VOLITION SYSTEMS
P.O. Box 1236, Del Mar, CA 92014 USA • (619) 481-2286

Modula-2 Installation Notes - 0.3j for the IBM PC

This document presents useful system information which is not covered by the regular documentation. The following topics are discussed:

● Features not supported ● Basic key commands ● RAM disk configuration ● Serial and parallel ports ● 8087 support ● p-NIX shell ● ASE installation notes

Features Not Supported In This Release

This section describes system features that are not supported in the current release:

Serial and parallel port interrupts (i.e. IOTRANSFER vectors 4 and 7) are not implemented.

There are currently no facilities for creating your own device drivers to link into the interpreter. Documentation will be available in the near future for this facility, but it will probably require that you have the version IV p-System and its assembler.

Hard disks are not supported.

Assembly procedure linkage is not supported.

Graphics are not supported.

Basic Key Commands

There are some basic key commands you should know about. Most of these command are described more fully in section 2.3 of the **Modula Operating System Manual.**

Typing the Del key while holding down the Ctrl and Alt keys reboots the system.

The Enter key is the <return> command.

The Backspace key erases the last character typed in to a prompt.

Typing control-Backspace erases all characters typed in to a prompt.

The Esc key is the <escape> command.

Control-C serves as both the <eof> and <accept> commands.

Control-S is the start/stop command which suspends console output.

Control-F is the flush command which discards all console output.

Control-Break is the program break command which terminates the current program with an execution error.

NOTE- The ASE key definitions are described further on in this document.

WARNING- The program break key control-Break should not be used while a Modula program is starting up: it may crash the system. Once a Modula program is running, control-Break can be safely used.

RAM Disk Configuration

The RAM disk driver provided in the preconfigured interpreters has the disk length set to 0, thus preventing a RAM disk from being allocated. To create a RAM disk, you must run the IBMUtil program and set the RAM disk length (see the **Implementation Guide** for details).

Note that if you set the RAM disk length to "All available", the switch settings on your memory expansion boards must reflect the actual amount of memory on board; otherwise, the RAM disk driver may cause the system to halt (while starting) with a hardware parity error while it searches for the high limit of available memory. This problem can be avoided by setting the RAM disk length to a fixed number of blocks.

Serial and Parallel Ports

The serial and parallel port drivers supplied in IOCOMS and IOLPTS do not perform buffering of input or output characters.

8087 Support

The 8087 numeric coprocessor is required in order to perform floating point arithmetic; there is no software floating point support. Floating point is accessed via the data type REAL in both Pascal and Modula. The 64-bit internal format results in 17 digits of precision. Math functions in both Pascal and Modula-2 use the 8087's math support.

When an 8087 is installed, the Pascal compiler supports 64-bit "long" integers via the data type WIDE. Wide integers may be freely mixed in expressions with regular integers and reals. Integer constants with a magnitude larger than 32767 are treated as wide constants. The standard procedures TRUNC and ROUND work with wide (rather than regular) integers. Wides may not be used as count variables or bound expressions in FOR statements.

p-NIX Shell

The shell command 'ed' cannot be used with nonseparate code and data interpreters because there is insufficient memory to run ASE on top of the shell. On separate code and data interpreters with 64K bytes (or more) of code space, 'ed' works fine.

ASE Installation Notes

ASE is preconfigured for the IBM PC and already installed on the SYS disk as SYSTEM.EDITOR. The predefined ASE key definitions are described below.

New ASE Commands

Version 0.9 of ASE contains one feature that is not described in the **ASE User's Manual**. The commands InsertCh and DeleteCh can be used from the main editor prompt; they are no longer restricted to being used in the X(change command. See the description of the X(change command for more information on InsertCh and DeleteCh.

ASE Key Definitions

The following table displays the ASE key definitions for the IBM PC. Note that the "prefixed" keys are actually one-key sequences; the <ctrl> or <alt> keys are pressed at the same time as the companion key to perform the command.

If you're interested in knowing what character codes are generated by these key definitions, try the S(et E(nvironment E(nter Fn Key command. See the manual for details.

Command	Key	Command	Key
'<'	<	'>'	>
'<'	,	'>'	.
'?'	?	del	ctrl-backspace
Accept	ctrl-c	Adjust	A and a
BackSpace	backspace	BeginLine	B and b
Copy	C and c	Delete	D and d
DeleteChar	Del	Down	down arrow
Edit	E and e	Equal	=
Escape	Esc	Find	F and f
GetChar	g	GetAgain	G
Home	Home	Insert	I and i
InsertChar	Ins	Jump	J and j
Kolumn	K and k	Left	left arrow
LineEnd	L and l	Margin	M and m
Next	N and n	OppPage	O and o
Page	P and p	Quit	Q and q
Replace	R and r	Return	Enter
Right	right arrow	Set	S and s
Slash	/	Space	space bar
Tab	tab	ToDisk	T and t
Up	up arrow	UpTop	U and u
Verify	V and v	WordMove	W and w
Xchange	X and x	Zap	Z and z
f1	F1	f2	F2
f3	F3	f4	F4
f5	F5	f6	F6
f7	F7	f8	F8
Record	F9	Takeup	F10

Alternate (nonprinting) Key Commands

BeginLine	ctrl-b	Column1	ctrl-o
Delete	alt-d	DirChange	ctrl-x
GetChar	ctrl-g	GetAgain	ctrl-n
Insert	alt-i	LineEnd	ctrl-l
OppPage	PgUp	Page	PgDn
UpTop	ctrl-u	WordMove	ctrl-w

WARNING- Beware of the NumLock key. If you accidentally type it, the arrow keys fail to work correctly in ASE. If this happens, type NumLock again and the problem should disappear.

Modula-2 User's Manual

Release:	0.3
Date:	26 August 1983
Author:	Richard Gleaves

Distributed by Springer-Verlag
Berlin Heidelberg New York Tokyo
ISBN 3-540-14008-5

Acknowledgements

The Modula-2 language was designed by Niklaus Wirth and his colleagues at the Institut fur Informatik, ETH Zurich.

The Modula-2 compiler and library for UCSD Pascal are the work of Roger Sumner and Joel McCormack.

The Modula-2 p-code interpreters were written by Mark Allen, Randy Bush, Joel McCormack, and Roger Sumner (6502, 68000, Z80, and 8086 respectively).

Al Hoffman of Apple Computer, Bill Bonham of Sage Computer, and Erik Smith of Scenic Computer contributed valuable assistance to the Modula-2 development effort. Special thanks are due Jim Merritt for his careful review of the manual.

This document was produced with the ASE text editor and the Sprinter-2 text formatter.

Table Of Contents

1 Preface

Modula-2 is a general purpose programming language designed primarily for systems implementation. Based on the language Pascal, Modula-2 is suitable for programming entire computer systems, from high-level machine-independent application programs down to low-level machine-specific software such as device drivers. Modula-2 also provides facilities for constructing large programs from separately compiled parts written by different programmers.

Modula-2 is the third in a series of languages designed by Niklaus Wirth over the past decade. The first and most famous is Pascal. Originally intended as a teaching language, Pascal elegantly embodied the then-new principles of structured programming, and was embraced by both educators and computer professionals as the language of choice in a wide range of computing applications. Unfortunately, Pascal's rampant use in applications beyond its intended scope resulted in the development of several incompatible dialects, and — as its popularity increased — criticism of the language's "limitations".

Wirth's next language was named MODULA (an acronym for "modular language"). MODULA was designed as a special purpose language for programming small real-time control systems. It consisted of a minimal subset of Pascal, to which was added the module structure, an improved syntax, and facilities for multiprocessing and low-level machine access. Because of its bare-bones nature and narrow scope of use, MODULA never received the attention Pascal garnered; however, the module concept gained recognition as a significant programming construct.

Wirth's latest language, Modula-2, has inherited the best features of its predecessors. It combines the module concept, improved syntax, and low-level programming facilities of MODULA with the general utility of Pascal. In addition, the design of Modula-2 systematically addresses Pascal's problems.

2 Scope of This Manual

This manual describes Volition Systems' implementation of the Modula-2 language for the version II UCSD Pascal system. It is written for the Pascal programmer who wishes to learn the Modula-2 language and write Modula-2 programs for the UCSD Pascal system. Familiarity with UCSD Pascal will prove useful, as Modula-2 runs under the UCSD Pascal operating system.

This manual is neither a reference manual for the Modula-2 language nor a user's manual for the UCSD Pascal system. It should be used with the book **Programming in Modula-2** and a UCSD Pascal system manual.

The language tutorial contained in this manual is designed for readers who are already familiar with Pascal. This approach was chosen for its ability to teach Pascal programmers Modula-2 with minimal effort. If you do not know Pascal, the introductory chapters of **Programming in Modula-2** provide a basic introduction to the Modula-2 language.

3 Organization

This manual consists of six separate documents. Note that each document has its own table of contents and index.

- **Modula-2 User's Manual:** An introduction to the remaining documents. In particular, it explains how to use the manual.
- **Introduction to Modula-2:** An introduction to the Modula-2 language for Pascal programmers. It describes differences between Modula-2 and Pascal: concepts unique to Modula-2 are presented in tutorial fashion, while minor language differences are organized for ease of reference.
- **Standard Library:** Describes the standard library modules provided with the Modula-2 system. The standard library modules provide basic system facilities; they constitute a portable operating system for all Modula-2 programs.
- **Utility Library:** Describes additional library modules provided with the system. These modules provide miscellaneous system facilities. The utility library will grow as new library modules are written and incorporated into subsequent releases.
- **The Modula-2 System:** Presents the Modula-2 language implementation for the UCSD Pascal system. It describes the library and Modula-2 compiler, and explains how to use the system.
- **Implementation Guide:** Presents information unique to the host computer system. This section includes an installation guide, machine-level representations of data types, and system-dependent facilities.

4 How to Use This Manual

The first step is to learn how to run Modula-2 programs on your system. The **Implementation Guide** explains how to install the Modula-2 system. Chapter 4 in **The Modula-2 System** document explains how to compile and execute Modula-2 programs (the sections on library management may be skipped initially, as they assume an understanding of Modula-2's separate compilation facilities).

If you are not familiar with Modula-2, read **Introduction To Modula-2** and the book **Programming in Modula-2.** As your grasp of the language improves, refer to **The Modula-2 System** for implementation and operation details.

Before you design or write any serious programs, be sure to read all of **The Modula-2 System.** In general, it presents information which contributes to the efficiency of both programs and programmers. In particular, section 3.2 identifies implementation differences from the Modula-2 language definition.

Standard Library and **Utility Library** present a wide range of system facilities available to Modula-2 programs. Skim through both of these documents to familiarize yourself with the various library modules; later, when you wish to use a specific module, go back and read the appropriate section in detail. (After a while, using many of these modules becomes second nature, and you will no longer need to consult the manual.)

If you have any trouble finding something in the manual, keep in mind that each document has a separate table of contents and index; what you need to know may be described in a different document than the one you are currently reading. (This problem vanishes with time, as each document covers a well-defined topic; once you understand the manual structure, you will be able to immediately consult the proper document.)

5 Notation

This section describes the notation used in this manual.

Programming language manuals normally define a notation for specifying the syntax of a language. Such a notation is not presented here, as this manual provides only an informal description of the Modula-2 language; the book **Programming in Modula-2** contains a complete language definition. It uses a notation known as "EBNF" (short for "Extended Backus-Naur Form") to describe the syntax of Modula-2 programs.

This manual defines a number of terms for describing the Modula-2 language and its implementation. When new terms are introduced, they appear in **boldface** and are followed by either a definition or a reference to the defining section.

NOTE- Paragraphs beginning with the word **NOTE** contain interesting or useful information related to the current topic.

WARNING- Paragraphs beginning with the word **WARNING** point out potential problems associated with the current topic.

Intra-document references have the form "x.y.z. ...", where x, y, and z denote digits. The first digit indicates the chapter; subsequent digits indicate sections within the chapter. For instance, the phrase "see 3.4" refers to section 4 in chapter 3.

Inter-document references refer to the external document by (boldfaced) name. For instance, the phrase "see 1.4 in **The Modula-2 System**" refers to section 1.4 in the document named "The Modula-2 System".

6 Terminology

This section defines terms used in **The Modula-2 System** and the **Implementation Guide.** Most of this terminology is inherited from the UCSD Pascal environment.

The following terms are used to describe file I/O in Modula-2: **file name, file block, block number, unit,** and **unit number.**

A **file name** is a character string that conforms to the file naming conventions of the UCSD Pascal file system.

A **file title** is the part of a disk file name that is not a **file suffix.** For instance, the file name "LIB.TEXT" contains the file title "LIB" and the file suffix ".TEXT".

A **file block** is the basic unit of disk file storage; a block contains 512 bytes. A **block number** is a number specifying a file block within a disk file; the first block of a disk file is block 0.

A **unit** corresponds to a physical I/O device. Each unit is identified by a unique **unit number.** For instance, unit 1 is the system console, unit 6 is the printer, and units 4 and 5 are the disk drives.

NOTE- The UCSD Pascal system manual contains additional information on the I/O system.

The following terms are used to describe the operation of the compiler and library manager: **<cr>**, **<esc>**, and **<space>.** These terms refer to keyboard commands. <cr> denotes the RETURN key, <esc> the ESCAPE key, and <space> the space bar.

Index

Introduction to

Modula-2

Release:	0.3
Date:	26 August 1983
Author:	Richard Gleaves

Table Of Contents

1 Introduction

Modula-2 is a Pascal-based general purpose programming language. While it includes most of Pascal's features, Modula-2 differs from Pascal in three ways:

- It extends Pascal upwards to encompass system design. Modula-2 is capable of expressing large software systems without requiring support from an underlying operating system. (In fact, it is an excellent language for writing operating systems.)
- It extends Pascal downwards to allow machine-level programming. Modula-2 eliminates the need for assembly language in the lowest levels of a computer system.
- It introduces a number of minor changes to Pascal which simplify programming and improve program readability and efficiency.

Modula-2 superficially resembles Ada and the growing crowd of Pascal supersets. Wirth himself credits Xerox's Mesa language as a design influence: many of Modula-2's features are borrowed directly from Mesa. Beyond these similarities, however, Modula-2 departs from the "more is better" philosophy of most of its cousins. Modula-2's design philosophy benefits more from a close comparison with that of the older "C" programming language.

C has proven that a small, expressive language can be efficient enough to displace assembly language, yet simple enough to be preferable to many so-called "powerful" high-level languages. However, its cryptic syntax and weak type checking are serious deficiencies, especially in light of Pascal's proven reputation for clarity, safety, and rigorous design.

Like C, Modula-2 provides facilities for relaxed type checking and direct access to memory words and addresses, enabling it to replace assembly language. Unlike C, Modula-2 does not freely provide these facilities; instead, their use is tied to specific language constructs. Modula-2 thus provides language-level support for the separation of machine-independent software from machine-specific software.

Like C, Modula-2 provides only primitive operations close to the level of the machine; routines for I/O, storage allocation, and process scheduling are programmed in Modula-2 and stored in a library. Unlike C, Modula-2 can enforce type checking of parameters to library routines. Modula-2 thus provides language-level support for ensuring error-free separate compilation.

In short, Modula-2 demonstrates that a highly structured, "protective" language need not sacrifice power, simplicity, or ease of use; it is therefore as much a successor to C as it is a successor to Pascal.

The module concept is of central importance in Modula-2. The purpose of a module is to contain a group of related procedures and data. A module allows some of its objects to be visible outside of the module, but hides the existence of other objects from the rest of the program. Modules allow large programs to be structured in a more readable fashion than is possible with block-structured languages: small collections of modules (sharing relatively few objects) replace the traditional army of procedures interconnected by pages of global declarations.

The ability of modules to separate a program into semi-independent parts provides the foundation for separate compilation. Modula-2 defines a special type of module which is compiled separately from a main program. Large programs can be constructed as collections of separately compiled modules; alternatively, separately compiled modules can be installed in a library for use by many programs. Standard utility modules are an integral part of every Modula-2 implementation, as they are used in almost every program.

The ability of modules to contain and hide objects allows Modula-2 to maintain machine independence in the face of low-level machine access. Machine-dependent items can be encapsulated in specific modules, and thus isolated to small portions of a program. These modules reveal only a high-level interface through which the machine-dependent items are accessed. When programs are transported to different systems, the bulk of the software remains unchanged; only the machine-specific modules need be rewritten.

Separately compiled modules can make a type identifier visible while hiding the structure of the associated type; this permits the definition of "abstract data types". (All operations on abstract data types are provided via procedure calls; a familiar example is Pascal's file type.)

NOTE- Chapter 2 presents concepts unique to Modula-2, and thus new to Pascal programmers. Read this chapter first, as it presents some key language concepts. Chapter 3 is light reading — it describes minor syntactic and semantic differences from Pascal. Because it covers finer points in the language, chapter 3 is organized for ease of reference; you will find yourself thumbing through it quite often while your programming habits shift from Pascal to Modula-2.

2 New Concepts

This chapter introduces language features unique to Modula-2. It presents enough information on these features for you to understand their rationale and write programs using them, but does not provide complete descriptions. Detailed information is provided in other parts of this manual and in the Modula-2 language report; references are provided at the appropriate points in this chapter.

New concepts include **modules, separate compilation, module libraries, standard utility modules, low-level machine access, coroutines & interrupts**, and **procedure variables.** As noted before, modules are the key concept in Modula-2. Separate compilation is accomplished with variants of modules; the standard library is a collection of commonly used "standard" modules; low-level machine access and process schedulers are provided by standard modules. Of the new concepts, only procedure variables are unrelated to modules; they follow from a new data type known as the "procedure type".

NOTE- This chapter uses small Modula-2 programs to illustrate the use of the new language features; in doing so, it reveals a number of the syntactical differences described in the next chapter. Fortunately, the new syntax is only slightly different, and quite easily understood; it will not hamper your comprehension of the programs.

2.1 Modules

Before explaining modules, it is worthwhile to review Pascal's concepts of **scope** and **block.**

A fundamental aspect of Pascal (and most other modern programming languages) is that it is a **block-structured** language. Block structure has proved useful as a method of program organization; it allows things to be declared locally to a procedure block so that they are unknown outside the block. Well-designed programs exploit block structure to improve their readability and understandability; when a variable or procedure is needed in only one place, it is declared in the local block so as not to impose on the rest of the program.

The range in which an object (e.g. a variable or procedure) is known is called the object's **scope.** Blocks can be nested, and an object's scope is the block in which it is declared; therefore, scopes can be nested. The general scope rule is as follows: the scope of an object extends from the block in which it is declared down through all nested blocks. Another way of looking at scope is the **visibility** rule: for a given block, any objects declared in nested blocks are invisible, but all objects declared in enclosing blocks are visible.

Block structure controls not only an object's scope, but also its **existence** at runtime. Objects local to a block exist only while the program executes statements inside the block; they are created when the block is entered, and destroyed when the block is exited. The existence rule implies that local variables cannot maintain their values across calls; the only way for them to do so is by declaring them in an outer block (where they become visible to the rest of the program). Thus, block structure binds a variable's existence to its visibility.

In the design of large programs, block structure proves inadequate for two reasons:

- There is a need to separate visibility from existence. It should be possible to declare variables that maintain their values, but are visible only in a few parts of a program.

- There is a need for closer control of visibility. A procedure should not be able to access every object declared outside of it when it only needs to access a few (if any) of them.

Modula-2 introduces the **module** structure to address these problems. Syntactically, modules closely resemble procedures, but they have different rules about visibility and the existence of their locally declared objects. Consider the following declarations:

```
PROCEDURE Outside;
  VAR x,y,z: INTEGER;

  MODULE Mod;
    IMPORT x;
    EXPORT a,P1;
    VAR a,b,c: INTEGER;

    PROCEDURE P1;
    BEGIN
      a := a + 1;
      x := a;
    END P1;

  END Mod;
...
END Outside;
```

```
PROCEDURE Outside;
  VAR x,y,z: INTEGER;

       (* no module here *)


       a,b,c: INTEGER;

  PROCEDURE P1;
  BEGIN
    a := a + 1;
    x := a;
  END P1;


...
END Outside;
```

The only syntactic differences between the module Mod and a normal procedure declaration are the reserved word beginning the declaration (MODULE instead of PROCEDURE) and the presence of IMPORT and EXPORT declarations following the module heading.

The semantic differences are more interesting. The objects declared within Mod (a, b, c, and P1) exist at the same level as the variables x, y, and z. In terms of the variables, this means that a, b, and c are created at the same time as x, y, and z, and exist as long as procedure Outside is active. The objects named in Mod's **import list** (the list of identifiers following the reserved word IMPORT) are the only externally declared objects visible within Mod; thus, Mod is able to access the variable x, but y and z are invisible. The objects named in Mod's **export list** (the list of identifiers following the reserved word EXPORT) are the only locally declared objects visible outside Mod; thus, a and P1 are accessible from Outside, but b and c remain hidden inside Mod.

Note that from Outside's point of view, a and P1 appear to be regular locally declared objects; they have the same visibility and existence as x, y, and z. Note also that b and c lead a similar, but merely hidden, existence. A reasonable conclusion to reach from these observations is that (unlike procedures) modules themselves do not really exist! This is more or less true — modules affect visibility (a compile-time phenomenon), but not existence (a run-time phenomenon). A module can be thought of as a syntactically opaque wall protecting its enclosed objects. The export list names identifiers defined inside the module that are also to be visible outside. The import list names the identifiers defined outside the module that are visible inside.

Here is a summary of the rules for visibility and existence in modules:

- Locally declared objects exist as long as the enclosing procedure remains activated.

- Locally declared objects are visible inside the module; if they appear in the module's export list, they are also visible outside. Objects declared outside of the module are visible inside only if they appear in the module's import list.

So far, all that has been presented are the mechanics of the module structure. How are modules to be used? The following examples demonstrate the essence of modularity:

```
MODULE MainProgram;

  ...

  MODULE RandomNumbers;
    IMPORT TimeOfDay;
    EXPORT Random;
    CONST Modulus   = 2345;
          Increment = 7227;
    VAR Seed: INTEGER;

    PROCEDURE Random(): INTEGER;
    BEGIN
      Seed := (Seed + Increment)
              MOD Modulus;
      RETURN Seed;
    END Random;

  BEGIN
    Seed := TimeOfDay;
  END RandomNumbers;

  ...

BEGIN (* MainProgram *)
  ...

  WriteInt(Random(), 7);
  ...
END MainProgram.
```

```
MODULE MainProgram;

  VAR Seed: INTEGER;


  ...


  PROCEDURE Random(): INTEGER;
    CONST Modulus   = 2345;
          Increment = 7227;
  BEGIN
    Seed := (Seed + Increment)
            MOD Modulus;
    RETURN Seed;
  END Random;


  ...


BEGIN (* MainProgram *)
  Seed := TimeOfDay;
  ...
  WriteInt(Random(), 7);
  ...
END MainProgram.
```

The random number generator in these examples uses a seed variable to generate the next random number; the seed must maintain its value across function calls. The program on the right shows the classical block-structured solution. Note how Seed's declaration floats to the top of the program (to avoid the existence rule), forcing its initialization to sink to the bottom. Two obvious disadvantages arise from the scattering of Seed across the face of the program: its occurrences become hard to find (imagine that this program is 10,000 lines long!), and it becomes accessible to every other procedure in the program (when it should be safely buried in Random).

The example on the left demonstrates the usefulness of the module structure. Everything having to do with the random number generator is contained in one place; only the procedure Random is visible. Because the module is declared at the outermost level, Seed is initialized only once, and exists for the life of the program.

The random number module introduces another feature of modules; unlike the module in the first example, this module contains both declarations and a statement part. **Module bodies** are the (optional) outermost statement parts of module declarations; they serve to initialize a module's variables. As it was mentioned before that modules are purely syntactic entities, the presence of executable statements might seem questionable. The consistency of this presentation is preserved by the fact that a module's body is analogous to a module's variables; though subjected to the module's restrictive visibility rules, module bodies conceptually belong to the enclosing procedure rather than the modules themselves.

Module bodies are automatically executed when the enclosing procedure is called. (Recall that a module's variables come into existence at the same time.) If a procedure contains several modules, the module bodies are executed in the order in which they occur within the procedure (see the following example). A procedure's statement part executes only after its module bodies have been executed. Just as module variables should be considered to exist at the same level as the enclosing procedure's variables, module bodies should be considered as prefixes to the enclosing procedure's body.

NOTE- Though module bodies are treated here as implicitly included statements, they are implemented as procedures which are automatically called at the start of the enclosing procedure's body.

Example of module body execution:

```
PROCEDURE Enclosing;

  MODULE M1;
    EXPORT x, y;

    VAR x: INTEGER;

    MODULE M2;
      EXPORT y;
      VAR y: INTEGER;
    BEGIN
      y := 0;
    END M2;

  BEGIN
    x := -1;
  END M1;

  MODULE M3;
    EXPORT z;
    VAR z: INTEGER;
  BEGIN
    z := 1;
  END M3;

  VAR coordinate: INTEGER;

BEGIN (* Enclosing *)
  (* M2's body automatically called here *)
  (* M1's body automatically called here *)
  (* M3's body automatically called here *)
  ...
  WriteInt(coordinate, 7);
END Enclosing;
```

The next example illustrates (in a high-level fashion) the organizational differences between large programs written in Pascal and in Modula-2. The program in question is a one-pass compiler (say, the Pascal P-compiler). It would typically consist of about 5000 lines of source text.

