.

D_Select — A set of file objects to be removed. Removal of file objects is subject to the criteria described in section 5.0.0.3. The definition of the set is as follows:

```
D_Name_Type = (D_Vol, D_Code, D_Text,
               D_Data, D_Free, D_Temp);
D_Choice    = Set Of D_Name_Type;
```

All scalar types except D_Free apply to D_Rem_Files. Disk free spaces cannot be removed from the directory; thus, D_Free is ignored if it is included in D_Select.

D_Rem_Files returns a value of type D_Result. D_Rem_Files can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay	No error. D_NAME was found. If D_Vol is included in D_Select, and a volume matches the file name argument in D_NAME, the volume is taken off-line. If D_Text, D_Code, D_Data, or D_Temp are included in D_Select, disk files of those types which match D_NAME are deleted from their directories.
D_Not_Found	No such file/volume found. No match found for D_NAME. No change made.
D_Name_Error	Illegal file name syntax in D_NAME. No change made.
D_Off_Line	Volume/unit off-line. The volume/unit specified by D_NAME was not on-line. No change made. This error occurs only if the volume id in D_NAME specifies a single volume which is off-line. If the volume id in D_NAME contains wildcards, but does not match any on-line volume, D_Rem_Files returns D_Not_Found.
D_Other	Unknown error. No change made. D_Rem_Files encountered an unidentified error which prevented successful completion of the operation.

5.0.2.4.1 Programming Example

The following program demonstrates the use of D_Rem_Files.

```
Program Rem_Test;
Uses Pattern_Match, Dir_Info;
Var Result   : D_Result;
    Select   : D_Choice;
    Ch       : Char;
    Rem_File : String;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var Ch : Char;
  Begin
    Write('     ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }

Begin { Rem_Test }
  Select := [];
  Writeln('D_RemFiles Test --');
  Repeat
    Write('File(s) to remove : ');
    Readln(Rem_File);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Temp Files ', [D_Temp]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    Result := D_RemFiles(Rem_File, Select);
    Case Result Of
      D_Okay : Writeln('files removed');
      D_Name_Error : Writeln('error in file name');
      D_Off_Line : Writeln('volume off line');
      D_Not_Found : Writeln('file not found');
      D_Other : Writeln('miscellaneous error');
    End; { cases }
    Writeln;
    Write('Continue ? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End.  { Rem_Test }
```

5.0.2.5 D_Change_End

Syntax:

```
Function D_Change_End (D_File_Name : String;
                       D_New_End   : Integer) : D_Result;
```

D_Change_End searches for a file designated by the file name contained in D_File_Name and changes the number of valid data bytes in its last block to the value contained in New_End.

D_Change_End only operates on disk files, changing only one at a time. Thus, it does not accept file names containing wildcards; however, it may be combined with other Dir_Info routines to create user-defined file name changing routines which accept wildcards (see section 5.0.2.2.2 for an example).

5.0.2.5.0 D_Change_End Parameters and Function Result

D_Change_End accepts the following parameters:

D_File_Name	A string containing the name of the disk file to be changed. If the file name is either invalid or empty, D_Change_End returns D_Name_Error. Note that wildcard characters are interpreted literally.
D_New_End	An integer containing the number of valid bytes in the last record of the file designated by D_File_Name. Change_End returns D_Other if this value is not between 1 and 512.

D_Change_End returns a value of type D_Result. D_Change_End can return only some of the scalar values defined in D_Result; these values have the following meanings:

D_Okay	No error. D_Old_Name was found and its name changed.
D_Not_Found	No such file/volume found. No match found for D_Old_Name. No change made.
D_Name_Error	Illegal file name syntax in D_Old_Name or D_New_Name. No change made.
D_Off_Line	Volume/unit off-line. Volume/unit specified by D_Old_Name was not on-line. No change made.
D_Other	Invalid range or Unknown. Either D_New_End was not in the range 1..512 or the error could not be identified. No change made.