One-pass compilers are notable for having a number of things going on at once; principal activities include scanning (reading the source file), parsing (checking the syntax), and code generation (producing a code file). Each of these activities is reflected in the compiler by collections of constants, types, variables, and procedures comprising the scanner, parser, and code generator. The compiler also contains general purpose variables and procedures used in all parts of the compiler.

The following examples demonstrate the effects of block structure upon the organization of such a compiler:

```
MODULE Compiler;                  program Compiler;

  <general consts, types,           <consts for generals, scanner,
   vars, and procedures >            parser, and code generator  >

  <scanner module>                  <types for generals, scanner, etc.>

  <parser module >                  <vars for generals, scanner, etc. >

  <code generator module>           <procs for generals, scanner, etc.>

<main program>                    <main program>

END Compiler.                     end. (* Compiler *)
```

The compiler written in Pascal is a jumble of declarations; the order reflects the syntactical structure of Pascal rather than the logical structure of the compiler. The compiler written in Modula-2 is organized logically; each module can be expected to import some globally declared objects and export some (but certainly not all) of its own objects. (Note that this results in fewer global declarations.) The Modula-2 program is more readable, more understandable, and less prone to erroneous side effects.

NOTE- The next section shows how separate compilation of modules can also make the Modula-2 program much easier to maintain than its Pascal counterpart.

The rest of this section is devoted to additional information on modules.

Like procedures, modules can be declared at any level; the visibility and existence rules hold for nested module declarations.

Import and export lists immediately follow the module heading. Both lists are optional: a module can have an import list but lack an export list, or vice-versa (see the following example). Modules can contain several import lists (i.e. several occurrences of the reserved word IMPORT followed by a list of identifiers), but only one export list. Import lists must precede the export list.

Examples of import and export lists:

```
MODULE abc;                       MODULE trader;
  IMPORT i,j,k;                     EXPORT commodities;
  IMPORT x,y,z;                     ...
  EXPORT AlphabetSoup;            END trader;
  ...
END abc;
```

Any kind of object can be imported or exported by naming its identifier in an import or export list. Exporting a record type makes its fields visible. Exporting an enumeration type makes its enumeration constants visible. Exporting a module makes all of its exported identifiers visible. Procedures retain the structure of their parameter list, but do not transport parameter type identifiers; thus, parameter types must be exported separately. These rules apply to both imports and exports.

NOTE- Imported identifiers must be unique with respect to each other and to locally declared/exported identifiers, as identifier clashes are analogous to declaring an identifier twice (a syntax error). Things get interesting when imported identifiers are records or enumerations. Record field identifiers are local to their record type, and thus cannot clash with other imported identifiers. Importing an enumeration, however, may cause one of the (implicitly imported) enumerated constant identifiers to clash with some other imported identifier.

Example of exporting various types of objects:

```
MODULE stuff;
  EXPORT
    Rec,     (* field names R1, R2, and Ch are visible in Rec  *)
    things,  (* constants Some, No, and Any become visible     *)
    DoIt,    (* calls must match DoIt's procedure heading      *)
    Bird;    (* Eggs and Twigs become visible                  *)

  PROCEDURE DoIt (RSKfactor: INTEGER);
  BEGIN ... END DoIt;

  TYPE Rec = RECORD
               R1, R2: REAL;
               Ch: CHAR;
             END;

  TYPE things = (Some, No, Any);

  MODULE Bird;
    EXPORT Eggs, Twigs;
    ...
  END Bird;
  ...

END stuff;
```

NOTE- Modula-2's standard identifiers are automatically imported into every module. Thus, attempts to redefine them within a module will cause a syntax error ("identifier declared twice"). Standard identifiers can be redefined within procedures, however.

Identifiers obtained by importing or exporting are used like normally declared identifiers; that is, as if they did not originate from a module. However, they can also be referenced as **qualified identifiers.** An identifier is qualified by preceding it with the name of its module; the syntax is identical to record field access in Pascal. For example, an identifier named "Ident" imported from the (visible) module "Mod" can be referenced either as "Ident" or "Mod.Ident" (see the following example).

NOTE- Qualified identifiers may not appear in import or export lists.

References to exported identifiers can be qualified or unqualified; however, if the symbol EXPORT is followed by the symbol QUALIFIED, identifier references outside of the module must be qualified. This is known as **qualified export.** Qualified export allows a module to avoid identifier clashes caused by other modules exporting the same identifier; it should be used when the names declared outside a module are unknown (e.g. a standard library module imported by many different programs).

Example of qualification:

```
MODULE Latoo;

  MODULE M1;
    EXPORT OverLoad, A;                      (* unqualified export *)
    CONST A = 'a';
    VAR OverLoad: INTEGER;
  END M1;

  MODULE M2;
    EXPORT QUALIFIED OverLoad, Canada;    (* qualified export *)
    VAR OverLoad: INTEGER;      (* ...averts name clash with M1 *)

    PROCEDURE Canada;
    BEGIN
      HALT;
    END Canada;

  END M2;

  VAR i: INTEGER;
      ch: CHAR;
BEGIN
  ch := A;             (* unqualified reference *)
  ch := M1.A;          (* optionally qualified reference     *)
  i  := OverLoad;      (* unqualified refers to M1's var     *)
  i  := M1.OverLoad;   (* optionally qualified reference     *)
  i  := M2.OverLoad;   (* M2 qualification averts clash      *)
  M2.Canada;           (* M2's objects must be qualified     *)
END Latoo;
```

Preceding an import list with the symbol FROM followed by a module identifier has the effect of unqualifying identifiers exported by the named module. This is known as **unqualifying import.** Unqualifying import lists can only contain identifiers exported by the named module. (This is why multiple import lists are allowed in module declarations.) Unqualifying import lists are useful for limiting the scope of unqualified identifiers to small portions of a program.

Example of unqualification:

```
MODULE A;

  MODULE M1;
    EXPORT v1,v2;              (* unqualified export *)
    VAR v1,v2: INTEGER;
    ...
  END M1;

  MODULE M2;
    EXPORT QUALIFIED z1,z2; (* qualified export *)
    VAR z1,z2: INTEGER;
    ...
  END M2;

  MODULE M3;
    IMPORT M1;
    EXPORT QUALIFIED t1,t2;    (* qualified export *)
    VAR t1,t2: INTEGER;
  BEGIN
    t1 := v1;
    t2 := v2;
  END M3;

  MODULE HOST;
    FROM M1 IMPORT v1,v2;      (* FROM is optional here, but...     *)
    FROM M2 IMPORT z1;         (* required from qualified export    *)
    IMPORT M3;                 (* qualified import of t1 & t2       *)
  BEGIN
    z1 := v1 + v2;             (* qualification unnecessary here    *)
    v1 := M3.t1 + M3.t2;       (* qualification required here       *)
  END HOST;

END A;
```

NOTE- The following is an extreme (and thus illustrative) example of the potential interactions between nested modules and qualified identifiers:

```
MODULE Nesting;
  FROM InOut IMPORT Write;

  MODULE A;
    EXPORT B;
                         (* x or B.x or C.x or B.C.x visible here *)
    MODULE B;
      EXPORT C;
                         (* x or C.x visible here *)
      MODULE C;
        EXPORT x;        (* x visible here *)
        CONST x = '!';
      END C;

    END B;

  END A;

BEGIN
  Write(x);              (* these all refer to x *)
  Write(A.x);
  Write(B.x);
  Write(C.x);
  Write(A.B.x);
  Write(A.C.x);
  Write(B.C.x);
  Write(A.B.C.x);
END Nesting;
```

NOTE- See chapter 11 in the Modula-2 language report for more information on modules.

2.2 Separately Compiled Modules

The basic textual unit accepted by the compiler is called a **compilation unit.** Modula-2 programs are constructed from two kinds of compilation units: **program modules** and **library modules.**

Program modules are single compilation units; their compiled forms constitute executable programs. Because they are the outermost modules of a program, program modules can have import lists, but no export list. A program module's import lists name objects defined in the library; specifically, in separately compiled library modules. The library is an integral part of Modula-2, for it provides the system-level environment from which objects (such as operating system routines) are imported into a program. (See **The Modula-2 System** for more information on the library.)

Examples of program modules:

```
MODULE Foon;
  FROM InOut IMPORT WriteString;
  (* WriteString obtained from library module InOut *)
BEGIN
  WriteString('hi!');
END Foon.                (* period marks this as a program module *)


MODULE Yeen;
  IMPORT InOut;          (* module InOut obtained from library *)
BEGIN
  InOut.WriteString('hi!');
END Yeen.
```

A compilation unit can import entire library modules or individual objects from library modules. A library module is imported by naming it in an import list; all of its exported objects become available, but they must be referenced as qualified identifiers (e.g. "Yeen" in the example above). Individual objects are obtained from a library module by unqualifying import; they are then referenced as regular identifiers (e.g. "Foon" above).

Library modules are divided into two compilation units: **definition modules** and **implementation modules.** Definition modules contain declarations of the objects which a library module exports to other compilation units. Implementation modules contain the code implementing the library module. Definition and implementation modules always exist in pairs; they are related by being declared with the same module identifier.

Definition modules are similar to program modules, but are prefixed with the symbol DEFINITION. A definition module contains constant, type, and variable declarations, and procedure headings. It does not contain module declarations, procedure bodies, or a module body. The import lists name objects imported from the library into the definition module. The export list specifies objects declared in the definition module which can be imported by other compilation units.

NOTE- Only qualified export may be used in definition modules.

Example of a definition module:

```
DEFINITION MODULE StringIO;
  FROM StringOps IMPORT String;          (* obtained from library *)
  EXPORT QUALIFIED ReadStr, WriteStr;    (* visible from StringIO *)

  PROCEDURE WriteStr(S: String);         (* like a forward declaration *)

  PROCEDURE ReadStr(VAR S: String);

END StringIO.
```

Implementation modules have the same syntax as program modules, but are prefixed with the symbol IMPLEMENTATION. Like program modules, implementation modules may not contain an export list. The import lists name objects imported from the library into the implementation module.

All objects declared in a definition module are automatically available in the corresponding implementation module (implying that definition modules must be compiled before implementation modules). Objects imported into a definition module are not made available in the implementation module; if needed, they must be imported again.

NOTE- The implementation module must contain complete declarations of procedures declared in the definition module; unlike a forward declared procedure in Pascal, a secondary declaration must include its parameter list (which must be identical to the one in the definition module declaration).

Example of an implementation module:

```
IMPLEMENTATION MODULE StringIO;
  FROM CharIO IMPORT ReadCh, WriteCh;    (* obtained from library *)
  FROM StringOps IMPORT String, Length, MaxString;
  FROM ASCII IMPORT nul;

  PROCEDURE WriteStr(S: String);         (* note repeated param list *)
    VAR I: CARDINAL;
  BEGIN
    FOR I := 0 TO Length(S)-1 DO
      WriteCh(S[I]);
  END WriteStr;

  PROCEDURE ReadStr(VAR S: String);
    VAR I: CARDINAL;
        ch: CHAR;
  BEGIN
    I := 0;
    REPEAT ReadCh(ch);
      S[I] := ch;
      INC(I);
    UNTIL (ch = nul) OR (I > MaxString);
  END ReadStr;

END StringIO.
```

Within a given program, a library module may be imported by more than one compilation unit; for example, a module A imports modules B and C, each of which import D. When this situation arises, Modula-2 defines that only one instance of a library module can exist at a time. In the example, modules B and C thus share D's exported objects (in particular, D's variables).

Implementation modules may contain module bodies. The system arranges the execution order of library module bodies so that imported library modules are initialized before the importing modules are. If module A imports B, which imports C, which imports D, then the initialization order is D, C, B, and finally A. This ensures that a module's initialization code can rely on variables imported from other library modules. (If imported library modules are mutually independent, their execution order is undefined.)

NOTE- Library modules are often imported in more than one place. A simple case is when a program imports a library module by name and also imports identifiers from the same module using unqualifying import. A more subtle case is when a program imports two library modules which both import a third module. In all such cases, a library module is initialized only once.

Why are library modules divided into separate definition and implementation modules?

Consider the design and development of a large software system, possibly by a group of programmers. The first step in designing such a system is to identify major subsystems and design interfaces through which the subsystems communicate. After this step is completed, development of the subsystems can proceed, with each programmer responsible for developing one (or more) of the subsystems.

Now consider the project requirements in terms of Modula-2's separate compilation facilities. Subsystems will most likely be composed of one or more compilation units. Defining and maintaining consistent interfaces is of critical importance in ensuring error-free communication between subsystems (especially when they are developed by different people). During the design stage, however, the subsystems themselves do not yet exist; they are known only by their interfaces.

The concept of a subsystem interface corresponds to the definition module construct; thus, interfaces can be defined as a set of definition modules before subsystem development (i.e. design and coding of the implementation modules) begins. These modules are distributed to all members of the programming group; throughout the project, they define the interfaces which all subsystems (and thus all programmers) must adhere to. Interface consistency is automatically enforced by the compiler.

Another advantage provided by separate definition and implementation modules is the ability of two library modules to import objects from each other. This would be impossible if library modules were single compilation units, as each would require previous compilation of the other in order to compile successfully. With separate definition modules, the modules can be imported in the implementation modules, allowing the definition modules to be compiled beforehand (independently of the mutual importation).

NOTE - Mutually importing library modules dictate arbitrary module initialization order. (Which is more nested?) In such cases, the modules' initialization bodies cannot depend on objects imported from the other module.

This section concludes with a description of Modula-2's facilities for defining data types whose only operations are provided by procedure calls.

Library modules can export two kinds of types: **transparent types**, and **opaque types.**

Normal type declarations are (by default) transparent types. The type identifier of a transparent type is associated with a structure which implicitly defines certain operations on objects of that type; in the case of structured types, the internal components are accessible. For example, array types imply a known base type and define the subscript operation to access individual array elements.

Opaque types are types whose internal structure is known only in the implementation module. Modules importing an opaque type can declare and assign objects of that type, but cannot perform any other operations (save those provided by procedures exported along with the opaque type). In particular, an opaque type's internal components are inaccessible.

Opaque types are declared in a definition module as identifiers lacking a type definition; like exported procedures, the complete declarations of opaque types are contained in the implementation module.

NOTE- Modula-2 limits opaque types to pointers and subranges of standard types. The most common opaque type is a pointer to a record (whose details remain hidden).

Example of opaque types:

```
DEFINITION MODULE Files;
  EXPORT QUALIFIED File, Open, Close, Read, Write;

  TYPE File;   (* note lack of type definition *)

  PROCEDURE Open(VAR f: File; name: ARRAY OF CHAR);

  PROCEDURE Read(f: File; VAR ch: CHAR);
  ...
END Files.



IMPLEMENTATION MODULE Files;
  FROM Storage IMPORT ALLOCATE;

  TYPE File = POINTER TO      (* complete decl. *)
                RECORD
                  DiskUnit: CARDINAL;
                  BlockNumber: CARDINAL;
                  NextByte: BuffIndex;
                END;

  PROCEDURE Open(VAR f: File; name: ARRAY OF CHAR);
  BEGIN ... END Open;

  PROCEDURE Read(f: File; VAR ch: CHAR);
  BEGIN ... END Read;
  ...

BEGIN ...
END Files.


MODULE UseFiles;
  FROM Files IMPORT File, Open, Write, Close;
  VAR f1, f2: File;

BEGIN
  Open(f1, "new.data");
  ...
END UseFiles.
```

A classic use of opaque types is the definition of files. The type File and its operations (Open, Read, and Write) can be expressed as a library module. Another facility well suited to opaque types is the semaphore and its operations (signal and wait) for process synchronization.

2.3 The Module Library

The module library is a collection of separately compiled modules that forms an essential part of every Modula-2 implementation. It typically contains the following kinds of modules:

- Low-level system modules which provide access to local system resources.
- Standard utility modules which provide a consistent system environment across all Modula-2 implementations.
- General-purpose modules which provide useful operations to many programs.
- Special-purpose modules which form part of a single program.

The library is stored in one or more disk files containing compiled forms of the library modules's compilation units. The compiled form of a definition module is called a **symbol file.** The compiled form of an implementation module is called an **object file.**

The library is accessed by both the compiler and the program loader. The compiler reads symbol files from the library when compiling programs that import library modules. The loader loads object files from the library when executing programs that import library modules.

Modules are compiled separately, but not independently. The division of programs into separately compiled modules forms dependence relations between library modules and their **clients** (i.e. the modules that import them). These dependencies affect the ability to recompile a module independently of the rest of the system.

The simplest example of such a dependence relation arises in the compilation of a single library module. The compiler must reference the module's symbol file in order to compile the implementation module; therefore, the definition module must be compiled first. Once an implementation module has been compiled, its object file is tied to the current symbol file, as the object code is based on procedure and data offsets obtained from the symbol file.

Similarly, client modules are tied to symbol files; programs which import a library module have to assume that the symbol file offsets are accurate reflections of the corresponding object file.

What happens if a definition module is changed without recompiling its implementation module? The procedure and variable offsets in the updated symbol file may no longer match the object code, yet subsequently compiled programs that import the module are assigned offsets defined in the new symbol file. If the implementation module is not recompiled (thus bringing the object file up to date with the new symbol file), the new programs may crash when they attempt to reference the library module.

All such problems can be avoided by following these rules:

- A definition module must be compiled prior to its client modules.
- An implementation module may be recompiled without recompiling any other modules in the system.
- When a module's definition and implementation are recompiled, all client modules are invalidated, and must be recompiled.

The Modula-2 system contains facilities for automatic enforcement of the last rule. The compiler assigns a unique value to the symbol file of every definition module it compiles; these values are called **module keys.** When a compilation unit imports a module, the compiler records the module key in the code file. When a program is executed, the loader checks that the module keys stored in the program match the keys in the imported library modules; if a mismatch is found, the loader issues an error message and aborts the program. Thus, the system prevents programs crashes caused by inconsistent module interfaces.

Module key checking is the system-level analogue to type checking within the compiler; they are of equal importance in Modula-2.

NOTE- Recompiling only a definition module does not prevent client programs from executing with the non-updated object file. Recompiling the matching implementation module produces an object file with a new module key — after which the client programs must be recompiled.

2.4 Standard Utility Modules

The Modula-2 language contains no standard procedures for I/O, memory allocation, or process scheduling; instead, these facilities are provided by **standard utility modules** stored in the library. Standard utility modules are expected to be available in every Modula-2 implementation; thus, by using only standard modules, Modula-2 programs become portable across all implementations. (See **Standard Library** for more information on standard utility modules.)

The advantages of expressing commonly-used routines as library modules (rather than part of the language) include a smaller compiler, smaller run-time system, and the ability to define alternative facilities when the standard facilities prove insufficient. Disadvantages include the need to explicitly import and bind library modules, and — occasionally — a less flexible syntax imposed by expressing standard routines as library modules (as opposed to their being handled specially by the compiler).

NOTE- Modula-2's ability to express general purpose routines is greatly enhanced by its facilities for relaxing type checking.

The rest of this section is devoted to comparing Pascal's standard procedures with the equivalent procedures provided by Modula-2's standard utility modules.

Pascal's standard procedures Read and Write are replaced by read and write routines obtained from the standard module **InOut.** Read and Write's parameter list sequences and overloaded parameter types are not expressible in Modula-2; instead, procedures are provided for handling single arguments of each data type. Thus, what appears in Pascal as:

```
writeln('Name = ',ID,' Value = ',Val:3);
```

... becomes in Modula-2:

```
WriteString('Name = ');
WriteString(ID);
WriteString(' Value = ');
WriteInt(Val, 3);
WriteLn;
```

NOTE- Though the Modula-2 version appears less efficient, the actual code is no larger than the Pascal version; the difference merely reflects a shift in the programming burden from the compiler to the programmer. Modula-2 does not share Pascal's ability to automatically translate Read and Write statements with multiple arguments into the requisite number of system calls.

Modula-2 provides Pascal's text files with the standard module **Texts.** File handling is performed by the standard module **Files,** which provides random access in addition to Pascal's sequential data access.

Because the compiler must perform special parsing for record variants (e.g. "NEW(Citizen, FALSE, widowed)"), NEW and DISPOSE remain as standard procedures in Modula-2; however, the compiler translates NEW and DISPOSE into equivalent calls to ALLOCATE and DEALLOCATE, which are procedures provided either by the standard module **Storage** or by special purpose procedures. This allows alternate storage implementations to take advantage of the compiler's ability to minimize storage allocated for record variants.

Example of NEW and DISPOSE:

```
MODULE Memory;
  (* obtain storage management facilities from the library *)
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  VAR p: POINTER TO INTEGER;

BEGIN
  (* ALLOCATE and DEALLOCATE must be visible here *)
  NEW(p);
  DISPOSE(p);
END Memory.
```

Mathematical functions (e.g. sin and cos) are provided by the standard module **MathLib0.** Modula-2 also includes standard modules for process scheduling, console I/O, and calling programs as procedures.

2.5 Low-level Machine Access

Modula-2 provides the following facilities for programming low-level, machine-specific operations:

- Type transfer functions allow programs to circumvent normal type compatibility rules.
- Variables may be declared to reside at fixed memory addresses.
- The module SYSTEM provides data types for manipulating machine-level data objects, and procedures for determining the memory address of variables and the machine-level representation of variables and types.

The use of these facilities should be confined to a few specific modules. The practice of concealing low-level operations in modules results in safer programming by preventing inadvertent access to machine-level objects; it also improves the potential for program portability (as only the low-level modules need be rewritten). In general, low-level modules are marked by the presence of the module identifier SYSTEM in their import list. Note, however, that the facilities for type transfer and fixed-address variables are generally available; thus, their use should be marked by discretion.

Type identifiers can be used as **type transfer functions.** Type transfer functions are restricted to conversion between types whose machine-level representations occupy the same number of words. The type identifier is used as a function identifier, and the variable to be converted is passed as the function argument. The function result is compatible with the type specified by the type identifier. Note that type transfer functions do not involve any actual computation — they merely relax compile-time type checking.

NOTE- Modula-2's standard procedures ORD, ODD, CHR, and VAL provide more respectable forms of type transfer. See 3.6.4 for details.

Example of type transfer functions:

```
MODULE LowLevel;

  TYPE Arr = ARRAY [1..3] OF INTEGER;
       Rec = RECORD                    (* occupies 3 words *)
               X: CHAR;
               Y,Z: INTEGER;
             END;
  VAR b: BOOLEAN;
      ch: CHAR;
      l: Arr;
      j: Rec;

BEGIN
  b := BOOLEAN(ch);
  l := Arr(j);
END LowLevel.
```

Variables can be declared to reside at fixed memory addresses. The address is specified as a cardinal constant (enclosed by square brackets) following the variable identifier. The variable itself may be of any type.

NOTE- Character variables declared at fixed addresses are usually accessed as byte quantities. See the **Implementation Guide** for details.

Example of a variable declared at a fixed address:

```
PROCEDURE Stuff;
  TYPE FlagBits = BITSET;
  VAR  Flaggy[400H]: FlagBits;
     (* 1 word set resides at byte address 400 hex *)
BEGIN
  ...
END Stuff;
```

All Modula-2 implementations include a module named SYSTEM; not surprisingly, it is called the **system module.** The system module provides the data types WORD and ADDRESS for manipulating machine-level data objects, the procedure ADR for obtaining the memory addresses of variables, and the procedures SIZE and TSIZE for determining the machine-level representations of variables and types.

NOTE- Because its exported objects have special properties, the system module is contained entirely in the compiler. The system module is called a **pseudo-module** because it is not part of the library.

The type WORD is used in general purpose routines which must operate on arguments of any type. Formal parameters of type WORD are type compatible with any actual parameter occupying one word of storage. Outside of parameter lists, however, the only operation allowed on type WORD is assignment; furthermore, WORD is incompatible with all other types. These limitations are overcome by using type transfer functions to perform the necessary operations.

"Open" array parameters of base type WORD are type compatible with all variables; in particular, records and sets. Such parameters allow any variable to be interpreted as a sequence of words. See 3.6.3 for more information.

Example of type WORD:

```
PROCEDURE OnesComplement(VAR arg: WORD);
(* uses 1-word set type to XOR with all 1's *)
BEGIN
  arg := WORD(BITSET(arg) / {0..15});
END OnesComplement;
```

The type ADDRESS is compatible with all pointers and also with the type CARDINAL (unsigned integer); thus, arithmetic operations can be performed on operands of type ADDRESS. ADDRESS allows programs to perform straightforward pointer and address arithmetic.

The formal definition of ADDRESS is:

```
TYPE ADDRESS = POINTER TO WORD;
```

Example of type ADDRESS:

```
PROCEDURE DumpMemory(memptr: ADDRESS; words: CARDINAL);
(* display contents of memory or dynamic variable *)
  VAR inx: CARDINAL;
BEGIN
  FOR inx := 1 TO words DO
    WriteHex(CARDINAL(memptr^), 6);
    INC(memptr, TSIZE(WORD)); (* next word in memory *)
  END;
END DumpMemory;
```

The function ADR(x) returns the memory address of the variable x; the result type is ADDRESS.

SIZE(x) returns the number of storage units assigned to the variable x. x can be a selected variable (e.g. "a[i].x"). SIZE returns the maximum possible size of records containing variants. SIZE does not accept open array parameters as arguments.

TSIZE(T) returns the number of storage units assigned to a variable of type T. Note that TSIZE recognizes the variant tag lists accepted by NEW and DISPOSE; thus, it can return the actual sizes of dynamically allocated records. SIZE and TSIZE return values of type CARDINAL. Storage units on most systems are bytes — see the **Implementation Guide** for details.

Example of ADR, SIZE, and TSIZE:

```
PROCEDURE Diddle;
  TYPE BIG = ARRAY [1..5] OF INTEGER;
       Rec = RECORD
               F1: CHAR;
               CASE B: BOOLEAN OF
                 TRUE : big: BIG |
                 FALSE: little: INTEGER;
               END;
             END;
  VAR a: ADDRESS;
      z: CARDINAL;
BEGIN
  a := ADR(z);
  z := SIZE(a);             (*  2 bytes *)
  z := TSIZE(BIG);          (* 10 bytes *)
  z := TSIZE(Rec);          (* 14 bytes *)
  z := TSIZE(Rec, TRUE);    (* 14 bytes *)
  z := TSIZE(Rec, FALSE);   (*  6 bytes *)
END Diddle;
```

NOTE- This section has described machine-dependent facilities for a typical 16-bit processor. Implementations on different processors may provide different types and compatibility rules.

2.6 Coroutines and Interrupts

Many modern systems programming languages (such as MODULA, Concurrent Pascal and Ada) define facilities for concurrent processes and process scheduling. Implementing such languages on single-processor computers requires an underlying "run-time system" to schedule processes for execution and to simulate concurrent execution by switching the processor between processes. Modula-2 was designed to write, rather than require, run-time systems; hence, it foregoes concurrent processes in favor of the simpler **coroutine** concept (1).

In Modula-2, coroutines provide a foundation for programming the more common forms of concurrency; thus, process schedulers are written in Modula-2 and stored in the library instead of being written in assembly language as part of a run-time system. Modula-2's approach has two advantages. First, there is no run-time system occupying memory; a process scheduler is loaded only if a program imports it. Second, Modula-2 is not limited to a single process scheduling algorithm; when a different scheduling algorithm is required, it can be programmed as a library module.

Coroutines are procedures which execute independently (but not concurrently). A Modula-2 program itself executes as a coroutine; however, this is irrelevant unless the program creates its own coroutines. Coroutines must be created before they can be called; a coroutine is created by specifying a procedure for the coroutine to execute and an area of memory for the coroutine to execute in. Once created, a coroutine becomes executable, but does not actually begin to execute; it remains inactive until it is called by another coroutine.

Coroutines spend their time alternating between two states: inactive, and executing. In a group of coroutines, only one coroutine executes at a time; the rest are inactive. Coroutines schedule their execution by calling each other; in a coroutine call, the calling coroutine becomes inactive and the called coroutine resumes execution.

NOTE- Coroutine calls are conceptually different from procedure calls. Coroutine calls are not recursive; unlike procedure calls, a coroutine call does not imply a subsequent return. Coroutine calls are best thought of as a direct transfer of control between two coroutines.

Generally, coroutines do not reach the end of their procedure, but continue to execute (between inactive periods) for the life of the surrounding program. To ensure this behavior, procedures executed by coroutines usually take the

(1) Knuth's **The Art of Computer Programming, Vol. 1** contains an excellent description of assembly language coroutines.

form of an unconditional loop containing one or more coroutine calls.

NOTE- A program is terminated if any coroutine reaches the end of its procedure body. When a program terminates, all of its coroutines are automatically terminated.

Example of a coroutine procedure:

```
PROCEDURE WriteHo;
BEGIN
  LOOP Write('H'); Write('o');
    INC(i);
    IF i > maxHiHo THEN
      WriteLn; i := 0;
    END;
    TRANSFER(Ho,Hi);
  END;
END WriteHo;
```

NOTE- The terms **coroutine** and **process** are synonymous in this section. **Process** is the preferred term in the context of Modula-2. **Coroutine** is the preferred out-of-context term, as it is technically more specific and cannot be confused with the usual concept of concurrent processes.

Coroutine facilities are obtained from the module SYSTEM, which exports the following identifiers: PROCESS, NEWPROCESS, TRANSFER, IOTRANSFER, and LISTEN.

All references to processes (e.g. coroutine calls) are made through process variables. Process variables are declared with type PROCESS; a process variable must be declared for each created process in order to distinguish the processes. A process variable can be thought of as a "pointer" to the actual process.

New processes are created with the NEWPROCESS procedure. NEWPROCESS has the following syntax:

```
PROCEDURE NEWPROCESS( P: PROC; A: ADDRESS;
                      N: CARDINAL; VAR P1: PROCESS);
```

P is the procedure that the new process will execute. P must be a parameterless procedure declared at the global (outermost) level in a compilation unit. (PROC is a standard type denoting a parameterless procedure.)

A and N specify the address and size of the area in which the process will execute. This area is usually declared as an array variable; the SYSTEM-supplied functions ADR and SIZE are used to obtain the array's size and memory address.

P1 is a process variable which is assigned the new process.

Example of using an array as a process space:

```
VAR Ho: PROCESS;
    B: ARRAY [1..200] OF WORD;
    ...

    NEWPROCESS(WriteHo, ADR(B), SIZE(B), Ho);
```

The area in which a process executes can be thought of as a miniature version of the system stack. A few words in each process space are used for storing an inactive process's execution state; the remaining space is available for the stack, which is used to store procedure call information and local variables belonging to the process procedure (and any other procedures called by it).

NOTE- Dynamic storage allocation is performed independently of multiprocessing; in particular, dynamic variables allocated by a process are allocated in the system storage area rather than in the process' own stack space.

WARNING- A process space operates as a miniature version of the system space; in particular, if a process stack fills up, a stack overflow occurs. Process stack overflows are generally handled poorly; the system is more likely to crash without warning than print a stack overflow message. Be prepared!

Because they execute in relatively small spaces, processes are more susceptible to stack overflow than main programs. It is a good idea to determine beforehand the amount of storage used by a process. On some systems, a process procedure consisting of nothing but a TRANSFER call successfully executes in as little as 30 words of space; however, this is an atypical case. The size of a process space is determined by adding together the process procedure's data size with the data sizes of procedures invoked in (possibly nested) call sequences made from the process. Be sure to also account for procedure call overhead; this amounts to about 10 words per procedure call.

In conclusion, be sure to allocate liberal amounts of storage to processes. Unless carefully calculated, process spaces should generally contain at least 100 words of storage.

NOTE- If a process invokes a system library module (even indirectly), determining its memory requirements may prove to be a difficult job. This problem is best solved by running processes in arbitrarily large process spaces (i.e. > 1000 words). Processes are generally intended for simple, low-level tasks with minimal resource requirements.

Coroutine calls are performed with the TRANSFER procedure. TRANSFER has the following syntax:

```
PROCEDURE TRANSFER(VAR OLD, NEW: PROCESS);
```

TRANSFER suspends the current process, assigns its execution state into the process variable OLD, and then resumes execution of the process identified by the variable NEW. Note that OLD should be passed the official process variable for the current process, so that it too may be subsequently resumed. Note also that NEW must already have been assigned a process.

WARNING- The system crashes if control is transferred to an uninitialized process variable.

NOTE- OLD is assigned the saved execution state after the process identified by NEW has been established as the currently executing process; thus, OLD and NEW can safely be assigned the same actual parameter. This implies that a single process variable P can be shared by two processes; each calls the other by calling TRANSFER(P,P).

Example of coroutines:

```
MODULE HiHo;

  FROM SYSTEM IMPORT
    WORD, ADR, SIZE, PROCESS, NEWPROCESS, TRANSFER;

  FROM Terminal IMPORT WriteLn, Write;

  CONST maxHiHo = 17;

  VAR i: CARDINAL;
      Hi, Ho, Main: PROCESS;
      A, B: ARRAY [1..200] OF WORD;

  PROCEDURE WriteHi;
  BEGIN
    LOOP Write('H'); Write('i');
      TRANSFER(Hi,Ho);
    END;
  END WriteHi;

  PROCEDURE WriteHo;
  BEGIN
    LOOP Write('H'); Write('o');
      INC(i);
      IF i > maxHiHo THEN
        WriteLn; i := 0;
      END;
      TRANSFER(Ho,Hi);
    END;
  END WriteHo;

BEGIN  i := 0;
  NEWPROCESS(WriteHi, ADR(A), SIZE(A), Hi);
  NEWPROCESS(WriteHo, ADR(B), SIZE(B), Ho);
  TRANSFER(Main, Hi);
END HiHo.
```

Processes gain real-time capabilities with the IOTRANSFER procedure. IOTRANSFER combines the concept of a transfer with that of processor **interrupts.** IOTRANSFER is similar to TRANSFER: the current process becomes inactive, and the specified process resumes execution. However, the next interrupt causes an unscheduled transfer back to the original process.

IOTRANSFER is used primarily in processes that control a computer's peripheral devices. (These are usually known as "interrupt handlers".) Peripheral devices are programmed to perform a specific operation; the devices then signal completion of the operation by interrupting the processor.

Peripheral devices can be considered as truly concurrent processes, for they operate in parallel with the execution of software processes.

NOTE- Because computers often have more than one peripheral device, processors often have more than one kind of interrupt. In order to distinguish interrupts caused by different devices, interrupts are assigned **interrupt vector** addresses. An interrupt is said to occur through its assigned interrupt vector address, thus identifying its origin. Interrupt vector addresses are system-dependent values.

IOTRANSFER has the following syntax:

```
    PROCEDURE IOTRANSFER(VAR OLD, NEW: PROCESS; VA: CARDINAL);
```

IOTRANSFER suspends the current process, assigns its execution state into the process variable OLD, and then resumes execution of the process identified by NEW. The next processor interrupt occurring through vector address VA causes an automatic TRANSFER(NEW, OLD): the currently executing process is suspended in NEW, and control is transfered to the interrupt-driven process. Note in the following example that when the interrupt-driven process completes its chore, the next IOTRANSFER call has the effect of resuming the process originally suspended by the interrupt.

Processors usually prioritize their interrupts according to the importance of the associated device; low-priority devices are prevented from interrupting the interrupt handlers of high-priority devices. Devices that operate at high speed or require immediate attention are assigned the highest priorities, ensuring them prompt servicing.

Modula-2 offers **module priorities** for controlling the occurrence of low-priority interrupts. A module's priority is specified in the module declaration just after the module identifier. The value associates a priority level with the module; the module's procedures can only be interrupted by occurrences of higher-priority interrupts. Thus, to create an interrupt handler for a device with priority n, the handler process is placed in a module declared with priority n. The procedure **LISTEN** temporarily lowers the current priority, allowing lower-priority interrupts to occur. LISTEN is most commonly used when a module must wait for one of its own interrupts (see example below).

NOTE- All but definition modules can be declared with module priorities. Modules lacking a priority specification have "null" priority; when called, their procedures inherit the calling module's priority. High-priority modules must not call procedures declared in lower priority modules; otherwise, the module's operational assumptions (e.g. critical sections) cannot be guaranteed. Module priority values are system-dependent.

Example of IOTRANSFER and module priority:

```
MODULE KeyBoard[4];  (* priority = 4 *)

  IMPORT ADR, SIZE, NEWPROCESS, LISTEN,
         PROCESS, TRANSFER, IOTRANSFER;

  EXPORT Read;

  CONST Q = 20;  enable = 6;

  VAR KeyStatus[177560B]: BITSET;
      KeyPort  [177562B]: CHAR;
      main, h: PROCESS;
      ps: ARRAY [1..100] OF WORD;
      queue: ARRAY [0..Q-1] OF CHAR;
      n, head, tail: CARDINAL;

  PROCEDURE Read(VAR ch: CHAR);
  BEGIN
    WHILE n = 0 DO LISTEN END;
    (* Assert: >= 1 characters queued *)
    ch := queue[head]; head := (head+1) MOD Q;
    DEC(n);
  END Read;

  PROCEDURE handler;
  BEGIN
    LOOP IOTRANSFER(h, main, 60B); (* va = 48 *)
      IF n < Q THEN
        queue[tail] := KeyPort;
        tail := (tail+1) MOD Q;
        INC(n);
      END;  (* ignore queue overflow *)
    END;
  END handler;

BEGIN  head := 0; tail := 0; n := 0;
  NEWPROCESS(handler, ADR(ps), SIZE(ps), h);
  INCL(KeyStatus, enable); TRANSFER(main, h);
END KeyBoard;
```

NOTE- The coroutine and interrupt facilities described in this section are system-dependent. See the **Implementation Guide** for more information on IOTRANSFER, interrupt vector addresses, and module priorities.

2.7 Procedure Variables

Modula-2 includes a new data type known as the **procedure type.** Variables declared with this type are called **procedure variables,** and take on procedures as values. Procedure variables are a generalization of Pascal's concept of procedure parameters; they are analogous (but not equivalent) to pointer variables, and can be thought of as "procedure pointers".

The only operations defined for procedure variables are assignment and invocation ("calling").

Calling a procedure variable invokes the procedure assigned to it. Procedure variable references are distinguished from procedure variable calls by the presence of a (possibly empty) parameter list. Consider the following declarations:

```
TYPE Cheese = (Jack, Cheddar, Swiss);

VAR G1, G2: PROCEDURE(Cheese,Cheese,Cheese);

PROCEDURE Grate(i,j,k: Cheese);
BEGIN
   ...
END Grate;
```

A "bare" occurrence of the procedure identifier Grate or the procedure variables G1 and G2 denotes the procedure as an object rather than a procedure call. For instance:

```
G1 := Grate;
G2 := G1;
```

An occurrence of the procedure identifier Grate or the procedure variables G1 and G2 with a parameter list denotes a procedure call:

```
Grate(Swiss, Jack, Cheddar);
G1(Swiss, Jack, Cheddar);
G2(Swiss, Jack, Cheddar);
```

Function procedures lacking a parameter list must be declared and called as follows (in order to distinguish them from procedure references):

```
PROCEDURE bald(): INTEGER;
 ...

  I := bald();
```

Modula-2 does not require empty parameter lists on normal procedure calls (e.g. "ProcCall;"), but it is good practice to use them anyways just to make the procedure calls stand out in the program text (e.g. "ProcCall();"). Note that Pascal does not allow empty parameter lists.

Procedure type checking is determined by the structure of the parameter lists; in particular, the order and types of the parameters must be identical.

Example of procedure type compatibility:

```
TYPE FuncKind = PROCEDURE(CHAR, VAR CARDINAL): INTEGER;

VAR F: FuncKind;

PROCEDURE Stuff(termch: CHAR; VAR val: CARDINAL): INTEGER;
BEGIN
  ...
END Stuff;

  ...

F := Stuff;  (* no parentheses on assignments *)
```

NOTE- Procedures are assignable only if they are declared at the global (outermost) level of a compilation unit. (Note that this includes procedures declared in global-level modules.) Standard procedures are not assignable; however, they can be "packaged" in a regular procedure declaration that is assignable.

Example of standard procedure "packaging":

```
PROCEDURE Ftrunc(r: REAL): CARDINAL;
BEGIN
  RETURN TRUNC(r);
END Ftrunc;
```

See 3.3.1, 3.4.1, and 3.5.2 for more information on procedure variables.

3 Differences From Pascal

This section describes differences between Pascal and Modula-2. It is divided into seven sections: **Vocabulary**, **Constants**, **Types**, **Expressions**, **Statements**, **Procedures**, and **Blocks**. Most of the differences are syntax changes; however, there is also a light sprinkling of new data types, operators, and statements.

NOTE- While this section is intended to be a complete description of Modula-2's differences from Pascal, it does not contain complete descriptions (e.g. syntax) of the Modula-2 features themselves. Such information can be found in the Modula-2 language report.

3.1 Vocabulary

Vocabulary includes identifiers, reserved words and symbols, and comments.

3.1.1 Identifiers

Identifiers are case-sensitive; for instance, the identifiers N and n are distinct, as are the identifiers FreeList and freelist.

NOTE- Prepare to have some problems with this rule at first; the longer you have been programming in Pascal, the more your mind is used to subconsciously mapping lower case to upper case (and vice versa). The problem manifests itself as an undeclared identifier flagged by the Modula-2 compiler which "obviously matches this declaration up here, see... whoops!".

Example of case-significant identifiers:

```
PROCEDURE Case;
  CONST N = 10;
  VAR   n: CARDINAL;
BEGIN
  n := 0;
  WHILE n < N DO
    LastSum := LastSum + (n*3);
    INC(n,2);
  END;
END Case;
```

Unlike Pascal, where only the first 8 characters can be assumed significant across most implementations, Modula-2 does not specify a standard significant identifier length; all characters in an identifier are considered significant.

The underscore character "_" — a valid character in many Pascal implementations — is not allowed in Modula-2 identifiers. It is a common practice in Modula-2 to capitalize the first letter of each word in multi-word identifiers.

Examples of Modula-2 identifiers:

```
N
succinct
AVeryLongIdentifier
LanguageTranslator
```

3.1.2 Reserved Words & Symbols

Reserved words must be written in capital letters. Though this "restriction" greatly improves program readability, reactionary Pascal programmers usually complain about it.

These Pascal reserved words are not present in Modula-2:

```
DOWNTO  FILE  GOTO  FUNCTION  PROGRAM  LABEL  PACKED
```

Modula-2's new reserved words include:

```
BY  DEFINITION  ELSIF  EXIT  EXPORT  FROM  IMPLEMENTATION
IMPORT  LOOP  MODULE  POINTER  QUALIFIED  RETURN
```

NOTE- NIL, a reserved word in Pascal, is now a standard identifier.

All of Pascal's nonalphabetic symbols (':=', '>=', etc.) are included in Modula-2, along with three new symbols. The vertical bar '|' serves as a delimiter in record variants and CASE statements. The ampersand '&' is an abbreviation for the reserved word AND. The pound sign '#' is an abbreviation for the reserved symbol '<>' ('#' denotes a crossed-out equal sign).

Examples of new symbols and reserved words:

```
IF i # 4 THEN WriteString("BigWhoop") END;

WHILE (n <= 10) & (a[n] # nul) DO INC(n) END;

CASE i OF
  1:   WriteString("one") |
  2:   WriteString("two") |
  3:   WriteString("many")
END;
```

3.1.3 Comments

Modula-2's comments are similar to those of Pascal, but with a couple of differences. First, Modula-2 allows only this form of comment:

```
(* <your comment here> *)
```

Braces ("{" and "}") cannot be used as comment delimiters; Modula-2 uses them to delimit set constants.

Unlike Pascal, comments may be nested.

Example of nested comments:

```
(*
   WriteString("This is not a test");
   (* The best defense is a good offense *)
   WriteString("For the next 60 million years...");
*)
```

3.2 Constants

Unlike Pascal, Modula-2 allows constant expressions everywhere that constants can be used. Constant expressions are useful in declaring constants and types that depend on other constant values. Constant expressions may not contain variable references or function calls; otherwise, there are no restrictions on their use.

Example of constant expressions:

```
CONST N = 4;
      MaxLength = 2*N;
      LastElement = MaxLength-1;
      SetExpression = {0,1,2} * {2..4};

TYPE  Elements = ARRAY [0..MaxLength-1] of INTEGER;
```

NOTE- The value ranges displayed in this section are for machines with 16-bit words.

3.2.1 Integers

Integer constants specify constant values for types INTEGER and CARDINAL (unsigned integer). Constant values range between -32768 and 65535. A constant's value determines whether it is compatible with type INTEGER or CARDINAL. Constants in the range -32768 to -1 are compatible only with INTEGER. Constants in the range 0 to 32767 are compatible with both INTEGER and CARDINAL. Constants in the range 32768 to 65535 are compatible only with CARDINAL.

Integer constants can be specified in three radices: decimal, hexadecimal, and octal.

Decimal constants are written as in Pascal.

Examples of decimal constants:

```
38      1982    29999   (*CARDINAL and INTEGER compatible*)
32768   40000   49999   (*CARDINAL compatible*)
-8      -2000   -32767  (*INTEGER compatible *)
```

Hexadecimal constants are constructed from hex digits ('0'..'9', 'A'..'F'), and are terminated with the letter H. Hex values range from 0H to 0FFFFH. Note that hex constants must begin with a decimal digit; thus, a leading '0' digit must be added to hex constants beginning with an alphabetic hex digit.

Examples of hex constants:

```
0H      3AH     247H    0BEACH
```

Octal constants are constructed from the octal digits ('0'..'7'), and are terminated with the letter B. Octal values range from 0B to 177777B.

Examples of octal constants:

```
0B      37B     1777B    177560B
```

3.2.2 Reals

The format of real constants is similar to Pascal, but with a couple of minor differences. Real numbers require a decimal point. The exponent character is denoted by 'E' only ('e' is not valid in Modula-2).

Invalid real constants:

```
1          1.03e24          1E10
```

Valid real constants:

```
1.0     1.      1.03E24      1.E10      6.023E-23
```

3.2.3 Characters

Character constants are compatible with type CHAR. Character constants can be specified in two forms: character values, and ordinal values. Character values consist of a single character delimited either by single or double quotes. Ordinal character constants consist of an octal value followed by the letter C.

Examples of character constants:

```
'A'     '!'     "'"     "@"     15C
```

3.2.4 Strings

String constants are similar to Pascal. The only syntactic difference is the method used to handle embedded quotes. Modula-2 does not use Pascal's method of denoting single quotes as quote pairs; instead, strings are delimited either by single or double quotes. Thus, if a string contains single quotes, it is delimited by double quotes. If a string contains double quotes, it is delimited by single quotes. (Note that this implies a string cannot contain both single and double quotes.)

Examples of string constants:

```
"We're strings, and you aren't!"

'Thanks, Al!'

'This sentence contains a "string constant".'
```

Strings must contain more than one character to qualify as string constants; quoted single characters are compatible only with type CHAR.

Unlike set constants, string constants are not explicitly typed. The implicit type of an N-character string constant is:

```
ARRAY [0..N-1] OF CHAR
```

NOTE- String constants cannot extend past the end of a source text line.

NOTE- Modula-2 is less strict than Pascal on type compatibility of character arrays and string constants; in particular, string constants may be assigned to character arrays longer than the string itself. See 3.5.1 for more information.

3.2.5 Sets

Set constants differ from Pascal in a few ways. Constants are delimited by braces ('{' and '}') rather than by square brackets, and set elements are now limited to (subranges of) constant expressions. Modula-2 remedies this restriction by providing the standard procedures INCL and EXCL (see 3.6.4 for details). Set constants can also be explicitly typed by preceding them with a type identifier.

Example of set constants:

```
PROCEDURE CheckChar;
  TYPE CharSet = SET OF CHAR;
  VAR  ch: CHAR;
       Valid: CharSet;
BEGIN
  ...
  IF ch IN CharSet{'a'..'c'} THEN
    INCL(Valid,ch);
  END;
  Valid := Valid + CharSet{'a','z'};
  ...
END CheckChar;
```

In the previous example, note how the set constants are preceded by the type identifier CharSet. This practice is foreign to Pascal, where a set constant's type is determined by its elements. Modula-2, on the other hand, rigidly enforces type checking in set expressions; set constants must be explicitly typed to match the other set operands.

Set constants lacking a preceding type identifier default to the standard type BITSET (see 3.3.7 for details).

More examples of set constants:

```
       {1,2,4,8}        (* These are identically typed *)
BITSET {1,2,4,8}
```

3.3 Types

This section describes differences from Pascal types, and introduces two new types: the type CARDINAL (unsigned integers), and procedure types. Note that Pascal's file type is missing; files are now provided by standard modules (see 2.4 for details).

3.3.1 Procedures

If you have not read it yet, see 2.7 for an introduction to procedure variables.

Variables declared with a procedure type are assigned procedures as values. Procedure type declarations may include parameter lists; in order to be type compatible with a procedure variable, procedures must have the same parameters as the procedure variable's type. In particular, the order and types of the procedure's parameters must correspond to those of the procedure type declaration.

Procedure variables cannot be assigned standard procedures or procedures declared local to another procedure.

Modula-2 includes the standard type PROC which denotes a parameterless procedure. The formal definition of PROC is:

```
TYPE PROC = PROCEDURE;
```

Examples of procedure types:

```
    TYPE  ProcType = PROCEDURE (CARDINAL, VAR INTEGER, CHAR);
          FuncType = PROCEDURE(): CARDINAL;
          Shortype = PROC;
```

3.3.2 Cardinals

Modula-2 provides the type CARDINAL for unsigned (i.e. "cardinal") integer operations. Cardinal variables take on the range 0 to 65535. Cardinals are used just like integers; all integer operations are also available for cardinals.