5.0.2.5.1 Programming Example

The following program demonstrates the use of D_Change_End.

```
Program End_Test;
Uses Pattern_Match, Dir_Info;
Var New_End : Integer;
    Name    : String;
    Ch      : Char;
Begin
  Writeln('D_Change_End Test --');
  Repeat
    Writeln;
    Write('Name to change : ');
    Readln(Name);
    Write('New end of file : ');
    Readln (New_End);
    Case D_ChangeEnd(Name, New_End) Of
      D_Okay : Writeln('     No error');
      D_Off_Line : Writeln('     Volume off line');
      D_Name_Error : Writeln('     Error in file name');
      D_Not_Found : Writeln('     File not found');
      D_Other : Writeln('     Bad end value');
    End; { cases }
    Writeln;
    Write('Continue ? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End.  { End_Test }
```

5.0.2.6 D_Init

Syntax:

```
Procedure D_Init;
```

Initialization of the Dir_Info unit is performed automatically when running under the Advanced Operating System. However, when running under other UCSD Pascal systems, D_Init must be called before any Dir_Info routines may be used; failure to do so may result in unexpected program behavior.

5.0.2.7 D_Lock

Syntax:

```
Procedure D_Lock;
```

D_Lock grants exclusive directory access rights to the task that executes it; however, a task may have to wait until another task releases the directory lock before it may continue execution past its call to D_Lock. See section 5.0.2.8.0 for an example of D_Lock.

NOTE - D_Lock calls should always be matched with D_Release calls to prevent system deadlocks.

The Dir_Info routines D_Lock and D_Release are provided for use in multitasking environments. When used properly, they ensure mutually exclusive access to directory information. See section 5.0.1.4 for more information on multitasking routines.

5.0.2.8 D_Release

Syntax:

```
Procedure D_Release;
```

D_Release releases exclusive access rights to the directory. Tasks already waiting for directory access are automatically awakened when the directory becomes available by a call to D_Release.

The Dir_Info routines D_Lock and D_Release are provided for use in multitasking environments. When used properly, they ensure mutually exclusive access to directory information. See section 5.0.1.4 for more information on multitasking routines.

5.0.2.8.0 Programming Example

The following program demonstrates the use of D_Lock and D_Release.

```
Program Protect_Test;
Uses Pattern_Match, Dir_Info;
Const Stack_Size = 2000;
Var Pid      : Processid;
    Old,
    New      : String;
    Date     : D_Date_Rec;
    M, D, Y  : Integer;
    Ch       : Char;

  Process Change_And_Check(Old, New : String;
                           Date     : D_Date_Rec);
  Var Result : D_Result;
  Begin { Change_And_Check }
    D_Lock;          { beginning of critical section }
    Result := D_ChangeDate(Old, Date, [D_Vol..D_Data]);
    If Result = D_Okay Then
      Result := D_ChangeName(Old, New, True);
    D_Release;       { end of critical section }
  End; { Change_And_Check }

Begin { Protect_Test }
  Repeat
    Write('Old file name: ');
    Readln(Old);
    Write('New file name: ');
    Readln(New);
    Writeln('New date:');
    Write('   Month: ');
    Readln(M);
    Write('     Day: ');
    Readln(D);
    Write('    Year: ');
    Readln(Y);
    With Date Do
      Begin
        Month := M;
        Day := D;
        Year := Y;
      End;
    Start(Change_And_Check(Old, New, Date), Pid, Stack_Size);
    Write('Start another? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End.  {Protect_Test }
```