Cardinal variables are assignable to integer variables (and vice versa); however, cardinal and integer variables cannot be mixed in expressions.

WARNING- Beware of cardinal underflow; i.e. cardinal variables becoming "less than" 0. Most implementations do not perform underflow checking; because "negative" results are treated as large cardinal values, subsequent comparisons will not work correctly (e.g. i-j < k, where i < j).

Example of type CARDINAL:

```
PROCEDURE MixNumbers;
  VAR a,b: INTEGER;
      l,m: CARDINAL;
BEGIN
  m := 60000;
  l := 30000;
  a := l;        (* this assignment is legal *)
  b := l+m-a; (* this expression is illegal *)
END MixNumbers;
```

3.3.3 Characters

Type CHAR is the same as in Pascal. Modula-2 defines the underlying character set to be ASCII, eliminating the problems caused by trying to accommodate different character sets. In particular, Modula-2 programs can take advantage of the character set ordering.

3.3.4 Subranges

Subrange types have one syntactic difference from Pascal. Subrange specifications are enclosed in square brackets.

Example of subrange declarations:

```
PROCEDURE SubrangeStuff;

  TYPE  GoodNums = [0..N-1];
        Alphabet = ['A'..'Z'];
        WeekDay = [Monday..Friday];
  VAR   Num: GoodNums;
        Char: Alphabet;
        Day: WeekDay;

BEGIN
  Num  := 4;
  Char := 'G';
  Day  := Friday;
END SubrangeStuff;
```

NOTE- This syntax change affects array declarations (see below for details).

3.3.5 Arrays

Array declarations are similar to Pascal, with only one difference. When a subrange identifier is used to specify the array index bounds, the square brackets are left out of the array declaration. (Explicitly declared index subranges are delimited by brackets, as in Pascal.)

Example of array declarations:

```
PROCEDURE Arrays;

  TYPE  GoodNums = [0..N-1];
        Alphabet = ['A'..'Z'];
        WeekDay  = [Monday..Friday];

  VAR   Num: ARRAY GoodNums OF CARDINAL;
        Alphabetic: ARRAY ['A'..'Z'] OF CHAR;
        Matrix: ARRAY [1..10], [1..20] OF REAL;
        Mixup: ARRAY Alphabet, WeekDay OF GoodNums;

BEGIN
  Num[4] := 56;
  Alphabetic['G'] := '1';
  Matrix[5,5] := 3.14159;
  Mixup['A',Monday] := 4;
END Arrays;
```

3.3.6 Records

The only differences in records involve record variants. Records can contain several case variant parts; unlike Pascal, each variant part terminates with an END symbol. A number of minor syntactic differences arise in variant part declarations. Case label lists can contain constant expressions and subranges. Variants can declare an ELSE field which catches unspecified case values. (The CASE statement includes a corresponding ELSE part for accessing this field.) Pascal's use of parentheses to delimit variants is replaced by separating variant declarations with a vertical bar '|'. (Note that '|' cannot appear before an ELSE field.)

NOTE- "Free" variants (such as the record KludgeRec in the following example) need no longer serve as tools for type abuse. Modula-2 provides better facilities for breaking type compatibility rules (see 2.5 for details).

Example of record declarations:

```
TYPE BirthDate = RECORD
                   Day:   [1..31];
                   Month: [Jan..Dec];
                   Year:  [0..99];
                 END;

      TrainRec = RECORD
                   CASE tag1: CARDINAL OF
                     0..9: x,y:   Letters |
                     11:   a,b:   Letters
                   ELSE  i,j: INTEGER
                   END;
                   Date: BirthDate;
                   Size: [8..15];
                   CASE tag2:      BOOLEAN OF
                     FALSE: r:     INTEGER |
                     TRUE:  s:     REAL
                   END;
                 END;

     KludgeRec = RECORD
                   CASE BOOLEAN OF
                     TRUE:  I:  INTEGER |
                     FALSE: C:  CARDINAL
                   END;
                 END;
```

3.3.7 Sets

Sets are relatively unchanged in Modula-2. As mentioned before, set values are delimited by braces (rather than brackets) and are explicitly typed (with a type identifier prefix).

Modula-2 defines the standard type BITSET as a set which fits in one machine word. Set operations are more efficient with bitsets than with larger sets.

The formal definition of BITSET is:

```
TYPE BITSET = SET OF [0..WordSize-1];
```

The following example uses BITSET to efficiently implement sets of arbitrary length:

Example of BITSET:

```
MODULE PowerSets;
  EXPORT PowerSet, Included,
         Include,  Exclude;

  CONST  WordSize = 16; SetSize = 100;

  TYPE   PowerSet = ARRAY [0..SetSize-1] OF BITSET;

  PROCEDURE Included(S: PowerSet; Bit: CARDINAL): BOOLEAN;
  BEGIN
    RETURN (Bit MOD WordSize) IN S[Bit DIV WordSize];
  END Included;

  PROCEDURE Include(VAR S: PowerSet; Bit: CARDINAL);
  BEGIN
    INCL(S[Bit DIV WordSize], Bit MOD WordSize);
  END Include;

  ...

END PowerSets;
```

3.3.8 Pointers

Pointer type declarations have a new syntax. The "up arrow" symbol "^", though still used in pointer references, has been replaced in pointer declarations with the reserved word sequence "POINTER TO". Pointer declarations are no longer restricted to type identifiers; any type or type structure can be named as the pointer's type.

Example of pointer declarations:

```
TYPE P  = POINTER TO INTEGER;
     P2 = POINTER TO
              RECORD
              a,b,c: BOOLEAN;
              END;

     MSCWP=   POINTER TO MSCW;
     MSCW =   RECORD
                Stat: MSCWP;
                Dyn:  MSCWP;
                IPC: CARDINAL;
              END;
```

3.4 Expressions

This section describes things worth knowing about Modula-2 expressions — the differences from Pascal are mostly minor.

3.4.1 Function Operands

Function procedures (i.e. procedures which return function results) can be referenced two ways within an expression. Function procedure identifiers accompanied by a (possibly empty) parameter list denote function procedure calls; the expression value is the value returned by the function procedure. Procedure identifiers lacking parameter lists refer to the procedure itself; this type of reference is used for assigning values to procedure variables.

NOTE- Function procedure calls cannot be selected; e.g. the expressions "func()^" and "func()[4].name" are not legal.

Example of function procedures and procedure variables:

```
MODULE FuncDemo;

  VAR i:   INTEGER;
      p1:  PROCEDURE(INTEGER): INTEGER;
      p2:  PROCEDURE(): INTEGER;

  PROCEDURE Func1(arg: INTEGER): INTEGER;
  BEGIN RETURN arg DIV 2 END Func1;

  PROCEDURE Func2(): INTEGER;
  BEGIN RETURN 77 END Func2;

BEGIN
  i  := Func1(7);     (* call Func1          *)
  p1 := Func1;        (* assign Func1 to p1  *)
  i  := p1(7);        (* call Func1 thru p1  *)
  i  := Func2();      (* call Func2          *)
  p2 := Func2;        (* assign Func2 to p2  *)
  i  := p2();         (* call Func2 thru p2  *)
END FuncDemo.
```

3.4.2 Operators

The operations '+', '-', '*', DIV, and MOD apply to cardinals in addition to integers and subrange variables. Unary '-' does not apply to cardinals. '/' denotes real division. Note that MOD is not defined for negative arguments.

The logical operators AND and OR are evaluated conditionally — they short-circuit expression evaluation if the expression result can be determined by the value of the left-hand argument.

```
p AND q   is equivalent to  "IF p THEN q
                              ELSE FALSE"

p OR  q   is equivalent to  "IF p THEN TRUE
                              ELSE q"
```

These definitions of AND and OR allow shorter, more efficient solutions to many programming problems. Beware of using functions with side effects as expression operands, however; the functions might not be called if expression evaluation is short circuited.

In the following example, conditional evaluation prevents a potential NIL pointer reference:

```
WHILE (Event <> NIL) AND (Event^.Time < Now) DO
  Event := Event^.Next;
END;
```

The relational operators AND and '<>' have alternate single-character names: '&' for AND, and '#' for '<>'.

Example of abbreviated operator names:

```
WHILE (i <= ArrayLength) & (A[i] # nul) DO
  INC(i);
END;
```

In addition to the standard set operators '+' (union), '-' (difference), '*' (intersection), and IN (inclusion), Modula-2 defines the set operator '/', which is defined as symmetric set difference. Symmetric set difference performs a bitwise exclusive OR operation.

3.4.3 Mixed Expressions

Operands of the types INTEGER, CARDINAL, and REAL cannot be freely mixed in expressions; unless type transfer functions are used, expressions must consist entirely of integers (including integer subranges), cardinals (including cardinal subranges), or reals.

Pascal allows integers and reals to be mixed in expressions; integer operands in real expressions are implicitly converted to reals. Modula-2 does not allow mixed integers and reals; instead, integer operands must be explicitly converted to type REAL with the standard procedure FLOAT. FLOAT accepts arguments of type CARDINAL and returns the equivalent real value.

For real-to-integer conversions, Modula-2 includes Pascal's standard function TRUNC(x). TRUNC accepts real arguments and returns a value of type CARDINAL. (Note that Modula-2 does not include ROUND(x).)

NOTE- Many Modula-2 implementations define TRUNC and FLOAT to work with type INTEGER.

Example of conversion between reals and integers:

```
PROCEDURE Numbers;
  VAR i,j,k: CARDINAL;
       x,y,z: REAL;

BEGIN
  i := j + TRUNC(z);
  x := y + FLOAT(k);
END Numbers;
```

Operands of type WORD are not compatible with other operand types in expressions.

Operands of type ADDRESS are compatible with cardinals and pointers in expressions; however, some interesting side effects can arise from the left-to-right order of expression parsing.

Consider the following example:

```
PROCEDURE Miscible;
  VAR Ptr: POINTER TO INTEGER;
      Addr: ADDRESS;

BEGIN
  Ptr := Ptr + Addr + 4;
  Ptr := Addr + Ptr + 4;
END Miscible;
```

The two expressions in this example are legal because the presence of the operand Addr converts the expression type to ADDRESS, which is compatible with the integer/cardinal constant. (In fact, the constant 4 in this example can be considered a pointer constant like NIL).

Now consider the following example:

```
PROCEDURE Immiscible;
  VAR Ptr: POINTER TO INTEGER;
      Addr: ADDRESS;

BEGIN
  Ptr := Ptr + 4 + Addr;
END Immiscible;
```

The expression in this example will be flagged by the Modula-2 compiler as erroneous, because pointer types are not compatible with integer/cardinal constants. (The expression is not known to be of type ADDRESS, because the operand Addr has not been parsed yet.) Thus, it can be seen that ordering restrictions exist for mixed address/pointer expressions. See 2.5 for more information on ADDRESS.

Finally, as mentioned before, set operands must possess the same type in order to be expression compatible (see 3.3.7 for details).

3.5 Statements

The major difference involves the reorganization of structured statements around statement sequences rather than compound statements.

In Modula-2, all structured statements end with an explicit closing symbol (UNTIL for the REPEAT statement, END for the rest). Pascal's compound statement — one or more statements delimited by the symbols BEGIN/END — does not exist in Modula-2; it has been replaced by the concept of **statement sequences.** Statement sequences are series of statements separated by semicolons; sequences are delimited by the enclosing structure rather than by explicit delimiting symbols.

Examples of statement structures:

```
PROCEDURE Structures;
BEGIN
  IF i > 0 THEN;
    WriteString('Truth');
    i := -1;
  END;

  WHILE j > 3 DO
    i := 4;
    DEC(j)
  END;

  REPEAT
    ;;;;;;;;
  UNTIL TRUE;
END Structures;
```

NOTE- Modula-2's use of statement sequences allows semicolons to be used more freely than in Pascal.

Pascal's GOTO statement is missing from Modula-2; in its place are the LOOP/EXIT and RETURN statements and the standard procedure HALT (3.6.4). LOOP/EXIT statements are used to express repetitive statement sequences which contain several exit points. RETURN is a limited form of GOTO; it transfers control to the end of the current procedure. HALT terminates execution of the current program.

3.5.1 Assignment Statements

This section presents the type compatibility rules for assignment. (3.6.2 describes parameter type compatibility.)

Operands are said to be **assignment compatible** if they are allowed to be assigned to each other.

The primary rule for assignment compatibility is that operands must be **compatible.** Operands are compatible if they are of the same type: a variable's type is determined by the type identifier it is declared with, while a constant's type is (usually) implicit. Operands are also compatible if one is declared as a subrange of the other, or if both operands are declared as subranges of the same type.

NOTE- Structured variables (e.g. records and arrays) are compatible only if they share the same type definition; in particular, variables declared with similarly structured types are not compatible.

The types INTEGER and CARDINAL (and their subranges) are defined as assignment compatible. (Note that they are not compatible in expressions.)

NOTE- Assignment compatibility of subranges and INTEGER/CARDINAL types implies the possibility of assigning illegal values to variables at run-time; thus, assignments of this kind — though valid at compile-time — may cause an execution error (i.e. "Range error") at run-time.

The type WORD is compatible only with itself. Operands of type ADDRESS are compatible with pointer types and CARDINALs.

Unlike Pascal, string constants are assignment compatible with string variables (i.e. 0-based character arrays) whose length exceeds that of the string. If the string is shorter than the array, the assignment operation places a null character (0C) into the array following the string constant.

Example of string assignment:

```
PROCEDURE Sass;
  VAR S1, S2: ARRAY[0..10] OF CHAR;
BEGIN
  S1 := "Short";          (* S1[5] contains 0C *)
  S2 := '123456789AB';    (* string fits exactly *)
END Sass;
```

Procedure types are compatible if the order and type of their formal parameters are the same. Procedures are assignable only if they are globally declared (i.e. not declared within another procedure). Standard procedures are not assignable.

Example of procedure type compatibility:

```
MODULE Procedures;

  PROCEDURE Demo(Ch: CHAR; I,J: INTEGER);
  BEGIN ... END Demo;

  VAR  P2: PROCEDURE(CHAR,
                     INTEGER,
                     INTEGER);

BEGIN
  P2 := Demo;
END Procedures.
```

3.5.2 Procedure Calls

Procedure calls are similar to those in Pascal; they consist of an identifier possibly followed by a list of parameters enclosed in parentheses. In Modula-2, the identifier can be either a procedure identifier or a procedure variable.

Unlike function procedure calls, regular procedure calls do not require a parameter list if the procedure contains no parameters. However, it is good practice to place empty parameter lists after the identifiers anyways (just to mark them as procedure calls).

Example of procedure calls:

```
MODULE ProcCall;

  PROCEDURE GlobalProc;
  BEGIN ... END GlobalProc;

  VAR P: PROC;

BEGIN
  GlobalProc;          (* call GlobalProc         *)
  GlobalProc();        (* call it again           *)
  P := GlobalProc;     (* assign GlobalProc to P  *)
  P;                   (* call GlobalProc thru P  *)
  P();                 (* call it again thru P    *)
END ProcCall.
```

3.5.3 WHILE Statements

The only difference with WHILE statements is the new syntax which requires the closing symbol END.

Examples of WHILE statements:

```
WHILE i > 0 DO DEC(i) END;

WHILE A[J] <= 0 DO
  A[J] := A[J] + A[J-1];
  INC(J);
END;
```

3.5.4 IF Statements

The IF statement requires the closing symbol END, and also contains the new symbol ELSIF which allows a single IF statement to express cascaded conditions.

The basic forms of the IF statement are:

```
IF <condition> THEN
  <statement sequence>
END;

IF <condition> THEN
  <statement sequence>
ELSE
  <statement sequence>
END;
```

Cascaded conditionals are written as:

```
IF <condition1> THEN
  <statement sequence>
ELSIF <condition2> THEN
  <statement sequence>
 ...
ELSIF <conditionN> THEN
  <statement sequence>
END;

IF <condition1> THEN
  <statement sequence>
ELSIF <condition2> THEN
  <statement sequence>
 ...
ELSE
  <statement sequence>
END;
```

Examples of IF statements:

```
IF i > 0 THEN i := 0 END;

IF scanning THEN
  GetNextSymbol
ELSIF skipping THEN
  FlushBuff
ELSE
  RETURN NoSymbol
END;
```

3.5.5 FOR Statements

The FOR statement requires the closing symbol END. As in Pascal, the default step value is 1; unlike Pascal, step values other than 1 can be specified. Step values are restricted to constants. Note that the symbol DOWNTO is missing from Modula-2; it is equivalent to a step value of -1.

The control variable cannot be part of a structured variable, nor imported, nor a parameter.

NOTE- As in Pascal, the FOR statement can step through any scalar values; however, integer constants are still used as step values.

Examples of FOR statements:

```
FOR i := 1 TO 10 DO A[i] := 4 END;

FOR j := 1 TO 9 BY 2 DO
  B[j] := A[j-1];
  WriteInt(j,3);
END;

FOR ch := 'z' TO 'a' BY -1 DO
  Alfa[ch] := '?';
END;
```

3.5.6 WITH Statements

The WITH statement requires the closing symbol END. Unlike Pascal, WITH accepts only one variable reference; a separate WITH statement is needed to unqualify each record variable.

Examples of WITH statements:

```
WITH t^.Person DO
  WriteString(Name);
  BlackListed := TRUE;
END;

WITH CreditCheck DO
  WITH DriversLicense DO
    Debtor := Name;
    License := Number;
  END
END;
```

3.5.7 CASE Statements

Subranges and constant expressions are allowed in case label lists. If the case value does not match any of the case labels, the statement sequence following the symbol ELSE is selected (but if the ELSE part is left unspecified, an execution error (value range error) occurs).

Statement sequences for each case are separated by the vertical bar "|". (Note that a "|" cannot appear before an ELSE part.)

Examples of CASE statements:

```
CONST N = 3;

CASE j OF
  0..9:  DEC(i,10);
         Stop := FALSE |
  10,11: Stop := TRUE  |
  200:   TuneBackEnd   |
  N+2:   DecipherCase;
         Stop := TRUE
  ELSE HALT
END;

CASE B OF
  FALSE: Proceed(3,4,5) |
  TRUE:  XXX := 102
END;
```

3.5.8 LOOP/EXIT Statements

The LOOP statement specifies cyclic execution of a statement sequence; in particular, it addresses two distinct programming situations that are not handled well by Pascal's repetitive statements. LOOPs without EXIT statements serve as "cycle" statements; they express the endless repetition of a group of statements. LOOPs containing EXIT statements are used to express repetitive statement sequences which have special exiting requirements; i.e. several exit points or a single exit point in the middle of the statement sequence.

EXIT statements can appear anywhere in a LOOP statement. EXIT transfers control to the statement following the LOOP statement.

NOTE- The LOOP statement represents a generalized form of repetition. In theory, WHILE, REPEAT, and FOR statements all can be expressed as LOOP statements containing a single EXIT. In practice, the resulting program would be less understandable. Use WHILE, REPEAT, and FOR whenever possible; use LOOP only when necessary.

Examples of LOOP statements:

```
LOOP
  GetAQuarter;
  PlayGame;
  WriteString("Sorry, you lose. Try again!");
END;

LOOP
  WITH Node^ DO
    IF Name = ID THEN
      EXIT;
    ELSIF Name < ID THEN
      IF LLink = NIL THEN EXIT
      ELSE Node := LLink END;
    ELSE
      IF RLink = NIL THEN EXIT
      ELSE Node := RLink END;
    END;
  END;
END;
```

3.5.9 RETURN Statements

The RETURN statement has two forms, and serves two purposes. In procedures and modules, RETURN terminates the enclosing procedure or module body. In function procedures, the RETURN symbol is always followed by an expression; the resulting statement assigns the expression value to the function result and then terminates the function procedure.

Non-function procedures and modules contain implicit RETURN statements at the end of their bodies. Explicit RETURN statements are optional; they indicate additional (possibly exceptional) termination points in the body.

NOTE- The Pascal equivalent of RETURN from a (non-function) procedure is a GOTO which jumps to the end of the procedure.

Example of RETURN in a procedure:

```
PROCEDURE ReadSequence(VAR Ch: CHAR);
BEGIN
  IF SequenceError THEN
    Ch := ' ';
    Propagate := TRUE;
    RETURN;     (* exit from procedure *)
  END;
  InChar(Ch);
  CheckErrors;
END ReadSequence;
```

In function procedures, RETURN performs the double duty of terminating the procedure and assigning the function result. For this reason, function procedures must terminate by executing a RETURN statement, and all RETURNs must be accompanied by an expression. (Note that the expression type must match the function result type.)

NOTE- The Pascal equivalent of a function procedure RETURN is a combination of assignment to the function variable and a GOTO which jumps to the end of the function block.

Example of RETURN in a function procedure:

```
PROCEDURE Signum(Freud: INTEGER): INTEGER;
BEGIN
  IF Freud > 0 THEN RETURN 1
  ELSIF Freud < 0 THEN RETURN -1
  ELSE RETURN 0;
END Signum;
```

3.6 Procedures and Functions

The only changes to procedures and functions are a new syntax for declaring functions and the addition of open array parameters. Modula-2 includes most of Pascal's standard procedures — a few are missing, and a few new ones have appeared.

3.6.1 Function Procedures

Functions are called **function procedures** in Modula-2. Function procedures are equivalent to Pascal's functions; the differences are mostly syntactic. Function procedure headings are virtually identical to regular procedure headings, the only difference being the presence of a function result type following the parameter list. Note that a (possibly empty) parameter list must precede the function result type declaration. Within function procedures, function results are returned with the RETURN statement. Function procedure calls consist of a procedure identifier followed by a (possibly empty) parameter list.

NOTE- Function procedure calls cannot be selected; e.g. the expressions "func()^" and "func()[4].name" are not legal in Modula-2.

Examples of function procedures:

```
PROCEDURE FunctionDemo;

  PROCEDURE IOResult(): CARDINAL;
  BEGIN
    ...
    RETURN 0 (* no error *)
  END IOResult;

  VAR IO: CARDINAL;

BEGIN
  ...
  IO := IOResult();
END FunctionDemo;
```

3.6.2 Parameter Type Compatibility

As in Pascal, there are two kinds of parameters: value parameters, and variable parameters. With value parameters, the actual parameter must be assignment compatible with the formal parameter. With variable parameters, the formal and actual parameters must have identical types.

NOTE- The types WORD and ADDRESS and open array parameters are exceptions to these rules.

If a formal parameter specifies a procedure type, corresponding actual parameters must be either global procedures, procedure variables, or procedure parameters; in all cases, the procedure types must be compatible. Standard procedures cannot be passed as procedure parameters.

3.6.3 Open Array Parameters

Modula-2 allows formal parameter types of the form:

```
ARRAY OF T
```

where T is an arbitrary base type. Note that the array bounds are omitted — this is known as an **open array parameter** and is compatible with all (one dimensional) arrays having the base type T.

Array elements of actual parameters are mapped into the range 0..N-1, where N is the number of elements in the actual parameter. The high bound of a open array parameter is obtained with the standard procedure HIGH(A), which returns the index of the high bound of array A. HIGH works on all arrays.

NOTE- Open arrays can only be accessed element-wise; they are not assignable as entire objects. Open arrays can be passed as actual parameters to other procedures containing open array parameters. If an empty string is passed to an ARRAY OF CHAR, HIGH returns 0 (the position of the terminating null character).

In the following example, B2 is an 11-element array whose indices range from 5 to 15. Inside the procedure Invert, B2 is viewed as A, an 11-element open array parameter whose indices range from 0 to HIGH(A). (In this case, HIGH(A) returns the value 10).

Example of open array parameters:

```
PROCEDURE DynArray;

  PROCEDURE Invert(VAR A: ARRAY OF REAL);
    VAR inx: CARDINAL;
  BEGIN
    FOR inx := 0 TO HIGH(A) DO
      A[inx] := 1.0 / A[inx];
    END;
  END Invert;

  VAR B2: ARRAY [5..15] OF REAL;

BEGIN
  ...
  Invert(B2);
  ...
END DynArray;
```

Open arrays are useful for writing general-purpose numerical and string-handling routines; for instance, the standard utility module Strings uses open array parameters to handle string arguments of any length.

If a formal parameter has the form ARRAY OF WORD, its corresponding actual parameter can be of any type; all parameter types (particularly records and sets) are treated as multi-word arrays.

Example of ARRAY OF WORD:

```
PROCEDURE Generic;

  PROCEDURE DisplayHex(A: ARRAY OF WORD);
    VAR inx: CARDINAL;
  BEGIN
    FOR inx := 0 TO HIGH(A) DO
      WriteHex(CARDINAL(A[inx]), 7);
      WriteLn;
    END:
    WriteLn;
  END DisplayHex;

  VAR AR: ARRAY [1..5] OF REAL;
      c:  CARDINAL;
      b:  BOOLEAN;
      R:  RECORD
            a,b,c: INTEGER;
            Ch: CHAR;
          END;

BEGIN
  ...
  DisplayHex(AR);
  DisplayHex(c);
  DisplayHex(b);
  DisplayHex(R);
END Generic;
```

3.6.4 Standard Procedures

Standard procedures are automatically imported into every module. (Note that this implies standard procedures may be redefined within procedures, but not within modules.)

Pascal's old favorites ABS, ODD, ORD, and CHR live on in Modula-2, but with a twist; ORD now returns a cardinal result.

ABS(x)	absolute value. x is an integer or real; result type = argument type
ODD(x)	return Boolean result TRUE if the integer/cardinal expression x is odd.
ORD(x)	ordinal value of x. x is an enumeration, character, integer, or cardinal; result type is cardinal.

CHR(x)	return the character with ordinal value x. x is a cardinal.

Modula-2 provides the standard procedure VAL, which performs the inverse operation of ORD.

VAL(T,x)	return the value with ordinal number x and type T. x is a cardinal. T is an enumeration, character, integer, or cardinal.

For instance, given the type:

Day = (Monday, Tuesday, Wednesday)

VAL(Day,1) returns the scalar constant Tuesday. The relationship between ORD and VAL is

VAL(T, ORD(x)) = x, if x is of type T.

TRUNC truncates a real value to its cardinal component. FLOAT performs the opposite action; it converts cardinal arguments to real values. (ROUND is not included in Modula-2.)

TRUNC(x)	return the integral part of the real number x.
FLOAT(x)	return the real number representation of the cardinal x.

Pascal's PRED and SUCC have been replaced with the more general operations INC and DEC. INC and DEC have two forms. INC(x) and DEC(x) replace x with its immediate successor or predecessor. INC(x,n) and DEC(x,n) replace x with its n'th successor or predecessor.

NOTE- INC and DEC accept characters, integers, and addresses along with subranges and enumerations. Note that "n" can be any expression compatible with "x".

INC(x)	x := x + 1
INC(x,n)	x := x + n
DEC(x)	x := x - 1
DEC(x,n)	x := x - n

Modula-2 provides the standard procedures INCL and EXCL for set manipulation. INCL ("include") adds a single element to a set; EXCL ("exclude") removes a single element from a set. Note that "e" can be any expression compatible with the base type of "s".

```
INCL(s,e)       s := s + {e}

EXCL(s,e)       s := s - {e}
```

The procedures NEW(p) and DISPOSE(p) perform the usual actions; however, they are translated into calls to the procedures ALLOCATE and DEALLOCATE which are usually provided by the standard utility module Storage (see 2.4 for details).

The procedure HALT terminates program execution. HALT is used to stop programs which detect unrecoverable error conditions.

The function procedure HIGH(A) returns a cardinal indicating the high index bound of the array A. HIGH is commonly used with open array parameters (see 3.6.3); however, it also accepts regular array variables.

Last (and probably least), the function procedure CAP(ch) returns the upper-case equivalent of lower-case character arguments.

3.7 Blocks

Despite the emphasis on modules, blocks still play an important part in Modula-2: implementation modules, program modules, local modules, and procedures share the same block syntax. Differences from Pascal include relaxed order of declarations, termination of all blocks by a procedure or module identifier, and optional statement parts.

Pascal imposes a strict order on the declaration of objects; within any given block, labels must be declared before constants, constants before types, types before variables, and so on. Modula-2 eliminates this restriction — declarations can appear in any order. Programs containing lots of declarations are easier to read and understand when related declarations are grouped together (regardless of their kind).

Example of relaxed declaration order:

```
MODULE Xlator;

  CONST MaxSym = 1024;
  TYPE   SymBuffer = ARRAY[1..MaxSym] OF CHAR;
  VAR     SymBuff1, SymBuff2: SymBuffer;
  ...

  CONST MaxCode = 512;
  TYPE   CodeBuffer = ARRAY[1..MaxCode] OF CHAR;
  VAR     CodeBuff: CodeBuffer;
  ...

END Xlator.
```

Every module and procedure declaration is terminated by its identifier. This improves the readability of large programs containing many levels of nested blocks.

Example of block identifiers:

```
MODULE Turboincabulator;

  MODULE Widget;

    PROCEDURE Stuff;
      ...

      PROCEDURE Nested;
        ...
      BEGIN
        ...
      END Nested;

    BEGIN
      ...
    END Stuff;
  ...

  END Widget;
  ...

END Turboincabulator.
```

Procedure and module bodies appear at the end of a block and are delimited by the symbols BEGIN and END.

As mentioned before, module bodies are optional; if present, they serve to initialize a module's variables.

Example of optional module bodies:

```
MODULE NoBody;
  EXPORT c,d;

  VAR c,d: CARDINAL;

END NoBody;

MODULE Body;
  EXPORT a,b;

  VAR a,b: CARDINAL;

BEGIN
  a := 1; b := 2;
END Body;
```

Note, however, that procedure bodies are optional, too! This seems nonsensical, as the only operation defined for a procedure is invocation, which causes the procedure body to be executed; thus, a bodyless procedure would appear incapable of performing any useful actions.

Bodyless procedures are justified because they can include local module declarations. Recall that module bodies actually belong to the body of the enclosing procedure; thus, a bodyless procedure has a de facto procedure body consisting of the module bodies from locally declared modules.

Example of a bodyless procedure:

```
PROCEDURE NoBody;

  MODULE Action1;
    ...

  BEGIN
    ...
  END Action1;

  MODULE Action2;
    ...

  BEGIN
    ...
  END Action2;

END NoBody;
```

Appendix 1 Reserved Words and Symbols

NOTE- All characters are significant in Modula-2 reserved words.

+	=	AND	FOR	QUALIFIED
-	#	ARRAY	FROM	RECORD
*	<	BEGIN	IF	REPEAT
/	>	BY	IMPLEMENTATION	RETURN
:=	<>	CASE	IMPORT	SET
&	<=	CONST	IN	THEN
.	>=	DEFINITION	LOOP	TO
,	..	DIV	MOD	TYPE
;	:	DO	MODULE	UNTIL
(	)	ELSE	NOT	VAR
[	]	ELSIF	OF	WHILE
{	}	END	OR	WITH
^	|	EXIT	POINTER	
'	"	EXPORT	PROCEDURE	

Appendix 2 Standard Identifiers

ABS	EXCL	NIL
BITSET	FALSE	ODD
BOOLEAN	FLOAT	ORD
CAP	HALT	PROC
CARDINAL	HIGH	REAL
CHAR	INC	TRUE
CHR	INCL	TRUNC
DEC	INTEGER	VAL
DISPOSE	NEW	

Appendix 3 ASCII Character Set

0	000	00	nul	32	040	20		64	100	40	@	96	140	60	`
1	001	01	soh	33	041	21	!	65	101	41	A	97	141	61	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	$	68	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	065	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	037	1F	us	63	077	3F	?	95	137	5F	_	127	177	7F	del

Index

Modula-2

Standard Library

Release:	0.3
Date:	26 August 1983
Author:	Richard Gleaves

Table Of Contents

3 Module Hierarchy

This section describes relationships between the standard library modules.

Some library modules modules are entirely self-contained, but most import facilities from other library modules. Such dependence relations form a hierarchy of library modules, with self-contained modules appearing at the bottom and highly dependent modules appearing at the top.

Intermodule dependencies are obscured by the fact that library modules are more commonly imported in (hidden) implementation modules than in (visible) definition modules. Module dependencies are documented not so much because it is necessary to understand how a module works, but because indirectly imported modules affect the amount of storage left for a program. Importing a module at the top of the hierarchy offers high-level facilities at the expense of the modules used to implement it; importing a module at the bottom of the hierarchy provides cruder facilities, but saves memory by reducing the number of resident modules.

The choice of where to enter the module hierarchy depends on the relative importance of portability and efficiency. Higher level modules stress portability and ease of use over efficiency and generality; lower level modules, the reverse. For instance, the module InOut is easier to use than Texts, as it hides all details of the text streams it writes to. Texts in turn offers generality (the ability to define new text streams) and efficiency (InOut is implemented in terms of Texts). The complete module hierarchy is described in the **Implementation Guide.**

Standard library module dependencies:

- **RealInOut** -> InOut, Reals
- **InOut** -> Conversions, Texts
- **Reals** -> Texts, Program
- **Texts** -> Conversions, Files, Storage, Program
- **Files** -> Storage, Program
- **Processes** -> Storage, Program
- **Storage** -> Program
- **Program** -> Storage

4 InOut

The module InOut is implemented as defined in **Programming in Modula-2.** InOut provides operations for reading and writing basic data types to the standard input and output text streams. Note that details pertaining to the actual streams are suppressed, allowing this module to be portable across all Modula-2 systems. Standard input and output defaults to the console, but can be redirected to other files.

The procedures Read, ReadString, ReadInt, and ReadCard read data from the input stream. Read returns a single character; if the character equals EOL, the end of a text line has been read.

NOTE- The value of EOL is system-dependent.

```
REPEAT
  InOut.Read(ch);
  line[inx] := ch;
  INC(inx);
UNTIL ch = EOL;
```

ReadString reads a string of (non-blank) characters. ReadString skips leading blanks and terminates on either a blank or control character; the terminating character is returned in the variable termCH. ReadInt reads a string and converts it to an integer. ReadCard reads a string and converts it to a cardinal number. When reading from the console, backspace deletes the last character typed in.

```
ReadInt(i);
ReadString(s);

ReadCard(c);
IF termCH = EOL THEN WriteLn END;
```

The end of an input stream is recognized by checking the variable Done. Done is set after every read operation: TRUE indicates that the preceding operation was successfully completed; FALSE indicates that the previous operation failed (either because the end of the stream was reached or because of an error). Read returns a null character (0C) if the end of the stream has been reached.

```
LOOP
  InOut.Read(ch);
  IF InOut.Done THEN EXIT END;
  process(ch);
END;
```

The procedures Write, WriteString, WriteInt, WriteCard, WriteOct, and WriteHex write data to the output stream. WriteLn writes a line terminator. WriteHex and WriteOct write out fixed numbers of digits: i.e. four hexadecimal digits and six octal digits respectively. The argument "n" in most of these routines is for output formatting. If fewer than n characters are required to write out the data value, leading blank characters are written to pad the result out to n characters.

NOTE- WriteLn is defined to be equivalent to Write(EOL).

```
Write(ch);
WriteString("Hi there! The answer is: ");
WriteCard(sum, 7);
```

Standard I/O is directed to the console, but can be redirected to disk files (or other devices). OpenInput and OpenOutput both issue console prompts requesting the name of the file to redirect I/O to. If a file name ends with a period, the default extension "defext" is appended to the file name before the file is opened. (Note that this convention is the exact opposite of some other file systems.)

If the specified file is successfully opened, it becomes the source (sink) of standard I/O. OpenInput and OpenOutput return status results in the variable Done; Done is set to TRUE if the file was successfully opened.

```
OpenInput("TEXT");
IF NOT Done THEN WriteString("File open error") END;
```

Redirected standard I/O is returned to the console by calling CloseInput or CloseOutput. Redirected files are closed. Done is set to TRUE if the redirected file was successfully closed.

NOTE- InOut uses the predefined text variables Texts.input and Texts.output as the (default) standard input and output streams. If these variables are modified by a program importing Texts, the operation of InOut is affected.

Example of InOut:

```
MODULE SumLines;

(* Sum each line in the input file *)

  FROM InOut IMPORT
    OpenInput, CloseInput,
    EOL, termCH, Done, WriteLn,
    ReadInt, WriteInt, WriteString;

VAR i, sum: INTEGER;

BEGIN
  OpenInput("TEXT");
  IF NOT Done THEN
    WriteString("File not opened");
    HALT;
  END;

  sum := 0;
  ReadInt(i);
  WHILE Done DO
    INC(sum, i);
    IF termCH = EOL THEN
      WriteInt(sum, 7);
      WriteLn;
      sum := 0;
    END;
    ReadInt(i);
  END;

  CloseInput;
END SumLines.
```

```
DEFINITION MODULE InOut;

EXPORT QUALIFIED
  EOL, Done, termCH,
  OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadInt, ReadCard,
  Write, WriteLn, WriteString, WriteInt, WriteCard, WriteOct, WriteHex;

CONST EOL =  15C;  (* system dependent *)

VAR Done: BOOLEAN;
VAR termCH: CHAR;

PROCEDURE OpenInput  (defext: ARRAY OF CHAR);
PROCEDURE OpenOutput (defext: ARRAY OF CHAR);
PROCEDURE CloseInput;
PROCEDURE CloseOutput;

PROCEDURE Read       (VAR ch: CHAR);
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
PROCEDURE ReadInt    (VAR x: INTEGER);
PROCEDURE ReadCard   (VAR x: CARDINAL);

PROCEDURE Write       (ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE WriteInt    (x: INTEGER; n: CARDINAL);
PROCEDURE WriteCard   (x,n: CARDINAL);
PROCEDURE WriteOct    (x,n: CARDINAL);
PROCEDURE WriteHex    (x,n: CARDINAL);

END InOut.
```

5 RealInOut

The module RealInOut reads and writes real numbers to the standard input and output streams. The parameter "n" in WriteReal is used for output formatting (see InOut for details). WriteReal displays real numbers in exponent notation. WriteRealOct displays real numbers in internal format; the variable contents are written as multi-word octal values. The variable Done is set after every call to ReadReal; it indicates whether the previous read operation was successfully completed (Done = TRUE indicates a real number was successfully read).

NOTE- RealInOut accesses streams via InOut's procedures ReadString and WriteString; therefore, redirecting I/O in InOut affects the operation of RealInOut. Note that InOut and RealInOut export separate Done variables.

```
DEFINITION MODULE RealInOut;

EXPORT QUALIFIED
  ReadReal, WriteReal, WriteRealOct, Done;

VAR Done: BOOLEAN;

PROCEDURE ReadReal(VAR x: REAL);
PROCEDURE WriteReal(x: REAL; n: CARDINAL);
PROCEDURE WriteRealOct (x: REAL);

END RealInOut.
```

6 Texts

The module Texts provides operations for reading and writing basic data types to text streams. The model for input text streams is a sequence of characters structured into lines; this model is implemented as a sequence of character strings terminated by null characters (0C). Note that null characters serve as both line and stream terminators; line separation and the end of an input stream are indicated by the procedures EOL and EOT.

Control characters are nonprintable ASCII characters (other than nulls) that are not interpreted by the underlying implementation. Interpreted control characters do not themselves appear in a text stream. An example of an interpreted control character is a carriage return read from the console; it is translated into a (line-terminating) null character.

Text stream I/O is performed through variables declared with type TEXT.

```
VAR listing, errors: TEXT;
```

The exported text variables **input** and **output** are connected to the system terminal and represent the standard input and output streams. Programs performing only standard I/O do so through input and output. The text variable **console** is connected to the console; it is used for writing console messages (in case the standard text streams have been redirected).

```
MODULE ZZ;
FROM Texts IMPORT output, WriteString;
BEGIN
  WriteString(output, "Hi there!");
END ZZ.
```

The procedures Connect and Disconnect are used to open and close text variables for text I/O operations. Text streams do not directly access external files; instead, they are "connected" to file variables which in turn have already been opened. Connect associates a text stream with an existing (open) file variable. (Note that Connect does not affect the file state.) Disconnect disassociates a text stream from its file variable. Connect and Disconnect return a value (of type TextState) indicating the result of the operation. Text I/O cannot be performed on an unconnected (or disconnected) text stream.

```
IF Connect(listing, listfile) # TextOK THEN HALT END;
```

Read returns every character in a text stream (including nulls and control characters). ReadInt and ReadCard skip leading blanks and control characters and terminate after reading a non-digit character. ReadLn reads the rest of a text line. EOL always returns TRUE after calls to ReadLn. ReadAgain causes the last character read to be read again by the following read operation. When reading a text stream from the console, typing a backspace deletes the last character typed in.

```
ReadCard(input, c);

LOOP
  ReadLn(t, s);
  IF EOT(t) THEN EXIT END;
  WriteString(listing, s);
  WriteLn(listing);
END;
```

EOL and EOT are set after every read operation. EOL returns TRUE if the line-terminating null character was read or if EOT is TRUE. EOT becomes TRUE if the previous operation failed (either because of an error or because the end of the text stream was reached). Note that EOT is set to TRUE if any operation returns a result value other than TextOK.

NOTE- On calls to ReadInt and ReadCard, EOL is set to TRUE only if the end-of-line marker is the terminating character. If a line of numbers contains trailing blanks, EOL is not set to TRUE after reading the last number on the line.

```
LOOP
  LOOP Read(t, ch);
    IF EOL(t) THEN EXIT END;
    Process(ch);
  END;
  IF EOT(t) THEN EXIT END;
  ProcessLine;
END;
```

TextStatus returns a value (of type TextState) indicating the status of the specified text variable; in particular, the result of the last text stream operation. TextStatus returns an undefined value for text streams which have not been connected. (Note that EOL and EOT are both set to TRUE if any operation returns a text result other than TextOK.)

Text results have the following meanings:

TextOK	- The last operation was successful.
FileError	- Error in underlying file operation.
FormatError	- Invalid data format.
ConnectError	- Invalid operation on (un)connected text stream.

The parameter 'n' in the numerical write operations (WriteInt, WriteCard) is used for output formatting. If a numerical string contains fewer characters than are specified by n, it is preceded by enough blank characters to make the resulting output n characters long. If n specifies fewer characters than are in the numerical string, it is ignored.

NOTE- The file positioning operations defined in Files can be applied to files connected to text streams, allowing random access of text streams; however, file positions are restricted to line boundaries (i.e. when EOL(t) = TRUE). The effect of positioning the file within a text line is not defined.

SetTextHandler allows error handling procedures to be bound to specific text variables; if a text operation sets TextStatus to a value other than TextOK, the associated procedure is automatically invoked. Error handlers are useful when large numbers of operations are performed on a text variable; they eliminate the need for explicit error checking code after every text operation. Note that the handling procedure's parameter list must be compatible with type TextHandler; the text parameter informs the handler of the text result causing the error. Text handlers can only be set on open text variables. Connect and Disconnect do not invoke text handlers.

NOTE- Texts automatically disconnects any text streams left connected by a program (on return from "unshared" subprogram calls — see the module Program for details).

WARNING- Handler procedures should limit their operations to calling Disconnect and/or writing error messages, as further operations on the erroneous text stream may reinvoke the handler. Also, subprograms should not install local handler procedures in text variables declared outside the subprogram; the system may crash if Texts attempts to invoke a handler procedure which is no longer memory resident (because its host program has terminated).

Examples of Texts:

```
MODULE AddingMachine;

FROM Texts IMPORT
  input, output, ReadInt, WriteInt, WriteString, WriteLn;

VAR i1, i2: INTEGER;

BEGIN
  LOOP
    WriteString(output, "a: ");
    ReadInt(input, i1);

    WriteString(output, "b: ");
    ReadInt(input, i2);

    WriteString(output, "a+b = ");
    WriteInt(output, i1 + i2, 1);
    WriteLn(output);
    IF (i1 = 0) & (i2 = 0) THEN EXIT END;
  END;
END AddingMachine.
```

```
MODULE AddResults; (* sum each line of integers in f *)

FROM Files IMPORT
  FILE, Open, Create, Close, FileOK, SetFileHandler;

FROM Texts IMPORT
  console, output, TEXT, Connect, Disconnect, EOT, EOL, TextState,
  SetTextHandler, ReadInt, WriteInt, WriteLn, WriteString;

VAR f: FILE;
    t: TEXT;
    i, sum: INTEGER;

PROCEDURE handler (error: TextState);
BEGIN WriteString(console, "Text error");
  HALT;
END handler;

PROCEDURE IOError;
BEGIN WriteString(console, "I/O error");
  HALT;
END IOError;

BEGIN
  IF Open(f,"ints.text") # FileOK THEN IOError END;
  IF Connect(t, f) # TextOK THEN IOError END;
  SetTextHandler(t, handler);

  LOOP ReadInt(t, i);
    IF EOT(t) THEN EXIT END;

    sum := 0;
    LOOP INC(sum, i);
      IF EOL(t) THEN EXIT END;
      ReadInt(t, i);
    END;

    WriteInt(output, sum, 0);
    WriteLn(output);
  END;

  IF Disconnect(t) # TextOK THEN IOError END;
  IF Close(f) # FileOK THEN IOError END;
END AddResults.
```

```
DEFINITION MODULE Texts;

FROM Files IMPORT FILE;

EXPORT QUALIFIED
  TEXT, input, output, console, Connect, Disconnect,
  EOT, EOL, TextStatus, TextState, SetTextHandler,
  Read, ReadInt, ReadCard, ReadLn, ReadAgain,
  Write, WriteString, WriteInt, WriteCard, WriteLn;

TYPE TEXT;

VAR input, output, console: TEXT;      (* Predeclared text files *)

PROCEDURE EOT (t: TEXT): BOOLEAN;      (* End of text read *)
PROCEDURE EOL (t: TEXT): BOOLEAN;      (* End of line read *)

TYPE TextState = (TextOK, FormatError, FileError, ConnectError);

PROCEDURE TextStatus (t: TEXT): TextState;

TYPE TextHandler = PROCEDURE (TextState);

PROCEDURE SetTextHandler (t: TEXT; handler: TextHandler);

PROCEDURE Connect    (VAR t: TEXT; f: FILE): TextState;
PROCEDURE Disconnect (VAR t: TEXT): TextState;

PROCEDURE Read       (t: TEXT; VAR ch: CHAR);
PROCEDURE ReadInt    (t: TEXT; VAR i: INTEGER);
PROCEDURE ReadCard   (t: TEXT; VAR c: CARDINAL);
PROCEDURE ReadLn     (t: TEXT; VAR s: ARRAY OF CHAR);
PROCEDURE ReadAgain  (t: TEXT);

PROCEDURE Write      (t: TEXT; ch: CHAR);
PROCEDURE WriteString (t: TEXT; s: ARRAY OF CHAR);
PROCEDURE WriteInt   (t: TEXT; i: INTEGER; n: CARDINAL);
PROCEDURE WriteCard  (t: TEXT; c, n: CARDINAL);
PROCEDURE WriteLn    (t: TEXT);

END Texts.
```

7 Reals

The module Reals provides I/O and conversion routines for floating point numbers. The procedures RealToStr and StrToReal convert real numbers between character and internal representations; they return TRUE after successful conversions. Strings passed to StrToReal cannot have any leading or trailing blanks.

The parameter 'n' in WriteReal is used for output formatting (see Texts for details). The parameter 'digits' in WriteReal and RealToStr determines whether the number is to be displayed in fixed point or exponent notation.

'digits' < 0 specifies exponent notation, with ABS(digits) fractional digits displayed. A mantissa sign appears only if the mantissa is negative. The number of fractional digits displayed is not constrained by the number of significant digits in the underlying implementation. The exponent part always appears as the letter 'E' followed by an exponent sign (either '+' or '-') and the exponent digits. The number of exponent digits is fixed; therefore, exponent values may contain leading zeroes.

'digits' >= 0 specifies fixed point notation, with 'digits' fractional digits displayed. A sign character appears only if the number is negative. Specifying zero fractional digits suppresses the display of the decimal point. The number of fractional digits displayed is not constrained by the number of significant digits in the underlying implementation.

```
DEFINITION MODULE Reals;

FROM Texts IMPORT TEXT;

EXPORT QUALIFIED RealToStr, StrToReal, ReadReal, WriteReal;

PROCEDURE ReadReal   (t: TEXT; VAR r: REAL);

PROCEDURE WriteReal  (t: TEXT; r: REAL;
                      n: CARDINAL; digits: INTEGER);

PROCEDURE RealToStr  (r: REAL; digits: INTEGER;
                      VAR s: ARRAY OF CHAR): BOOLEAN;

PROCEDURE StrToReal  (s: ARRAY OF CHAR;
                      VAR r: REAL): BOOLEAN;

END Reals.
```

8 Files

The module Files implements data files. Both random and sequential ("stream") file access are supported.

The logical model of a file is a sequence of bytes with a current position (the next byte in the sequence to be accessed) and an end position (the position past the last byte in the sequence).

Files are accessible as byte streams or as sequences of word-oriented records. Files also provides access to the underlying file system; operations are provided for connecting files to external files and for renaming and deleting external files.

File I/O is performed through variables declared with type FILE.

```
VAR source, code: FILE;
```

EOF is set after every read operation. EOF returns TRUE if the previous operation failed (either because of an I/O error or because the file position was at the end of the file). Note that EOF is set to TRUE if any operation returns a result value other than FileOK.

FileStatus returns a value (of type FileState) indicating the status of the specified file; in particular, the result of the last file operation.

The file results have the following meanings:

FileOK	- The last operation was successful.
NameError	- Specified external file was not available.
UseError	- Invalid external file operation.
StatusError	- Attempt to access a closed file.
DeviceError	- Error in underlying I/O system.
EndError	- File position exceeds end of file.