5.0.3 The Directory Information Unit Interface

This section displays the text of the interface section belonging to the directory information unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "D_". The P_Pat_Rec_P identifier is declared in the Pattern_Match unit; it is a pointer to a list of pattern information records.

```
Unit Dir_Info;
Interface
Uses Pattern_Match;

Type

  D_Date_Rec = Packed Record
                 Month : 0..12;
                 Day   : 0..31;
                 Year  : 0..100;
               End;

  D_Name_Type = (D_Vol, D_Code, D_Text,
                 D_Data, D_Temp, D_Free);

  D_Choice = Set of D_Name_Type;

  D_List_P = ^D_List;
  D_List = Record
             D_Unit          : Integer;
             D_Volume        : String[7];
             D_VPat          : P_Pat_Rec_P;
             D_Next_Entry    : D_List_P;
             Case D_Is_Blkd : Boolean Of
              True: (D_Start,
                     D_Length : Integer;
                     Case D_Kind : D_Name_Type Of
                      D_Vol,
                      D_Temp,
                      D_Code,
                      D_Text,
                      D_Data: (D_Title : String[15];
                               D_FPat  : P_Pat_Rec_P;
                               D_Date  : D_Date_Rec;
                               Case D_Name_Type of
                                D_Vol: (D_Num_Files : Integer);
                                D_Temp,
                                D_Code,
                                D_Test,
                                D_Data: (D_End : Integer)));
           End;

  D_Result = (D_Okay, D_Not_Found, D_Exists,
              D_Name_Error, D_Off_Line, D_Other);
```

```
Function D_Dir_List(D_Name    : String;
                    D_Select  : D_Choice;
                Var D_Ptr     : D_List_P;
                    D_PInfo   : Boolean) : D_Result;
    {Creates a pointer to a list of names of specified NameTypes
     (D_Select), matching specified D_Name (wildcard characters
     allowed). Includes information about pattern matching if
     requested (by D_PInfo)}


Function D_Scan_Title(D_Name       : String;
                  Var D_VolID,
                      D_TitleID    : String;
                  Var D_Type       : D_Name_Type;
                  Var D_Segs       : Integer) : D_Result;
    {Parses the file name in D_Name into volume, title, type, and
     length specifier}


Function D_Change_Name(D_Old_Name,
                       D_New_Name : String;
                       D_Rem_Old  : Boolean) : D_Result;
    {Changes file name in D_Old_Name to name in D_New_Name,
     removing already existing files of name in D_New_Name if
     D_Rem_Old is set}


Function D_Change_Date(D_Name    : String;
                       D_NewDate : D_Date_Rec;
                       D_Select  : D_Choice) : D_Result;
    {Changes date of directory or file name in D_Name to date
     specified by D_NewDate. D_Name may contain wildcards}


Function D_Rem_Files (D_Name   : String;
                      D_Select : D_Choice) : D_Result;
    {Removes file of specified name (wildcards allowed)}


Function D_Change_End(D_File_Name : String;
                      D_New_End   : Integer) : D_Result;
    {Changes the number of valid data bytes in the last block of
     the file name in D_File_Name to the value of D_New_End}


Procedure D_Init;
    {Initializes Dir_Info unit}


Procedure D_Lock;
Procedure D_Release;
    {Provide means to limit use of DirInfo routines to one task at
     a time in multitasking environments}
```

5.1 The File Information Unit

This file information unit (named File_Info) allows user programs to obtain block file attributes normally accessible only to system programs.

The file information unit provides the following information on block files:

Resident Volume Name - The name of the volume containing the external file (and whether or not the volume is a disk volume).

Resident Disk Unit - The number of the unit containing the external file.

Title - The file title of the external file.

Date - The date assigned to the external file.

Starting Block - The absolute block number of the first block in the external file.

Length - The number of blocks of data in the external file.

Section 5.1.0 describes the File_Info routines. Chapter 5.1.1 displays the text comprising the file information unit's interface section. Chapter 5.1.2 contains a programming example.

NOTE - File_Info only accepts block files as arguments; other file types are not accepted. See the Programmer's Manual for a description of block files and the System User's Manual for a description of the file system. This section uses the term title to specify the file title and suffix combined; this differs from the terminology used in the System User's Manual.

5.1.0 Using the File Information Unit

This section describes how to use the file information unit. See section 5.1.1 for type and routine declarations and section 5.1.2 for a programming example.

The file information unit is available as an option with the AOS. It is imported into a program by the "USES File_Info;" statement (see the Programmer's Manual for details). It may reside in the intrinsics library, the system library, or a user library. All imported identifiers begin with "F_" to prevent conflicts with program identifiers.

The file information unit uses no global variables and no other units.

All File_Info routines return a value reflecting the result of the operation. The result indicates either that the operation was successful or that the specified file is not open.

File_Info defines a scalar type to describe the file status result:

```
Type F_Result = (F_Okay, F_Not_Open);
```

NOTE - When a File_Info routine returns a file status result indicating a closed file, the other information returned is not valid.

5.1.0.0 Resident Volume

The F_Volume and F_Is_Blocked routines return information describing the name and type respectively of the volume containing the external file.

F_Volume accepts a block file and a string variable as parameters, and returns a file status result. The volume name of the external file is returned in the string parameter. If the external file lacks a defined volume name, F_Volume returns a volume id constructed from a unit number (e.g. "#3").