Open connects a file to an existing external file. Create creates a new external file and connects it to the file. Close disconnects a file, preserving the external file. Release disconnects a file and deletes the external file. Open, Create, Close, and Release return a value (of type FileState) indicating the result of the operation. File I/O cannot be performed on unopened (closed) files.

```
IF Open(f, 'accounting.data') # FileOK THEN
  WriteString('File not opened'); HALT
END;
```

Rename changes the name of an existing external file. Rename returns a value indicating the result of the operation; 'FileOK' indicates that the file was successfully renamed. If the file's new name matches the name of another file on the volume, that file is deleted. The specified external file must not be open.

Delete removes an existing external file from the directory. Delete returns a value indicating the result of the operation; 'FileOK' indicates that the file was successfully deleted. The specified external file must not be open.

Read reads a character from the file. EOF returns TRUE after a read operation is attempted at the end-of-file position.

ReadRec reads a word-oriented record from the file. EOF returns TRUE when a read operation is attempted at the end-of-file position. If ReadRec attempts to read more data than is available at the end of a file, the contents of the input variable are undefined and FileStatus is set to EndError.

ReadBytes reads a stream of bytes from the file and returns the number of bytes actually read. EOF is set to true if the number of bytes read is less than the number of bytes specified.

Write writes a character to the file. WriteRec writes a word-oriented record to the file. WriteBytes writes a stream of bytes to the file and returns the number of bytes written. If the amount of data actually written to a file is less than the amount specified, FileStatus is set to DeviceError.

NOTE- All write operations may overwrite existing data in a file. A file can be extended only by appending data to the immediate end-of-file position (i.e. the file position returned by GetEOF).

```
LOOP
  Read(infile, ch);
  IF EOF(infile) THEN EXIT END;
  Write(outfile, ch);
END;
```

SetPos sets the current file position to the specified value. The file position cannot be set past the current end of the file; attempts to do so cause FileStatus to return EndError. GetPos returns the current file position.

SetEOF sets the end file position to the specified value. The end file position indicates the file position of the byte following the last byte in a file. The end file position cannot be set in front of the current file position or past the current end file position; attempts to do so cause FileStatus to return EndError. GetEOF returns the end file position.

```
GetEOF(F, endpos);
SetPos(F, endpos);
WriteByte(F, 0C);
```

NOTE- FileStatus returns UseError if the file positioning operators (GetEOF, SetEOF, GetPos, SetPos) are called on files connected to serial (i.e. nondisk) files.

File position values are stored in variables of type FilePos.

```
VAR startpos, endpos: FilePos;
```

The procedure CalcPos computes absolute file positions; it translates a record number and record size into an absolute file position. Record sizes are defined in terms of the storage unit of the underlying machine; this convention is compatible with the values returned by the system-defined procedures SIZE and TSIZE. The first record in a file is defined as record 0.

```
CalcPos(blknum, TSIZE(block), startpos);
```

NOTE- Values stored in variables of type FilePos are implementation-dependent. File positions are intended for use as abstract file markers (i.e. where arguments to SetPos are obtained from GetPos, GetEOF, or CalcPos).

SetFileHandler allows error-handling procedures to be bound to specific file variables; if a file operation sets FileStatus to a value other than FileOK, the associated procedure is automatically invoked. Error handlers are useful when large numbers of operations are performed on a file variable; they eliminate the need for explicit error-checking code after every file operation. Note that the handling procedure's parameter list must be compatible with type FileHandler; the file result parameter allows the handler to indicate the file result causing the error. File handlers can only be set on open file variables. File handlers are not invoked by Open, Close, Create, or Release.

NOTE- Files automatically closes any files left open by a program (on 'unshared' subprogram calls — see the module Program for details).

WARNING- Handler procedures should limit their operations to closing the file or writing error messages, as further operations on the erroneous file may reinvoke the handler. Also, subprograms should not install local handler procedures in file variables outside the subprogram; the system may crash if Files attempts to invoke a handler procedure which is no longer memory resident (because its host subprogram has terminated).

```
MODULE FileCopy;

FROM Files IMPORT
  FILE, Open, Create, Close, FileStatus, FileState,
  SetFileHandler, Read, Write, EOF;

FROM Terminal IMPORT ReadLn, WriteString, WriteLn;

PROCEDURE handler(error: FileState);
BEGIN
  WriteString('File I/O error');
  HALT;
END handler;

TYPE FProc = PROCEDURE(VAR FILE, ARRAY OF CHAR): FileState;

PROCEDURE FileOpen(VAR f: FILE; fcall: FProc; s: ARRAY OF CHAR);
VAR name: ARRAY [0..20] OF CHAR;
BEGIN
  LOOP WriteString(s);
    ReadLn(name);
    IF fcall(f, name) = FileOK THEN EXIT END;
    WriteString("Can't open ");
    WriteString(name); WriteLn;
  END;
  SetFileHandler(f, handler);
END FileOpen;

PROCEDURE FileClose(VAR f: FILE);
BEGIN
  IF Close(f) # FileOK THEN
    WriteString('Error closing file');
    HALT;
  END;
END FileClose;

VAR infile, outfile: FILE;
     ch: CHAR;

BEGIN
  FileOpen(infile, Open, 'Input file? ');
  FileOpen(outfile, Create, 'Output file? ');
  LOOP Read(infile, ch);
    IF EOF(infile) THEN EXIT END;
    Write(outfile, ch);
  END;
  FileClose(infile);
  FileClose(outfile);
  WriteString('file copy complete');
END FileCopy.
```

```
DEFINITION MODULE Files;

FROM SYSTEM IMPORT WORD, ADDRESS;

EXPORT QUALIFIED
  FILE, EOF, FileStatus, FileState, SetFileHandler,
  Open, Create, Close, Release, Rename, Delete,
  FilePos, SetPos, GetPos, SetEOF, GetEOF, CalcPos,
  Read, Write, ReadRec, WriteRec, ReadBytes, WriteBytes;

TYPE FILE;

PROCEDURE EOF (f: FILE): BOOLEAN;   (* End of file encountered *)

TYPE FileState = (FileOK, NameError, UseError, StatusError, DeviceError, EndError);

PROCEDURE FileStatus (f: FILE): FileState;   (* file I/O status *)

TYPE FileHandler = PROCEDURE (FileState);

PROCEDURE SetFileHandler (f: FILE; handler: FileHandler);

PROCEDURE Open    (VAR f: FILE; name: ARRAY OF CHAR): FileState;
PROCEDURE Create  (VAR f: FILE; name: ARRAY OF CHAR): FileState;

PROCEDURE Close   (VAR f: FILE): FileState;
PROCEDURE Release (VAR f: FILE): FileState;

PROCEDURE Delete  (name: ARRAY OF CHAR): FileState;
PROCEDURE Rename  (old, new: ARRAY OF CHAR): FileState;

TYPE FilePos;

PROCEDURE GetPos  (f: FILE; VAR pos: FilePos);
PROCEDURE GetEOF  (f: FILE; VAR pos: FilePos);

PROCEDURE SetPos  (f: FILE; pos: FilePos);
PROCEDURE SetEOF  (f: FILE; pos: FilePos);

PROCEDURE CalcPos (recnum, recsize: CARDINAL; VAR pos: FilePos);

PROCEDURE Read       (f: FILE; VAR ch: CHAR);
PROCEDURE ReadRec    (f: FILE; VAR rec: ARRAY OF WORD);
PROCEDURE ReadBytes  (f: FILE; buf: ADDRESS; nbytes: CARDINAL): CARDINAL;

PROCEDURE Write      (f: FILE; ch: CHAR);
PROCEDURE WriteRec   (f: FILE; VAR rec: ARRAY OF WORD);
PROCEDURE WriteBytes (f: FILE; buf: ADDRESS; nbytes: CARDINAL): CARDINAL;

END Files.
```

9 Terminal

The standard module Terminal provides basic routines for reading characters from the keyboard and writing characters to the screen.

Read waits for a character to be typed; the character is echoed to the console when it is read.

BusyRead immediately returns a null character (0C) if a character has not been typed. Characters are not echoed when they are read.

ReadAgain places the last character read back into the buffer so it can be subsequently re-read.

ReadLn reads characters until a carriage return is typed. Characters are echoed to the screen as they are read. Typing backspace deletes the last character typed in. The carriage return is read, but is not returned in the string parameter.

```
DEFINITION MODULE Terminal;

EXPORT QUALIFIED  Read, BusyRead, ReadAgain, ReadLn,
                  Write, WriteString, WriteLn;

PROCEDURE Read        (VAR ch: CHAR);
PROCEDURE ReadLn      (VAR s: ARRAY OF CHAR);
PROCEDURE BusyRead    (VAR ch: CHAR);
PROCEDURE ReadAgain;

PROCEDURE Write       (ch: CHAR);
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE WriteLn;

END Terminal.
```

10 Storage

The module Storage provides dynamic storage allocation and deallocation.

ALLOCATE allocates a storage area containing 'size' storage units (as returned by SIZE and TSIZE) and returns the storage address in 'p'. DEALLOCATE deallocates the storage area specified by p and size and sets p to NIL. Available returns TRUE if a storage area of the indicated size can be allocated.

NOTE- Calls to the standard procedures NEW and DISPOSE require the identifiers ALLOCATE and DEALLOCATE to be visible.

NOTE- A program is terminated with StorageError if it attempts to a) allocate too large of a storage area or b) deallocate a storage area that has already been deallocated.

NOTE- On 'unshared' subprogram calls, all storage allocated by a subprogram is automatically deallocated when the subprogram terminates. (See the module Program for details.)

NOTE- Dynamic storage is always allocated in the system work space; processes are unable to allocate dynamic storage within their private work spaces.

```
DEFINITION MODULE Storage;

FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED ALLOCATE, DEALLOCATE, Available;

PROCEDURE ALLOCATE   (VAR p: ADDRESS; size: CARDINAL);

PROCEDURE DEALLOCATE (VAR p: ADDRESS; size: CARDINAL);

PROCEDURE Available  (size: CARDINAL): BOOLEAN;

END Storage.
```

11 Program

The module Program is used to perform subprogram calls. It also provides exception handling facilities and a mechanism allowing library modules to specify initialization and termination procedures which are automatically invoked on subprogram calls.

Call loads and executes the program module specified by the module identifier passed in the parameter 'programName'. Any library modules imported by the subprogram that are not already in memory are also loaded. When the subprogram terminates, the program module and all modules loaded by it are released from memory, and control returns to the calling program.

The Call parameter 'calltype' specifies whether the subprogram shares its dynamic storage with the calling program. Setting calltype to 'Unshared' causes all storage allocated by the subprogram to be automatically deallocated when the subprogram terminates. Setting calltype to 'Shared' retains any dynamic storage left behind by the subprogram.

The Call parameter 'errors' specifies whether execution errors generated by the called subprogram are to be acted upon by the system or merely returned as a result value to the calling program. Setting errors to 'SystemTrap' causes execution errors to invoke a system-defined error handler (usually an execution error message or debugger — see **The Modula-2 System** for details). Setting errors to 'CallerTrap' returns control directly to the calling program. Note that in either case control is defined to eventually return to the calling program.

Call returns a value (of type CallResult) indicating the execution result of the called subprogram; this value reflects either a program load error, an execution error, or normal program termination.

NOTE- **The Modula-2 System** explains how the loader accesses the library.

NOTE- Subprograms can be called only from the main process.

The following example shows how Program can be used to write a simple "shell" program which emulates the UCSD Pascal system:

```
MODULE Shell;

FROM Program IMPORT Call, CallResult, Unshared, SystemTrap;
FROM Terminal IMPORT ReadLn, WriteString, WriteLn, Read;
FROM Screen IMPORT ClearScreen, GotoXY;

VAR ch: CHAR;
     rslt: CallResult;
     name: ARRAY [0..40] OF CHAR;

BEGIN
  ClearScreen;
  LOOP
    GotoXY(0, 0);
    WriteString("Command: X(ecute, F(iler, E(ditor, H(alt [0.3] ");
    Read(ch);
    ClearScreen;
    ch := CAP(ch);
    IF    ch = "H" THEN RETURN
    ELSIF ch = "F" THEN name := "SYSTEM.FILER."
    ELSIF ch = "E" THEN name := "SYSTEM.EDITOR."
    ELSIF ch = "X" THEN
      WriteString("Execute what file? ");
      ReadLn(name);
    ELSE
      name[0] := 0C;
    END;
    IF name[0] # 0C THEN
      rslt := Call(name, Unshared, SystemTrap);
    END;
  END;
END Shell.
```

Program results have the following meanings:

NormalReturn	- Program terminated normally.
ProgramHalt	- Program executed 'HALT'.
RangeError	- Value range error.
SystemError	- Invalid code structure.
FunctionError	- Function did not execute 'RETURN'.
StackOverflow	- System stack exceeded.
IntegerError	- Integer overflow.
DivideByZero	- Divide by zero.
AddressError	- Invalid address reference.
UserHalt	- Program terminated by user.
CodeIOError	- System I/O error; code not loaded.
UserIOError	- User I/O error raised by program.
InstructionError	- Unimplemented instruction.
FloatingError	- Floating point arithmetic error.
StringError	- String overflow or invalid index.
StorageError	- Dynamic storage exhausted.
VersionError	- Module version error.
MissingProgram	- Subprogram not found.
MissingModule	- Library module not found.
LibraryError	- Incorrect library structure.
NotMainProcess	- Attempted Program call by process.
DuplicateName	- Duplicate library module names.

NOTE- Programs are responsible for monitoring the result values returned by subprogram calls and acting accordingly in the event of an error.

Execution errors are usually generated by the system; however, programs can terminate themselves with an error by calling Terminate with the appropriate error value as an argument:

```
PROCEDURE StopProgram(cause: CallResult);
BEGIN
  WriteString("Fatal error #");
  WriteCard(ORD(cause), 0);
  Terminate(cause);
END StopProgram;
```

SetEnvelope allows library modules to define initialization and termination procedures which are automatically executed before and after subprogram calls; these procedure pairs are called **envelopes** because of the way they "surround" subprograms. Envelopes are used to manage system resources that must be started up and shut down independently of their use by programs. For instance, the module Files uses envelopes to close files accidentally left open by subprograms.

A library module establishes an envelope by calling SetEnvelope in its outer block, passing two of its own procedures as arguments. The parameter 'init' indicates the initialization procedure, the parameter 'term' the termination procedure.

The value passed to the parameter 'mode' specifies how often an envelope is invoked. 'FirstCall' invokes the envelope only once - around the subprogram that loads the library module. 'UnsharedCalls' invokes the envelope around nested "unshared" subprogram calls. 'AllCalls' invokes the envelope around all nested subprogram calls.

"unshared" envelopes are useful when the resource being managed requires dynamic storage; they allow shared subprograms to establish resources for use by the calling program. For instance, the module Files uses an "unshared" envelope to allow shared subprograms to open files used by the calling program.

On subprogram calls, initialization procedures are invoked after all new library modules have been loaded but before their outer blocks are executed. After a subprogram finishes, termination procedures are invoked before any library modules are released from memory.

When called, SetEnvelope immediately calls the passed initialization procedure. This ensures that library modules are initialized when they are first loaded into memory.

NOTE- On subprogram calls, the execution order of initialization procedures is determined by the order in which they are installed with SetEnvelope; the first procedure established is the first procedure executed. Termination procedures are executed in the reverse order of the initialization procedures.

WARNING- SetEnvelope should be called only from the outer blocks of library modules; calling it in arbitrary places in a program may crash the system. Execution errors within initialization/termination sections are returned as if they originated in the called subprogram.

In the following example, the library module MyTexts uses envelopes to keep track of the text streams created by a subprogram. If the subprogram neglects to disconnect a text variable, the termination procedure does so automatically. Note that texts are left connected on 'shared' calls.

```
IMPLEMENTATION MODULE MyTexts;
  ...

TYPE TEXT     =  POINTER TO RECORD
                   open: BOOLEAN;
                   next : TEXT;
                 END;

     TextMark =  POINTER TO MarkRec;
     MarkRec  =  RECORD
                   Texts: TEXT;
                   Prev : TextMark;
                 END;

VAR TopMark: TextMark;

PROCEDURE MarkText;
VAR p: TextMark;
BEGIN NEW(p);
  p^.Prev  := TopMark;
  p^.Texts := NIL;
  TopMark := p;
END MarkText;

PROCEDURE ReleaseText;
VAR p, q: TEXT;
     t: TextMark;
     r: TState;
BEGIN
  t := TopMark;
  p := t^.Texts;
  TopMark  := t^.Prev;
  DISPOSE(t);
  WHILE p # NIL DO
    q := p;
    IF p^.open THEN r := Disconnect(p) END;
    p := q^.next;
    DISPOSE(q);
  END;
END ReleaseText;

BEGIN
  SetEnvelope(MarkText, ReleaseText, UnsharedCalls);
END MyTexts.
```

```
DEFINITION MODULE Program;

EXPORT QUALIFIED
  Call, CallMode, ErrorMode, CallResult,
  Terminate, SetEnvelope, EnvMode;

TYPE CallResult = (NormalReturn, ProgramHalt, RangeError, SystemError,
                   FunctionError, StackOverflow, IntegerError,
                   DivideByZero, AddressError, UserHalt, CodeIOError,
                   UserIOError, InstructionError, FloatingError,
                   StringError, StorageError, VersionError,
                   MissingProgram, MissingModule, LibraryError,
                   NotMainProcess, DuplicateName);

TYPE CallMode  = (Shared, Unshared);
TYPE ErrorMode = (SystemTrap, CallerTrap);

PROCEDURE Terminate (exception: CallResult);

PROCEDURE Call (programName: ARRAY OF CHAR;
                calltype    : CallMode;
                errors      : ErrorMode): CallResult;

TYPE EnvMode = (AllCalls, UnsharedCalls, FirstCall);

PROCEDURE SetEnvelope (init, term: PROC; mode: EnvMode);

END Program.
```

12 Processes

Modula-2's coroutines are a low-level facility provided by the system module; most concurrent programs are expected to import process schedulers from the library. The standard utility module Processes implements the concept of **sequential processes.** Processes exports the identifiers SIGNAL, StartProcess, SEND, WAIT, Awaited, and Init.

The only difference between coroutines and sequential processes is the method used for scheduling the execution of individual processes. (Note that the restrictions on the use of coroutines apply equally to sequential processes.) Coroutine scheduling is performed explicitly; transfers occur only between named coroutines. Sequential process scheduling is left to the process scheduler; sequential processes synchronize their execution by operating on shared variables known as **signals.** Signals are declared as variables of type SIGNAL.

Signals serve the same purpose in sequential processes that process variables do in coroutines; they point to suspended processes, and are referenced whenever a process is suspended or resumed. The difference is that a process variable points to a single coroutine process, while a signal points to a first-in-first-out queue of sequential processes.

Sequential processes have three states: executing, ready, and suspended. As with coroutines, only one process executes at a time. A process is ready if it is stored in a special queue known as the ready queue, and suspended if stored in a signal's queue.

Sequential processes are created with the procedure StartProcess. StartProcess has the following syntax:

```
PROCEDURE StartProcess (P: PROC; n: CARDINAL);
```

P is the procedure which the new process will execute. P must be a parameterless procedure declared at the global (outermost) level in a compilation unit. n specifies the size of the work space in which the process will execute. (Note that a process variable is not specified.) When a sequential process is started, it is placed on the ready queue.

A process suspends itself on a signal's queue by calling WAIT. WAIT has the following syntax:

```
PROCEDURE WAIT (VAR s: SIGNAL);
```

s is the signal on which the process is to be suspended. The process scheduler adds the suspended process to the end of the signal queue, and selects a process from the ready queue for execution.

NOTE- Processes are selected from the ready queue on a first-in-first-out basis.

WARNING- If a process suspends itself on a signal and there are no processes on the ready queue, the program is halted. (This condition is called "deadlock".)

A process resumes the execution of a suspended process by calling SEND. SEND has the following syntax:

```
PROCEDURE SEND (VAR s: SIGNAL);
```

s is the signal from which a process is resumed. The process scheduler stops the calling process (placing it on the ready queue) and selects a process from the head of the signal queue for execution. If the signal named in SEND has an empty queue, the calling process continues to execute.

Before they are used, signals must be initialized with the procedure Init.

```
Init(S1);
```

The procedure Awaited indicates whether a signal queue is empty.

```
IF NOT Awaited(S1) THEN HALT END;
```

Unlike coroutines, sequential processes can be considered to execute in parallel. In place of direct transfers, sequential processes synchronize their execution by performing SEND and WAIT operations on shared signal variables. The following example is the sequential process analog of the coroutine program presented in **Introduction to Modula-2**; in this example, the processes compete (not necessarily fairly) for access to the console.

```
MODULE HiHo;

  FROM Processes IMPORT
    StartProcess, SIGNAL, SEND, WAIT, Init;

  IMPORT Terminal;

  MODULE Console[1];
    IMPORT Terminal, Init, SIGNAL, SEND, WAIT;
    EXPORT Write;

    CONST MaxHiHo = 17;
    VAR  n: CARDINAL;
         busy: BOOLEAN;
         free: SIGNAL;

    PROCEDURE Write(s: ARRAY OF CHAR);
    VAR i: CARDINAL;
    BEGIN
      IF busy THEN WAIT(free) END;
      busy := TRUE;
      FOR i := 0 TO HIGH(s) DO Terminal.Write(s[i]) END;
      INC(n);
      IF n > MaxHiHo THEN Terminal.WriteLn; n := 0 END;
      busy := FALSE;
      SEND(free);
    END Write;

  BEGIN n := 0; busy := FALSE;
    Init(free);
  END Console;

  PROCEDURE WriteHi;
  BEGIN LOOP Console.Write('Hi') END
  END WriteHi;

  PROCEDURE WriteHo;
  BEGIN LOOP Console.Write('Ho') END
  END WriteHo;

  VAR forever: SIGNAL;

BEGIN Init(forever);
  StartProcess(WriteHi, 200);
  StartProcess(WriteHo, 200);
  WAIT(forever);
END HiHo.
```

```
DEFINITION MODULE Processes;

EXPORT QUALIFIED SIGNAL, StartProcess, SEND, WAIT, Awaited, Init;

TYPE SIGNAL;

PROCEDURE StartProcess (P: PROC; n: CARDINAL);
  (*start a sequential process with program P
       and workspace of size n*)

PROCEDURE SEND (VAR s: SIGNAL);
  (*one process waiting for s is resumed*)

PROCEDURE WAIT (VAR s: SIGNAL);
  (*wait for some other process to send s*)

PROCEDURE Awaited (s: SIGNAL): BOOLEAN;
  (*Awaited(s) = 'at least one process waiting for s'*)

PROCEDURE Init (VAR s: SIGNAL);
  (*compulsory initialization*)

END Processes.
```

Appendix 1 Text & File Results

Texts

Text results have the following meanings:

0	TextOK	- The last operation was successful.
1	FileError	- Error in underlying file operation.
2	FormatError	- Invalid data format.
3	ConnectError	- Invalid operation on (un)connected text stream.

Files

File results have the following meanings:

0	FileOK	- The last operation was successful.
1	NameError	- Specified external file was not available.
2	UseError	- Invalid external file operation.
3	StatusError	- Attempt to access a closed file.
4	DeviceError	- Error in underlying I/O system.
5	EndError	- File position exceeds end of file.

Appendix 2 Program Results

Program results have the following meanings:

0	NormalReturn	- Program terminated normally.
1	ProgramHalt	- Program executed "HALT".
2	RangeError	- Value range error.
3	SystemError	- Invalid code structure.
4	FunctionError	- Function did not execute "RETURN".
5	StackOverflow	- System stack exceeded.
6	IntegerError	- Integer overflow.
7	DivideByZero	- Divide by zero.
8	AddressError	- Invalid address reference.
9	UserHalt	- Program terminated by user.
10	CodeIOError	- System I/O error; code not loaded.
11	UserIOError	- User I/O error raised by program.
12	InstructionError	- Unimplemented instruction.
13	FloatingError	- Floating point arithmetic error.
14	StringError	- String overflow or invalid index.
15	StorageError	- Dynamic storage exhausted.
16	VersionError	- Module version error.
17	MissingProgram	- Subprogram not found.
18	MissingModule	- Library module not found.
19	LibraryError	- Incorrect library structure.
20	NotMainProcess	- Attempted Program call by process.
21	DuplicateName	- Duplicate library module names.

Index

Modula-2

Utility Library

Release:	**0.3**
Date:	**26 August 1983**
Author:	**Richard Gleaves**

Table Of Contents

1 Introduction

This document describes the utility library. The utility library is a collection of modules which serve as an adjunct to the standard library; it is expected to grow in future releases of the Modula-2 system.

The utility library provides the following facilities:

- Numerical functions
- Decimal arithmetic
- String manipulation
- Format conversion
- ASCII control characters

2 Overview

The utility library contains the following modules:

- **MathLib0:** Mathematical functions — sqrt, exp, ln, sin, cos, arctan, entier.
- **Decimals:** Arithmetic operations for 19-digit decimal numbers. COBOL-style "picture" editing for formatting dollar quantities.
- **Strings:** String manipulation — Assign, Compare, Insert, Delete, Concat, Copy, Pos.
- **Conversions:** Format conversion between strings and numbers.
- **ASCII:** Symbolic character constants for ASCII control characters.

3 Module Hierarchy

This section describes dependencies between the utility library modules and other library modules. See **Standard Library** for more information on module dependencies.

Utility module dependencies:

- **MathLib0->** Program
- **Decimals->** Strings
- **Strings->** Program

4 MathLib0

The module MathLib0 provides basic mathematical functions. Arguments to trigonometric functions are in units of radians. 'real' converts its integer argument to a real number. 'entier' returns the largest integer that is less than or equal to the real argument.

If passed invalid arguments (e.g. square root of -1), MathLib0 halts the program with the program result FloatingError.

NOTE- The procedure 'real' may not be provided in some implementations. See **The Modula-2 System** for details.

```
DEFINITION MODULE MathLib0;

EXPORT QUALIFIED
  sqrt, exp, ln, sin, cos, arctan, real, entier;

PROCEDURE sqrt     (x: REAL): REAL;
PROCEDURE exp      (x: REAL): REAL;
PROCEDURE ln       (x: REAL): REAL;
PROCEDURE sin      (x: REAL): REAL;
PROCEDURE cos      (x: REAL): REAL;
PROCEDURE arctan   (x: REAL): REAL;
PROCEDURE real     (x: INTEGER): REAL;
PROCEDURE entier   (x: REAL): INTEGER;

END MathLib0.
```

5 Decimals

The module Decimals provides integer arithmetic and formatting routines suitable for business-oriented computation.

Decimal integers may contain up to 19 digits. Decimal variables are declared with type DECIMAL.

```
VAR Sales, Costs, Profit: DECIMAL;
```

The following procedures perform arithmetic operations:

```
AddDec (a,b)  ->  a + b
SubDec (a,b)  ->  a - b
MulDec (a,b)  ->  a * b
DivDec (a,b)  ->  a DIV b
NegDec (a)    ->  -a
```

These procedures accept decimal integers as arguments and return decimal integers as function results.

```
Profit := SubDec(Sales, Costs);
```

The Boolean variable DecValid is set after every arithmetic and conversion operation; its value indicates the result of the last operation. DecValid is set to FALSE if the previous operation failed.

```
FOR office := Bangor TO Bangkok DO
  Profit := AddDec(Profit, Net[office]);
END;
IF DecValid THEN HomeFree
ELSE CallTheAuditors END;
```

When an operation fails, the procedure DecStatus can be called to determine the actual arithmetic error. DecStatus returns a value (of type DecState) indicating the error status of the specified decimal variable: NegOvfl indicates negative overflow, PosOvfl positive overflow, and Invalid an invalid integer result.

DecStatus also indicates the sign of valid decimal integers. DecStatus returns a value indicating an integer's sign: Minus indicates a negative value, Plus a positive value, and Zero the value 0.

```
TYPE DecState = (NegOvfl, Minus, Zero, Plus, PosOvfl, Invalid);
```

Once a decimal variable assumes an erroneous state (e.g. NegOvfl), the error condition propagates through subsequent operations involving the variable. The following tables show how errors propagate through the arithmetic operations. For operations of the form "A <op> B", the leftmost column denotes states of A and the topmost row states of B.

Error propagation in addition and subtraction:

A \ B	**NegOvfl**	**Minus**	**Zero**	**Plus**	**PosOvfl**	**Invalid**
NegOvfl	NegOvfl	NegOvfl	NegOvfl	NegOvfl	Invalid	Invalid
Minus	NegOvfl				PosOvfl	Invalid
Zero	NegOvfl				PosOvfl	Invalid
Plus	NegOvfl				PosOvfl	Invalid
PosOvfl	Invalid	PosOvfl	PosOvfl	PosOvfl	PosOvfl	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Error propagation in multiplication:

A \ B	**NegOvfl**	**Minus**	**Zero**	**Plus**	**PosOvfl**	**Invalid**
NegOvfl	PosOvfl	PosOvfl	Zero	NegOvfl	NegOvfl	Invalid
Minus	PosOvfl	Plus	Zero	Minus	NegOvfl	Invalid
Zero	Zero	Zero	Zero	Zero	Zero	Invalid
Plus	NegOvfl	Minus	Zero	Plus	PosOvfl	Invalid
PosOvfl	NegOvfl	NegOvfl	Zero	PosOvfl	PosOvfl	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Error propagation in division:

A \ B	**NegOvfl**	**Minus**	**Zero**	**Plus**	**PosOvfl**	**Invalid**
NegOvfl	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Minus	Invalid	Plus	Invalid	Minus	Invalid	Invalid
Zero	Zero	Zero	Invalid	Zero	Zero	Invalid
Plus	Invalid	Minus	Invalid	Plus	Invalid	Invalid
PosOvfl	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Error propagation in negation:

A	**NegOvfl**	**Minus**	**Zero**	**Plus**	**PosOvfl**	**Invalid**
	PosOvfl	Plus	Zero	Minus	NegOvfl	Invalid

After a division operation, the character variable Remainder contains a remainder digit; i.e. the next lower digit of the quotient. For instance, dividing 33 by 7 returns the decimal value 4 and a remainder (in Remainder) of "5". If a division operation sets DecValid to FALSE, Remainder is assigned the character "?".

```
MilesPerHr := DivDec(TotalMiles, TotalHours);
IF Remainder >= "5" THEN
  MilesPerHr := AddDec(MilesPerHr, One);
END;
```

The procedure CompareDec compares two decimal integers and returns an INTEGER value indicating the comparison result: -1 if A is less than B; 0 if A equals B; 1 if A is greater than B.

```
IF CompareDec(Sales, Costs) < 0 THEN
  Blame(Economy);
END;
```

The procedure SetDecHandler allows error handlers to be bound to the decimal module. If any operation fails, the handler procedure is automatically invoked. Error handlers are most useful for trapping errors during series of arithmetic operations; handlers allow programs to perform decimal arithmetic (and format conversion) without having to check the variable DecValid after every operation.

```
PROCEDURE handler(error: DecState);
BEGIN
  WriteString("Decimal arithmetic error #");
  WriteCard(ORD(error), 3);
  HALT;
END handler;

  ...

SetDecHandler (handler);
```

5.1 Pictures

Decimal integers have two formats: internal and external. Arithmetic and logical operations are performed on integers stored in internal format. External format is used for reading and writing integers in human-readable form to the console or printer. The procedures StrToDec and DecToStr convert integers between internal and external format.

Decimal integers in external format are stored in character strings. In external format, an integer may contain a dollar sign, commas to separate thousands of dollars, and a decimal point separating dollars from cents.

Here is a decimal integer in external format:

```
$923,841,371.38
```

External format is controlled by string parameters known as **pictures.** Pictures serve as masks indicating how decimal integers should appear in external format; they control the inclusion of such things as leading zeros, signs, and decimal points.

For instance, the picture used to print the decimal integer shown above is:

```
$,$$$,$$$,$$$,$$9.99
```

Without the picture, it would have appeared as:

```
92384137138
```

When converting internal to external format, pictures add the appropriate punctuation characters into the integer. Pictures may contain only the characters '9', 'Z', '$', 'S', ',', or '.'; in particular, blanks may not appear in a picture.

9 -	digit
Z -	nonzero digit or leading blank
$ -	nonzero digit, leading blank, or '$'
S -	sign: '+' or '-'
. -	decimal point
, -	comma or leading blank

Dollar signs ('$') are used to denote the digits of integers displayed as dollar amounts. Dollar amounts are displayed with a currency character and no leading zeros. The currency character floats across any leading blanks so that it appears adjacent to the leftmost digit.

NOTE- If a decimal value contains as many digits as its corresponding picture, no currency character is displayed, as each dollar sign character is replaced by a digit. (This is best avoided by specifying big pictures!)

Integers displayed without leading zeros represent their digits with 'Z's. A 'Z' is replaced by a digit if there is one; otherwise, it is replaced by a blank.

Integers that require leading zeros to be displayed represent their digits with '9's. A '9' is replaced by a digit if there is one; otherwise, it is replaced by a '0'.

The picture characters '$', 'Z', and '9' can be mixed together in a single picture to obtain the desired integer format. In the following picture, the '9's guarantee that small dollar amounts appear in standard form:

```
$$$,$$9.99
```

Here are some integers produced by this picture:

```
    $0.39
  $369.00
$48,327.04
```

A period '.' is replaced by a decimal point. 'S' prints a sign character: either '+' or '-'. Note that sign characters and decimal points do not float across leading blanks; they appear in their specified position. Commas are used to separate integers into the traditional groups of three digits; like 'Z' and '$' digits, commas are transformed into leading blanks when they appear to the left of an integer.

Pictures not only control integer formatting, but place range constraints on integer values. If an integer value exceeds its picture (i.e. the number of digits exceeds the integer of digit characters in the picture), DecToStr sets DecValid to FALSE, and returns an 'invalid' format string (see below). Thus, pictures can be used to control the maximum number of digits that can appear in an integer.

DecToStr displays erroneous decimal values as distinctively formatted strings. Error string length is determined by the length of the corresponding picture.

```
PosOvfl    ->   "+++++++"
NegOvfl    ->   "-------"
Invalid    ->   "???????"
```

5.2 Input Pictures

The picture formatting described so far has been limited to converting decimal integers to their external (string) format. Pictures can also be used to control an integer's input format; the procedure StrToDec uses pictures to convert integers from external to internal format.

Pictures in StrToDec work almost identically to those in DecToStr. If the input string is shorter than the picture string, leading blanks are added until it is the same length as the picture. A currency character can appear only once in the input string, and it must be adjacent to the highest order digit. Commas must be matched unless they appear to the left of an integer. The sign character must be matched by either a '+', '-', or blank. Decimals points must be matched unconditionally.

In the following picture, the '9's specify that small dollar amounts must be entered in standard form:

```
$$$,$$9.99
```

Here are some valid input strings for this picture:

```
     $0.79
   $121.11
    $99.44
$48,000.00
```

Pictures enforce format and range constraints on integer values passed as input strings. StrToDec sets DecValid to FALSE and the decimal result to Invalid in the following conditions:

- The input string does not match the picture specification.
- The input string is longer than the picture string.
- The input string and picture specify more than 19 digits.

```
DEFINITION MODULE Decimals;

EXPORT QUALIFIED
  DECIMAL, DecDigits, DecPoint, DecSep, DecCur, DecStatus,
  DecState, DecValid, StrToDec, DecToStr, NegDec, CompareDec,
  AddDec, SubDec, MulDec, DivDec, Remainder, SetDecHandler;

CONST   DecDigits= 19;
        DecCur   = '$';
        DecPoint = '.';
        DecSep   = ',';

TYPE    DECIMAL;
        DecState = (NegOvfl, Minus, Zero, Plus, PosOvfl, Invalid);

VAR DecValid: BOOLEAN;  (* set after every operation *)
     Remainder: CHAR;    (* remainder digit - set after DivDec *)

PROCEDURE StrToDec   (String  : ARRAY OF CHAR;
                      Picture: ARRAY OF CHAR): DECIMAL;

PROCEDURE DecToStr   (Dec          : DECIMAL;
                      Picture      : ARRAY OF CHAR;
                      VAR RsltStr: ARRAY OF CHAR);

TYPE DecHandler = PROCEDURE (DecState);

PROCEDURE SetDecHandler (handler: DecHandler);

PROCEDURE DecStatus (Dec: DECIMAL): DecState;

PROCEDURE CompareDec (Dec0, Dec1: DECIMAL): INTEGER;

PROCEDURE AddDec    (Dec0, Dec1: DECIMAL): DECIMAL;

PROCEDURE SubDec    (Dec0, Dec1: DECIMAL): DECIMAL;

PROCEDURE MulDec    (Dec0, Dec1: DECIMAL): DECIMAL;

PROCEDURE DivDec    (Dec0, Dec1: DECIMAL): DECIMAL;

PROCEDURE NegDec    (Dec0, Dec1: DECIMAL): DECIMAL;

END Decimals.
```

6 Strings

The module Strings provides routines for manipulating variable-length character strings.

The predeclared type STRING is provided for convenience; additional string types can also be used with the string operators, but they must be declared with a lower bound of zero in order to work correctly.

Assign assigns the contents of string variable source into string variable dest.

Insert inserts the string substr into str, starting at str[inx].

Delete deletes len characters from str, starting at str[inx].

Pos returns the index into str of the first occurrence of the substring substr. Pos returns the value HIGH(str)+1 if no occurrence of the substring is found.

Starting at str[inx], Copy copies len characters into result.

Concat returns the concatenation of s1 and s2 in result.

Length returns the number of characters in str.

CompareStr compares two strings and returns an integer value indicating the comparison result: -1 if s1 is less than s2; 0 if s1 equals s2; 1 if s1 is greater than s2.

NOTE- String operators terminate the program with program result StringError if the operation causes either an invalid string index or string overflow.

```
DEFINITION MODULE Strings;

EXPORT QUALIFIED  STRING, Assign, Insert, Delete,
                  Pos, Copy, Concat, Length, CompareStr;

TYPE STRING = ARRAY [0..80] OF CHAR;

PROCEDURE Assign (VAR source, dest: ARRAY OF CHAR);

PROCEDURE Insert  (substr: ARRAY OF CHAR;
                   VAR str: ARRAY OF CHAR;
                   inx : CARDINAL);

PROCEDURE Delete  (VAR str: ARRAY OF CHAR;
                   inx: CARDINAL;
                   len: CARDINAL);

PROCEDURE Pos (substr, str: ARRAY OF CHAR): CARDINAL;

PROCEDURE Copy    (str: ARRAY OF CHAR;
                   inx: CARDINAL;
                   len: CARDINAL;
                   VAR result: ARRAY OF CHAR);

PROCEDURE Concat  (s1, s2: ARRAY OF CHAR;
                   VAR result: ARRAY OF CHAR);

PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;

PROCEDURE CompareStr (s1, s2: ARRAY OF CHAR): INTEGER;

END Strings.
```

7 Conversions

The module Conversions provides representation conversions for the basic numeric types. A result value of TRUE is returned after successful conversions. String arguments cannot have any leading or trailing blanks.

```
DEFINITION MODULE Conversions;

FROM SYSTEM IMPORT WORD;

EXPORT QUALIFIED
  IntToStr, StrToInt, CardToStr, StrToCard, HexToStr, StrToHex;

PROCEDURE IntToStr    (i: INTEGER;
                       VAR s: ARRAY OF CHAR): BOOLEAN;

PROCEDURE StrToInt    (s: ARRAY OF CHAR;
                       VAR i: INTEGER): BOOLEAN;

PROCEDURE CardToStr   (c: CARDINAL;
                       VAR s: ARRAY OF CHAR): BOOLEAN;

PROCEDURE StrToCard   (s: ARRAY OF CHAR;
                       VAR c: CARDINAL): BOOLEAN;

PROCEDURE HexToStr    (w: WORD;
                       VAR s: ARRAY OF CHAR): BOOLEAN;

PROCEDURE StrToHex    (s: ARRAY OF CHAR;
                       VAR w: WORD): BOOLEAN;

END Conversions.
```

8 ASCII

The module ASCII defines symbolic names for the ASCII control characters.

```
DEFINITION MODULE ASCII;

EXPORT QUALIFIED
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em,  sub, esc, fs,  gs,  rs,  us,  del;

CONST
        nul = 00C;  soh = 01C;  stx = 02C;  etx = 03C;
        eot = 04C;  enq = 05C;  ack = 06C;  bel = 07C;
        bs  = 10C;  ht  = 11C;  lf  = 12C;  vt  = 13C;
        ff  = 14C;  cr  = 15C;  so  = 16C;  si  = 17C;
        dle = 20C;  dc1 = 21C;  dc2 = 22C;  dc3 = 23C;
        dc4 = 24C;  nak = 25C;  syn = 26C;  etb = 27C;
        can = 30C;  em  = 31C;  sub = 32C;  esc = 33C;
        fs  = 34C;  gs  = 35C;  rs  = 36C;  us  = 37C;
        del = 177C;

END ASCII.
```

Index

Modula-2

on the

UCSD Pascal System

Release:	**0.3**
Date:	**26 August 1983**
Author:	**Richard Gleaves**

Table Of Contents

1 Introduction

This document describes Volition Systems' implementation of the Modula-2 language for the version II UCSD Pascal system. It covers the following topics:

- The library system
- The Modula-2 compiler
- How to use the system

Chapter 2 describes the library system, including the library organization and system-dependent library modules. If you have not yet read it, see **Introduction to Modula-2** for an introduction to the library.

Chapter 3 describes the Modula-2 language implementation, including compile options, language extensions, deviations and restrictions, and implementation notes. For additional information see the **Implementation Guide.**

Chapter 4 explains how to use the system, including operation of the compiler and library manager, and programming techniques.

2 Library

This chapter describes the Modula-2 library implemented on the UCSD Pascal operating system. It covers the following topics:

- Library organization
- System-dependent library modules
- Standard library on UCSD Pascal

Section 2.1 describes the library organization, including module segment assignment, module version control, library access by the compiler and loader, and library usage during program development. Operation of the library manager program is described in chapter 4.

Section 2.2 presents library modules specific to the UCSD Pascal system. These modules do not themselves contain any code; they merely provide access to facilities defined in the underlying UCSD Pascal operating system. System-independent library modules are described in **Standard Library** and **Utility Library.**

Section 2.3 describes the implementation of the Modula-2 portable library on the UCSD Pascal system, including library module segment assignment, file naming conventions, and the mapping of Pascal system errors onto the standard error results.

2.1 Library Organization

This section describes the Modula-2 library organization. It covers the following topics:

- Module segment assignment
- Compile-time modules
- Module version control
- Library files
- Library access
- Library usage

NOTE- This section makes references to the **loader.** The loader is a standard library module named "Program"; it is used by Modula-2 programs to perform subprogram calls; i.e. to call other programs as procedures. The loader is described in **Standard Library.**

2.1.1 Module Segment Assignment

Separate compilation in Modula-2 is related to UCSD Pascal's intrinsic unit concept. This scheme eliminates the need for linking, but requires units to be assigned segment numbers at compile time. Disadvantages of intrinsic units include the use of two segments per unit (one for code, another for data), a limited number of segments per program, and the requirement that intrinsic units be stored in the system library.

The Modula-2 system addresses each of these problems. Library modules in Modula-2 use one segment number for both code and data. The segment table has been increased to 64 segments and is saved on subprogram calls, allowing development of programs that use arbitrarily many segments. Library modules can be stored either in the (Modula-2) system library file, in program code files, or as individual files.

Only definition modules are assigned segment numbers; program modules always occupy segment 7, and implementation modules use the segment number assigned to their definition modules. Segment numbers are assigned with the compiler directive $SEG (3.2.2).

NOTE- The compiler issues an error when compiling definition modules that fail to specify a segment number.

Example of segment number assignment:

```
DEFINITION MODULE SegDemo;
(* $SEG := 43; *)      (* SegDemo is assigned segment 43 *)

  FROM SYSTEM IMPORT WORD;
  ...

END SegDemo.
```

A basic step in designing a Modula-2 program is the allocation of segment numbers for its separately compiled modules. Segment numbers must be unique with respect to the program's compilation units and imported standard library modules.

In a typical system configuration, the 64 segments are allocated as follows:

- Segments 0 through 6 are reserved for use by the Modula-2 system. The system may crash if you assign these numbers to your own library modules.
- Program modules always reside in segment 7.
- Segments 48 through 63 are provisionally reserved for the library modules provided with the system; they can be used only if the program does not import the corresponding standard library module. (See 2.3 for details.) Note that additional segments may be similarly reserved for user-defined system library modules.
- The remaining segments (8 through 47) are available for program-specific library modules. Because the segment table is saved on subprogram calls, these segments can be allocated without regard to the segments used by called subprograms.

Thus, the segment allocation strategy is to first use up the segments available for programs. To obtain additional segments, use the segment numbers of standard modules not imported by the program. If you need even more segments, it is time to divide your program into a number of subprograms, assigning segments to subprogram modules so they overlay the main program's segments. (Because of limited run-time space, it is unlikely that a single program will ever use all 64 segments; large programs are usually designed at the outset as collections of subprograms.)

NOTE- Subprograms cannot overlap the segment numbers of library modules that are imported by both the subprogram and the calling program (so-called "shared" modules).

2.1.2 Compile-time Modules

Large software systems often contain collections of constant and type declarations that are shared by a number of programs. In Modula-2, such declarations can be neatly encapsulated within a definition module. This offers a number of advantages over the common practice of using "include" files:

- Modules allow better control over the visibility of common types and constants.
- Modules are distributed in compiled form, so they cannot be modified by anyone but the distributor.
- The system performs automatic version checking on modules.

The Modula-2 system provides a special form of definition module for encapsulating constant and type declarations; they are called **compile-time modules.** Compile-time modules are syntactically identical to regular definition modules — the only difference is that compile-time modules are assigned segment number 1.

The compiler and loader treat compile-time modules specially. When a compile-time module is compiled, the compiler automatically produces an empty object file, so a matching implementation module need not be written. The loader performs the usual version checking, but does not allocate code or data segments for the module (hence the name "compile-time module"). The loader also ignores multiple occurrences of segment 1, allowing programs to import more than one compile-time module.

In short, compile-time modules offer the benefits of library modules without consuming segment numbers and run-time memory space.

NOTE- The compiler issues an error when compiling compile-time modules that contain procedure or variable declarations. It also flags implementation modules whose definition modules are assigned segment 1.

NOTE- Not all procedures are barred from compile-time modules. See 3.1.3 for details.

Example of a compile-time module:

```
DEFINITION MODULE Pcodes;
(* $SEG := 1; *)   (* Segment 1 marks this as compile-time *)

  EXPORT QUALIFIED LDB, STB, STO, NOT;

  CONST  LDB   =  0BEH;
         STB   =  0BFH;
         STO   =   9AH;
         NOT   =   93H;

END Pcodes.
```

2.1.3 Module Version Control

When a definition module is compiled, the compiler stores a unique value into the resulting symbol file; this value is known as a **module key.** When a **client** module (i.e. one that imports library modules) is compiled, the compiler stores the module key of each imported module into the resulting object file. Version control consists of checking that module keys stored in a client module match the module keys stored in the imported library modules.

Version control is performed both at compile time and run time. Compile-time checking detects the case of mismatched symbol files (i.e. where an imported definition module in turn imports another definition module, and the module key in the first symbol file's reference information does not match the module key in the second symbol file). When the compiler finds a version error, it prints an error message (86: "Incompatible versions of symbolic modules"), names the offending modules, and then terminates compilation. Run-time checking detects mismatched object files. When the loader finds a version error, it prints an error message (naming the offending modules) and aborts execution.

To generate unique module keys, the compiler maintains a disk-resident variable which — to extend the "key" metaphor — is called a **key holder.** When compiling a definition module, the compiler fetches a new module key from the key holder; to ensure continued uniqueness of module keys, the compiler then increments the value stored in the key holder. Thus, consecutively compiled definition modules have consecutively larger (but more importantly: unique) module key values.

The key holder is implemented as a 3-word record stored in the first block of the system library file (2.1.4). A module key consists of a two-word integer value (incremented by the compiler to generate unique keys) and a one-word integer known as the **library number.** The library number is assigned a value when a system library file is first created; its purpose is to ensure the uniqueness of module keys when program development is distributed across a number of systems (i.e. a group of programmers). Library numbers prevent aliasing of identically named library modules compiled on systems which happen to have identical values in their key holders.

When a system library is copied onto another system, the library number in its key holder should be assigned a new value; this task is accomplished with the library manager utility (4.3.1). The filer command T(ransfer can also copy system library files, but will not assign new library numbers to the copies. Solo programmers need not worry about library numbers, as the system's module key generation is sufficient to ensure version control.

2.1.4 Library Files

In this manual, the term "library" refers to the abstract notion of all accessible separately compiled modules in the system. The library is implemented as a collection of disk files which are called **library files.**

NOTE- The next three sections present a bottom-up description of the library file organization. Read 2.1.6 first if you want an overview of the structure and use of the library.

The library is composed of three parts: the **system library**, the **user library**, and the **program library.**

The **system library** is a disk file named "MODULA.LIBRARY". The system library contains all standard library modules, utility modules, and system-specific modules. Modules can be added to (or removed from) the system library with the library manager (4.3.2). The system library file normally resides on the system (boot) volume; however, the Modula-2 system first checks the prefixed volume for a system library file. (This feature allows you to test out new system library files without having to disturb the existing one.) If not on the prefixed volume, a system library file must appear on the system volume; otherwise, the Modula-2 system is inoperable.

The **user library** is a collection of disk files produced by compiling definition, implementation, and program modules. A user library file name consists of the module name followed by a file suffix indicating the module type. The suffix ".SYM" identifies symbol files of definition modules. ".MOD" identifies object files of implementation modules. ".CODE" identifies code files belonging to program modules. For instance, compiling a program module named Foon produces a code file named "FOON.CODE".

The **program library** is a collection of disk files produced by the library manager. Program files contain a program module and one or more of its subsidiary modules; like code files, program library files are identified by the file suffix ".CODE". Program and code files are called by file name, so their file names can be changed with impunity. For instance, the program file of a module named "Librarian" can be changed to (and called as) "LIB.CODE".