F_Is_Blocked accepts a block file and a boolean variable as parameters, and returns a file status result. The boolean parameter is set to TRUE if the volume containing the external file is a block-structured (i.e. disk) volume; otherwise, it is set to FALSE.

5.1.0.1 Resident Unit

F_Unit accepts a block file and an integer variable as parameters, and returns a file status result. The number of the unit containing the external file is returned in the integer parameter.

5.1.0.2 Title

F_File_Title accepts a block file and a string variable as parameters, and returns a file status result. The file title of the external file is returned in the string parameter. If the external file is a volume, F_File_Title returns an empty file title.

5.1.0.3 Date

F_Date accepts a block file and a date record as parameters, and returns a file status result.

The date record is declared in the interface section as follows:

```
Type F_Date_Rec = Packed Record
                    Month : 0..12;
                    Day   : 0..31;
                    Year  : 0..100;
                  End; { F_Date_Rec }
```

The current date of the external file is returned in the date record parameter. If the external file is a volume, the returned date record contains all 0's. If the external file is a temporary disk file, the month and day fields contain 0's and the year field contains 100.

5.1.0.4 Length

F_Length accepts a block file and an integer variable as parameters, and returns a file status result. The number of blocks of data in the external file is returned in the integer parameter. Note that this value may differ from the amount of disk space allocated for a disk file when a file is being generated.

If the external file is a disk volume containing a disk directory, F_Length returns the total number of blocks on the volume. If the external file is a volume lacking a disk directory, F_Length returns MAXINT as the file length.

5.1.0.5 Starting Block

F_Start accepts a block file and an integer variable as parameters, and returns a file status result. The block number of the first block in the external file is returned in the integer parameter.

If the external file is a disk file, the starting block number is relative to the first block on the enclosing disk volume. If the external file is a volume, F_Start returns 0 as the starting block.

5.1.1 The File Information Unit Interface

This section displays the text of the interface section belonging to the file information unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "F_" to prevent conflicts with host program identifiers.

```
Unit FileInfo;
Interface

  Type F_Result    = (F_Okay, F_Not_Open);
       F_File_Type = File;
       F_Date_Rec  = Packed Record
                       Month : 0..12;
                       Day   : 0..31;
                       Year  : 0..100;
                     End; { F_Date_Rec }


Function F_Length (Var Fid         : F_File_Type;
                   Var File_Length : Integer)    : F_Result;
    {Returns the length of the file attached to the Fid
     identifier. If there is no file opened to Fid, the
     function result is returned F_Not_Open and File_Length is
     undefined.}


Function F_Unit (Var Fid       : F_File_Type;
                 Var File_Unit : Integer)    : F_Result;
    {Returns the unit containing the file attached to the Fid
     identifier. If there is no file opened to Fid, the
     function result is returned F_Not_Open and File_Unit is
     undefined.}


Function F_Volume (Var Fid         : F_File_Type;
                   Var File_Volume : String)  : F_Result;
    {Returns the name of the volume containing the file
     attached to the Fid identifier. If there is no file
     opened to Fid, the function result is returned F_Not_Open
     and File_Volume is undefined.}


Function F_File_Title (Var Fid        : F_File_Type;
                       Var File_Title : String)  : F_Result;
    {Returns the title (with suffix) of the file attached to
     the Fid identifier. If there is no file opened to Fid,
     the function result is returned F_Not_Open and File_Title
     is undefined.}
```

```
Function F_Start (Var Fid        : F_File_Type;
                  Var File_Start : Integer)    : F_Result;
    {Returns the length (in blocks) of the file attached to
     the Fid identifier. If there is no file opened to Fid,
     the function result is returned F_Not_Open and File_Start
     is undefined.}


Function F_Is_Blocked (Var Fid       : F_File_Type;
                       Var F_Is_Blkd : Boolean) : F_Result;
    {Returns a boolean that is TRUE if the file attached to
     the Fid identifier is located on a block-structured unit.
     If there is no file opened to Fid, the function result is
     returned F_Not_Open.}


Function F_Date (Var Fid        : F_File_Type;
                 Var File_Date  : F_Date_Rec) : F_Result;
    {Returns a record indicating the last access date for the
     file attached to the Fid identifier. If there is no file
     opened to Fid, the function result is returned F_Not_
     Open.}
```

5.1.2 Programming Example

The following program demonstrates the capabilities of the file information unit.

```
{$I-}
Program File_Demo;
Uses File_Info;
Var Fid : File;
    Name,
    Title,
    Volume : String;
    Start,
    Blocks,
    Unit_Num : Integer;
    Is_Blocked : Boolean;
    Date : F_Date_Rec;
    Junk : F_Result;
Begin
  Write ('File Name ? ');
  Readln (Name);
  If Length (Name) = 0 Then
    Exit (File_Demo);
  Reset (Fid, Name);
  If F_Volume (Fid, Volume) = F_Okay Then
    Begin
      Junk := F_Is_Blocked (Fid, Is_Blocked);
      Junk := F_Unit (Fid, Unit_Num);
      Junk := F_File_Title (Fid, Title);
      Junk := F_Date (Fid, Date);
      Junk := F_Length (Fid, Blocks);
      Junk := F_Start (Fid, Start);
      Writeln;
      Write (Volume, ': (Unit ', Unit_Num);
      If Is_Blocked Then
        Begin
          Writeln (' blocked)');
          Writeln (Title, Blocks:7,
                   Date.Month:4, -Date.Day, -Date.Year,
                   Start:6);
        End {of If Is_Blocked}
      Else
        Writeln (' serial)');
    End {of If F_Volume}
  Else
    Writeln ('No such file');
End {of File_Demo}.
```

VI. I/O ROUTINES

This chapter describes a unit which performs printer spooling.

6.0 The Print Spooler Unit

The print spooler unit (named Spool_Unit) contains routines that list the contents of a specified text file on a specified serial output device. The spooler queues print requests received while it is processing other print requests. Queued requests are processed on a first come, first served basis. Printing is performed asynchronously with respect to the execution of the program.

The print spooler unit provides the following capabilities:

-- Text file queueing and concurrent printing.

-- Print suspend and resume functions.

-- Spooler status functions.

6.0.0 Using the Print Spooler Unit

The print spooler unit is provided with each AOS release. It is imported into a program by the "USES Spooler;" statement (see the Programmer's Manual for details). It may be installed in the intrinsics library, the system library, or the user library. All imported identifiers begin with "Sp" to prevent conflicts with program identifiers. The routines provided by this unit are described below.

The print spooler unit maintains 700 words of global variables. It uses no other units.

The print spooler is implemented as a concurrent task that is started during the execution of the Spool_Unit initialization section. The task is terminated during the execution of the unit termination section after all print requests have been satisfied. Since the system prevents program termination until all of a program's tasks have terminated, a program that uses the print spooler unit cannot terminate until all print requests have been processed.

NOTE - The printer spooler task executes at priority 64. Since most user programs execute at priority 128, the printer spooler task is locked out during compute-bound operations. The SU_Delay intrinsic (see section 2.4.0.1) may be used to suspend compute-bound tasks for short periods of time.

NOTE - When the print spooler is installed in the intrinsics library, the spooler task is initiated at system bootstrap time and terminated at system halt time. Programs that use the print spooler under these conditions may terminate, and other programs may execute, without waiting for the print spooler to process all

print requests.

6.0.0.0 Print Spooler Status Results

Some Spool_Unit routines return a value reflecting the result of the operation. The result indicates either the status of the print spooler or whether the operation was successful.

Spool_Unit defines a scalar type to describe the spooler status result:

```
Sp_Result = (Sp_Go, Sp_Stop, Sp_Queued,
             Sp_Full, Sp_Not_Found);
```

The meanings of these scalars are described along with the routines that return them.

6.0.0.1 Spool_File

The Spool_File function submits a print request and returns a result indicating the disposition of the request. Spool_File accepts four parameters: the name of the text file to list, the printer paper page size, the size of the print area on each page, and the I/O unit number of the print device. If the specified text file cannot be opened, the function returns the value Sp_Not_Found. Otherwise, an attempt is made to queue the print request for processing. If the print request queue is full (10 entries are allowed), the print request is denied and Spool_File returns the value Sp_Full. If the print request is accepted, the function returns the value Sp_Queued.

The file name parameter contains the name of the text file, including the ".TEXT" suffix, to be listed.

The printer paper page size and the size of the desired print area are expressed as a number of lines of print. This allows the print spooler to operate correctly with different types of printer paper. Most printer paper accomodates 66 lines per page. The standard print area for this paper contains 60 lines. The print spooler uses linefeeds to skip the remaining 6 lines. Continuous printing may be achieved by specifying the same values for the page size and the print area.

The I/O unit number should address a serial output volume. Different print requests may name different serial output devices. Note that print requests are processed one-at-a-time. Thus, addressing two print requests to different devices does not result in simultaneous output to both devices.

WARNING - No validation is performed on the I/O unit number. If it addresses a block-structured device, valuable data may be overwritten. The UNITSTATUS intrinsic (see the Programmer's Manual) may be used to verify that an I/O unit addresses a serial output device.

NOTE - A file queued for printing should not be removed or relocated until it has been completely printed. The volume containing the file should remain online throughout this period.

6.0.0.2 Spool_Stop and Spool_Restart

The Spool_Stop procedure suspends printing by the print spooler. Print requests continue to be accepted until the print request queue is full. The Spool_Restart resumes printing by the print spooler.

6.0.0.3 Spool_Status

The Spool_Status function returns a result of type SP_Result. The current status of the print spooler is returned as the function value. A value of SP_Go indicates that the print spooler is either processing a print request or ready to process one. A value of SP_Stop indicates that the print spooler is suspended (see section 6.0.0.2).

6.0.1 The Print Spooler Unit Interface

This section displays the text of the interface section belonging to the print spooler unit. The interface section text may also be viewed with the Libmap utility (see the System User's Manual for details). Note that all identifiers begin with "Sp" to prevent conflicts with host program identifiers.

```
Unit Spool_Unit;
Interface

  Type Sp_Result = (Sp_Go, Sp_Stop,
                    Sp_Queued, Sp_Full, Sp_Not_Found);

  Function Spool_Status : Sp_Result;
    {Gives the status of the spooler}

  Procedure Spool_Restart;
    {Re-starts the spooler if it is suspended}

  Procedure Spool_Stop;
    {Stops the spooler if it isn't already}

  Function Spool_File (Name : String;
                       Print_Space,
                       Page_Size,
                       Dest : Integer) : Sp_Result;
    {Queues a file for the spooler to print}
```

6.0.2 Programming Example

The following program demonstrates the use of the Spool_Unit in the construction of a program that outputs files to a printer. Note that if the Spool_Unit is installed in the intrinsics library, the program may terminate before the print request queue is exhausted. This allows concurrent printing and system operation.

```
Program Dump_Files;
Uses Spool_Unit;
Const Lines_In_Page  = 66;
      Print_Per_Page = 60;       {Leave 6 lines}
      Out_Unit       = 6;        {Printer:}
Var Name : String;

  Procedure Get_File_Name (Var Name : String);
  Begin
    Writeln;
    Write ('File to print ? ');
    Readln (Name);
  End {Get_File_Name};

Begin {Dump_Files}
  Spool_Stop;
  Get_File_Name (Name);
  While Length (Name) <> 0 Do
    Begin
      Get_File_Name (Name);
      Case Spool_File (Name, Print_Per_Page,
                       Lines_In_Page, Out_Unit) Of
        Sp_Not_Found : Writeln (Name, ' not found');
        Sp_Full      : Writeln ('Queue is full');
        Sp_Queued    : Writeln (Name, ' queued');
      End {Case};
    End {While};
  Spool_Restart;
End {Dump_Files}.
```

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: 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

ADDENDA FOR THE LIBRARY USER'S MANUAL

Section 2.0.0.0.0 (page 7)

The editor does not write to either the pre-declared file OUTPUT or to the STANOUT: unit. Therefore, output and t-output options may not be used to manipulate the editor's output stream. However, since it reads from STANIN:, input and t-input options may be used to manipulate the editor's input stream.

Section 2.0.0.0.1 (pages 7-8),
Section 2.0.0.0.2 (pages 8-9)

It should be noted that when a program is called, the data space and code segments used by the calling program are left in memory. Thus, the called program must operate in whatever memory is left.