WARNING- To convert library module identifiers into file names, the compiler converts all letters to upper case; the resulting name is truncated if it exceeds ten characters in length. This conversion process leaves the library system vulnerable to aliasing of similarly named modules; if modules are kept in the user library, compilation of the second module deletes the first, eventually resulting in a version error. Care in choosing library module names avoids this problem altogether.

2.1.5 Library Access

This section describes the algorithms used by the compiler and loader to locate modules in the library.

To locate an imported definition module, the compiler first searches the system library. If the module is not in the system library, the loader looks for a user library file named "<module name>.SYM", first on the prefix volume, then on the system volume. If the module is still missing, compilation terminates with an error message.

NOTE- The compiler is incapable of searching the program library. Symbol files must be in either the system library or the user library.

The loader is passed the name of a subprogram to call. This name is interpreted either as a module name or file name.

To locate a called subprogram, the loader first searches the subsidiary modules associated with the current program. If the module is not found, the loader searches the system library, then looks for a library file named "<subprogram name>.CODE" — first on the prefix volume, then on the system volume. If the subprogram cannot be found, the loader immediately returns with the result value MissingProgram.

NOTE- The loader recognizes file naming conventions of the Pascal system's X(ecute command. If a subprogram name ends with a period ("."), the loader does not append the code file suffix when it searches for a library file. This allows for calling arbitrarily named programs (e.g. "SYSTEM.FILER."). If a subprogram name includes a volume name, the loader searches only the specified volume. This allows for calling programs on specific volumes (e.g. "#5:PATCH").

Once a called program module is installed, the loader must locate each of its imported library modules. The loader first checks that an imported module is not already resident and in use by the calling program. If it is, the module need not be searched for; otherwise, the loader searches the library.

To locate an imported library module, the loader first searches the subsidiary modules associated with the program, then the system library, and finally for a file named "<module name>.MOD" (first on the prefix volume, then on the system volume). If the library module is still not found, the loader returns the result value MissingModule and writes an error message naming the missing module.

NOTE- The loader retains library information in memory to make subprogram calls more efficient. As a consequence, system library modules and subsidiary modules of all called programs must be uniquely named; otherwise, the loader returns the result value DuplicateName and writes an error message naming the duplicated modules.

NOTE- Before searching a library file, the compiler and loader verify that the internal file structure matches the library structure implied by the library file name. The compiler responds to improperly structured library files by terminating with an error message; the loader returns the result value LibraryError and writes an error message.

NOTE- Modules stored in a library file can be "hidden" from the compiler and loader so they appear not to be in the library (see 4.3.1 for details).

2.1.6 Library Usage

The Modula-2 library provides different kinds of libraries in order to efficiently support both program development and program execution. The user library is intended for program development, while the program library is intended for efficient execution of production programs. The system library constitutes an (extensible) operating system used by all programs. These libraries are characterized by the manner in which the compiler and loader access them.

The user library is suited to program development, where ease of recompiling and reexecuting is most important. With the user library, a program can be executed immediately after one of its modules is recompiled, as the user library can be updated without executing the library manager. (In fact, recompiling a module is sufficient to update the user library.) Programs in the user library load slowly, as the loader must search the disk volume for each referenced user library file.

Program library files are designed solely for program execution, where the time required to load a program becomes critical. The library manager must be used in order to create or update a program library. Programs in the program library load quickly, as the loader searches the disk volume only once for the program library file. Production programs are usually wholly contained in a single program library file.

The system library is designed for fast access during program development and execution, as it contains modules imported by most programs. In systems containing only a handful of system modules and relatively few program-specific modules, the system library might efficiently serve as the sole library file. In normal circumstances, however, the system library file is quite large to begin with; adding all the program-specific modules would make it unwieldy and inefficient to update. (Recall that the library manager utility must be executed to update the system library.) For this reason, modules specific to production programs are best kept in program library files.

See 4.4.4 for more information on efficient use of the library.

2.2 System-dependent Modules

This section describes the system-dependent library modules provided with the Modula-2 system. Note that these are compile-time modules, and thus contain no code; they merely serve as interfaces to facilities contained in the UCSD Pascal system.

The syntax (i.e. names and parameter lists) for Modula-2's UCSD system calls differs only slightly from the corresponding UCSD Pascal system calls; the semantics are identical. This section primarily describes syntactic differences; semantic details can be found in the UCSD Pascal system manual.

The main source of syntax differences arises from the replacement of UCSD Pascal's single-argument address parameters with dual-argument "byte-address pairs" specifying a base address and a byte offset. For example, a parameter passed as "ByteArray[3]" in UCSD Pascal is passed as the two parameters "ADR(ByteArray), 3" in Modula-2. The procedure ADR must be imported from SYSTEM in order to pass the address of statically declared variables. Non-indexed actual parameters (such as record variables) pass a byte offset of 0 along with the proper address.

2.2.1 Screen Control

The module Screen provides basic screen control functions. HomeCursor moves the cursor to the upper left hand corner of the screen. ClearScreen erases the entire screen and "homes" the cursor. EraseLine erases the screen from the cursor position to the end of the current line. GotoXY moves the cursor to the specified X-Y screen coordinates.

```
DEFINITION MODULE Screen;  (* $SEG := 1; *)

EXPORT QUALIFIED HomeCursor, ClearScreen, EraseLine, GotoXY;

PROCEDURE HomeCursor;  (* move cursor to upper left *)

PROCEDURE ClearScreen;  (* erase screen, home cursor *)

PROCEDURE EraseLine;  (* erase from cursor to end of line *)

PROCEDURE GotoXY(x, y: CARDINAL);  (* move to column x, row y *)

END Screen.
```

2.2.2 System Attributes

The module SystemTypes provides system-dependent attributes of the basic Modula-2 types. MinInt and MaxInt indicate the extreme values assumable by variables of type INTEGER. MaxCard indicates the maximum value assumable by variables of type CARDINAL (the minimum value is implicitly zero). AdrsPerWord indicates the number of address increments that span a word (in this case, the basic addressing unit is a byte, and two bytes constitute a word). CharsPerWord indicates the number of characters that can fit in a single word.

```
DEFINITION MODULE SystemTypes;  (* $SEG := 1; *)

EXPORT QUALIFIED
  MinInt, MaxInt, MaxCard, AdrsPerWord, CharsPerWord;

CONST   MinInt  = -32768;
        MaxInt  =  32767;
        MaxCard =  65535;

        AdrsPerWord  = 2;
        CharsPerWord = 2;

END SystemTypes.
```

2.2.3 Block File I/O

The low-level module BlockIO provides access to the block I/O facilities in the UCSD file system. File variables are declared with type FILE (as in UCSD Pascal). File variables must be initialized with the procedure InitFile before they are used.

File names passed to the Reset and Rewrite routines must be converted to the internal representation of a UCSD Pascal string variable; Modula-2 strings will not work! The first character in a UCSD-format string is a length byte indicating the number of characters in the string. The first letter is at index 1.

Passing the value -1 to the startblock parameter in BlockRead (BlockWrite) specifies that blocks are to be read from (written to) the next block in the file. This feature allows disk files to be read sequentially without having to specify a block number.

WARNING- If you do not close a file opened with Rewrite, the disk directory is left in an erroneous state. (The system command I(nit corrects it.)

```
DEFINITION MODULE BlockIO;  (* $SEG := 1; *)

FROM SYSTEM IMPORT ADDRESS, WORD;

EXPORT QUALIFIED FILE, BlockRead, BlockWrite, Reset, Rewrite,
                 Close, InitFile, FileName, CloseType;

TYPE
    FileName = ARRAY [0..39] OF CHAR;  (*UCSD format string*)

    CloseType= (Normal, Lock, Purge, Crunch);

    FILE     = ARRAY [0..30] OF WORD;

(* Note that INTEGER params are to be used as
   CARDINAL.  They are declared as INTEGER to
   match UCSD op sys declarations exactly *)

PROCEDURE InitFile(VAR f: FILE);
   (* Initialize FILE variable...must be done before any
      other routines can be called. *)

PROCEDURE Reset(VAR f: FILE; VAR fn: FileName);
   (* Open existing file *)

PROCEDURE Rewrite(VAR f: FILE; VAR fn: FileName);
   (* Open new file *)

PROCEDURE Close(VAR f: FILE; ftype: CloseType);
   (* Close file, and update directory...
          Normal      Leave if opened with Reset, remove if
                      opened with Rewrite.
          Lock        Save permanent entry in directory.
          Purge       Remove entry from directory.
          Crunch      Save permanent entry, but truncate
                      file at current file position. *)

PROCEDURE BlockRead(VAR f: FILE; buf: ADDRESS; byteindex: INTEGER;
                    nblocks, startblock: INTEGER): INTEGER;
   (* Read nblocks of the file into memory *)

PROCEDURE BlockWrite( VAR f: FILE; buf: ADDRESS; byteindex: INTEGER;
                    nblocks, startblock: INTEGER): INTEGER;
   (* Write nblocks of the file from memory *)

END BlockIO.
```

Example of block file I/O:

```
MODULE BlockExample;

FROM SYSTEM IMPORT ADR;

FROM UnitIO IMPORT IOResult, INoError;

FROM BlockIO IMPORT   FILE, FileName, InitFile, BlockRead,
                      BlockWrite, Reset, Rewrite, Close, CloseType;

FROM Terminal IMPORT WriteString;

VAR  input, output: FILE;
     blks: INTEGER;
     buff: ARRAY [0..9], [0..511] OF CHAR;
     name: FileName;

BEGIN
  name := " *MODULA.LIBRARY";(* note 1st blank for UCSD length byte *)
  name[0] := 17C;              (* length in octal as type CHAR *)
  InitFile(input);
  Reset(input, name);
  IF IOResult() # INoError THEN
    WriteString("Can't find input file");
    HALT;
  END;
  name := " *DUPLIB";
  name[0] := 7C;
  InitFile(output);
  Rewrite(output, name);
  IF IOResult() # INoError THEN
    WriteString("Can't open output file");
    Close(input, Normal);
    HALT;
  END;

  REPEAT
    blks := BlockRead(input, ADR(buff), 0, HIGH(buff) + 1, -1);
    IF BlockWrite(output, ADR(buff), 0, blks, -1) # blks THEN
      WriteString("Error writing output");
      Close(input, Normal);
      Close(output, Purge);
      HALT;
    END;
  UNTIL blks <= INTEGER(HIGH(buff));

  Close(input, Normal);
  Close(output, Lock);
END BlockExample.
```

2.2.4 Unit I/O

The low-level module UnitIO provides access to UCSD Pascal's unit I/O system. The scalar constants in type IOResultType are declared to match the standard I/O error values returned by the system. The BlkNum parameter is ignored when accessing a non-block-structured unit. The FlagWd parameter is a one-word bit array instead of an integer (as in UCSD Pascal), thus making explicit the manner in which this parameter is interpreted by the unit I/O system.

```
DEFINITION MODULE UnitIO;  (* $SEG := 1; *)

FROM SYSTEM IMPORT WORD, ADDRESS;

EXPORT QUALIFIED UnitRead, UnitWrite, UnitStatus, UnitClear,
                 UnitBusy, IOResult, IOResultType;

TYPE IOResultType = (INoError,    (* 0 *)
                     IHardErr,
                     IBadUnit,
                     IBadMode,
                     ITimeout,
                     ILostUnit,   (* 5 *)
                     ILostFile,
                     IBadTitle,
                     INoSpace,
                     INoUnit,
                     INoFile,     (* 10 *)
                     IDupFile,
                     IFileOpen,
                     INotOpen,
                     IBadFormat,
                     IBufOflow);  (* 15 *)


PROCEDURE IOResult(): IOResultType;

   (* Return value indicating the result
      of the previous I/O operation *)


PROCEDURE UnitStatus( UnitNo: CARDINAL;
                      Result: ADDRESS;
                      Option: CARDINAL);

   (* Return status of the specified unit
      - see UCSD Pascal manual for details *)
```

```
PROCEDURE UnitBusy(UnitNo: CARDINAL): BOOLEAN;

   (* Return TRUE if the specified unit is
      waiting for an I/O operation to complete *)

PROCEDURE UnitClear(UnitNo: CARDINAL);

   (* Set the specified unit back to its
      initial operating state *)

PROCEDURE UnitRead( UnitNo:  CARDINAL;
                    Buffer:  ADDRESS;
                    Index:   CARDINAL;
                    NBytes:  CARDINAL;
                    BlkNum:  CARDINAL;
                    FlagWd:  BITSET);

   (* Read bytes from I/O unit into Buffer *)


PROCEDURE UnitWrite( UnitNo:  CARDINAL;
                     Buffer:  ADDRESS;
                     Index:   CARDINAL;
                     NBytes:  CARDINAL;
                     BlkNum:  CARDINAL;
                     FlagWd:  BITSET);

   (* Write bytes in Buffer out to I/O unit *)

END UnitIO.
```

2.2.5 UCSD Standard Procedures

The low-level module Standards provides access to UCSD Pascal standard procedures.

NOTE- The scalar constants in the enumeration type ScanType are used to specify the scanning mode in the procedure Scan. The constant ScanUntil specifies that scanning continues until a scanned character matches the target character ("=" in UCSD Pascal). The constant ScanWhile specifies that scanning continues until a scanned character does not match the target character ("<>" in UCSD Pascal).

WARNING- The procedures Alloc, Mark, and Release provide low-level storage management, and cannot be used in conjunction with the standard library module Storage. Because most standard library modules import Storage (see **Standard Library** for details), these routines should be used **only** when a program limits itself to the system-dependent modules described in this section.

```
DEFINITION MODULE Standards;  (* $SEG := 1; *)

FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED MoveLeft, MoveRight, FillChar, Scan, Time, ScanType,
                 PowerOfTen, Alloc, Mark, Release, MemAvail;

TYPE ScanType = (ScanUntil, ScanWhile);

PROCEDURE MoveLeft( SrcAddr:  ADDRESS;
                    SrcInx:   CARDINAL;
                    DestAddr: ADDRESS;
                    DestInx:  CARDINAL;
                    NBytes:   CARDINAL);

   (*  Move bytes from Source to Destination, starting with
       the first byte in Source  *)

PROCEDURE MoveRight( SrcAddr:  ADDRESS;
                     SrcInx:   CARDINAL;
                     DestAddr: ADDRESS;
                     DestInx:  CARDINAL;
                     NBytes:   CARDINAL);

   (*  Move bytes from Source to Destination, starting with
       the last byte in Source  *)
```

```
PROCEDURE FillChar(DestAddr: ADDRESS;
                   DestInx:  CARDINAL:
                   NBytes:   CARDINAL;
                   FillVal:  CHAR);

   (* Initialize bytes in Dest with the byte value FillVal *)

PROCEDURE Scan( NumChars: INTEGER;
                ForPast:  ScanType;
                Target:   CHAR;
                Source:   ADDRESS;
                SrcInx:   CARDINAL): INTEGER;

   (* Starting at Source, scan for Numchars characters until
      Target character is found. Return offset from Source   *)


PROCEDURE Time(VAR Hi, Lo: CARDINAL);
   (* Return 32-bit system clock value in Hi and Lo  *)


PROCEDURE Alloc(VAR p: ADDRESS; words: CARDINAL);
   (* Allocate space on top of heap *)


PROCEDURE Mark(VAR p: ADDRESS);
   (* Save current heap position in p *)


PROCEDURE Release(VAR p: ADDRESS);
   (* Cut heap back to position specified by p *)


PROCEDURE MemAvail(): CARDINAL;
   (* Return # words between stack and heap top *)


PROCEDURE PowerOfTen(e: CARDINAL): REAL;
   (* Return 10 raised to e'th power *)

END Standards.
```

2.2.6 Bit Field Access

The module Bits provides efficient access to bit fields and byte fields of word quantities. Bits in a word are numbered 0 through 15. Bit 0 is the low order bit, bit 15 the high order bit. A bit field is specified by its rightmost (lowest order) bit and number of bits.

NOTE- In bit fields, bits 0 thru 7 always specify the least significant byte of word quantities. Byte access, however, involves physical byte addresses, and thus is independent of byte ordering in word quantities.

```
DEFINITION MODULE Bits;  (* $SEG := 1; *)

FROM SYSTEM IMPORT WORD, ADDRESS;

EXPORT QUALIFIED LoadByte, StoreByte, LoadField, StoreField;

PROCEDURE LoadByte (base: ADDRESS; offset: CARDINAL): CARDINAL;

  (* Load byte from byte address base[offset] *)


PROCEDURE StoreByte (base: ADDRESS; offset, ValueToStore: CARDINAL);

  (* Store byte at byte address base[offset] *)


PROCEDURE LoadField  (VAR w: WORD;
                      NumberOfBits: CARDINAL;
                      RightMostBit: CARDINAL): CARDINAL;

  (* Load specified bit field from word w *)


PROCEDURE StoreField  (VAR w: WORD;
                       NumberOfBits: CARDINAL;
                       RightMostBit: CARDINAL;
                       ValueToStore: CARDINAL);

  (* Store specified bit field into word w *)

END Bits.
```

2.3 Standard Library on UCSD Pascal

This section describes system-dependent details of the standard library modules on the UCSD Pascal system.

Module Segment Assignment

The following table indicates the segment numbers assigned to modules contained in the standard and utility libraries. Segments are assigned so that the most frequently used library modules reside in the highest numbered segments.

Many of the standard library modules are interdependent; importing one of these modules implies the importation of other modules, resulting in the use of extra segments. The **Implementation Guide** describes the library module hierarchy.

The utility module ASCII is a compile-time segment, and thus is assigned segment 1.

NOTE- Segments 59 through 63 are reserved for certain implementations. See the **Implementation Guide** for details.

Library module segment assignment:

Program	2	Reals	52
SubProgram	3	Strings	53
Storage	4	InOut	54
Decimal	48	Conversions	55
Processes	49	Texts	56
MathLib0	50	Files	57
RealInOut	51	Terminal	58

Program

The parameter programName is passed a string containing the name of the subprogram to be called. The interpretation of this name is described in 2.1.5.

Pascal programs are callable from the Modula-2 system with one restriction. If a Pascal program uses intrinsic units, their assigned segment numbers cannot overlap segment numbers occupied by any library modules.

The result value DuplicateName is returned if a subprogram attempts to import different library modules with the same segment number.

If a subprogram call specifies system-controlled execution error handling, the Modula-2 system displays an error message describing the execution error. See 4.4.6 for details.

The library module SubProgram is a subsidiary module of the loader. SubProgram is called (and thus resident) only between subprogram calls.

InOut

Note that the rule for appending file suffixes is the opposite of the UCSD Pascal convention — a period at the end of a file name causes a file suffix (e.g. "TEXT") to be automatically appended.

Texts & Files

File names in Texts and Files follow the UCSD Pascal file naming conventions. Files treats all disk files as pure data regardless of the file type. Texts reads and writes UCSD text file format when connected to a text file (e.g. file suffix ".TEXT").

Typing the <eof> key terminates a a console input file. The <eof> key is defined by the Pascal system; it is usually control-C.

If a file I/O error occurs in Files, the function procedure UnitIO.IOResult can be called to determine the system-specific I/O result.

Storage

Storage is implemented as a linear list of "free" areas sorted by address (highest address first). ALLOCATE traverses the first 10 free areas searching for perfect fit; if not found, it settles for first fit. DEALLOCATE collapses all adjacent free areas.

ALLOCATE and DEALLOCATE raise StorageError if passed a storage area size larger than 32766 storage units (bytes on most implementations).

Deallocating an invalid pointer variable usually causes a program to terminate with result StorageError; however, Storage cannot detect all improper deallocation of dynamic variables. Deallocating a variable with a different size than it was allocated with may crash the system. It is an error for 'unshared' subprograms to deallocate storage allocated by the calling program.

MathLib0

The procedure 'real' is not implemented. The equivalent operation is provided by the (implementation-dependent) definition of the standard procedure FLOAT.

3 Compiler

This chapter describes the Modula-2 language implementation on the UCSD Pascal system. It covers the following topics:

- Extensions
- Deviations and restrictions
- Compile options

Section 3.1 describes language constructs unique to this implementation of Modula-2; these extensions should not be used in programs intended for use on other Modula-2 implementations.

Section 3.2 describes the remaining differences; most are restrictions on the use of legal Modula-2 constructs, but a couple are deviations from the language definition.

Section 3.3 describes compile options. Compile options are implemented as directives placed within comments in a source program. Compiler directives control the use of language extensions, affect the compiler's mode of operation, and alter the code generated by the compiler.

3.1 Extensions to Modula-2

This section describes nonstandard language constructs available in this implementation of Modula-2. These constructs should not be used in programs written for portability.

NOTE- The compiler directive $STANDARD (3.3.3) controls the use of these extensions.

3.1.1 Packed Variables

The compiler attempts to compress the machine representations of records and arrays when their type definitions are prefixed with the (implementation-specific) reserved word **PACKED**. Packing significantly reduces the amount of memory needed to store certain data types, but at the expense of slightly increased execution time and code size required for packed field access. Packing is syntactically allowed for all types, but affects only records and arrays.

Examples of packed variable declaration:

```
TYPE   manybits =  PACKED ARRAY [0..31] OF BOOLEAN;
       smallrec =  PACKED RECORD
                     a,b: CHAR;
                     i: INTEGER;
                   END;
```

Machine representations of the basic data types are as follows:

type	unpacked	packed
BOOLEAN	1 word	1 bit
CHAR	1 word	8 bits
INTEGER	1 word	1 word
REAL	2 words	2 words
SET OF 0..x : x<16	1 word	(x+1) bits
subrange x..y : x>=0	1 word	(log2(y+1)) bits

Subrange types with negative lower bounds are not packable. Array and record subtypes are word aligned and thus unpackable. The compiler is limited to packing fields into single words; fields cannot be packed across word boundaries. Thus, records are packed only if they contain consecutively declared fields that can be packed into a single word, and arrays are packed only if their element types can be stored in 8 bits or less. Unpackable fields are referenced as unpacked data. (In records, this includes fields which cannot be packed because of adjacently declared unpackable fields.)

NOTE- Packed fields cannot be passed as VAR parameters. The Modula-2 compiler automatically packs character arrays; specifying them as PACKED is therefore unnecessary.

3.1.2 Forward Declarations

A procedure can be called prior to its declaration only if there is a **forward declaration.** A procedure is declared forward by following its heading with the (implementation-specific) reserved word **FORWARD** to indicate that it will be completely defined further down in the program. Forward declarations are necessary for mutually recursive procedures.

NOTE- Unlike Pascal, the parameter list and result type must be repeated exactly in the complete procedure declaration.

NOTE- The Modula-2 language definition says nothing about forward references, as procedures can be called before they are declared. This implementation requires forward declarations because the "one-pass" restrictions do not allow use before declaration (see 3.2 for details).

Example of forward declarations:

```
PROCEDURE Affine(a: T1); FORWARD;

PROCEDURE Infine(b: T1);
BEGIN
  Affine(b)
END Infine;

PROCEDURE Affine(a: T1); (* parameter list repeated *)
BEGIN
  Infine(a)
END Affine;
```

3.1.3 Code Procedures

A code procedure is a procedure declaration whose body consists of a sequence of constants denoting P-code instructions and operands. This code sequence is substituted inline for each code procedure call.

Code procedures are used to perform low-level operations and to access routines defined in the UCSD Pascal system. As in regular procedure calls, code procedure parameters are pushed onto the evaluation stack (in the order they appear) before the (inline) procedure code is executed.

WARNING- Code procedures must be used with utmost care, as any programming errors may cause the system to crash in mysterious ways. Be prepared! (The UCSD Pascal system manual describes the P-machine instruction set.)

Here is the extended syntax (in EBNF as used in the Report) for code procedure declarations:

```
ProcedureDeclaration =
    ProcedureHeading ";" (block | codeblock) ident.
codeblock = CODE CodeSequence END.
CodeSequence = code {";" code}.
code = [ConstExpression].
```

Example of code procedure declaration:

```
CONST LDB = 0BEH;

PROCEDURE UpperByte(VAR w: WORD): CHAR;
(* word address pushed as parameter *)
CODE
  1;          (* load constant byte offset *)
  LDB         (* load byte as function result *)
END UpperByte;
```

NOTE- Unlike regular procedure bodies, code procedure bodies can be declared in definition modules. This feature allows code procedures to be neatly encapsulated in library modules without requiring any run-time code or data, as such modules can be declared as compile-time modules (2.1.2).

Example of code procedures in compile-time modules:

```
DEFINITION MODULE ByteDiddler; (* $SEG := 1; *)
  EXPORT QUALIFIED UpperByte;

  CONST LDB = 0BEH;

  PROCEDURE UpperByte(VAR w: WORD): CHAR;
  CODE  1; LDB
  END UpperByte;

END ByteDiddler.
```

3.2 Differences and Restrictions

This section describes differences from the Modula-2 language definition and implementation restrictions in the current release.

- **One-pass restrictions:** Constants, types (excepting pointers), and variables must be declared before they can be referenced. Procedures can be forward declared with the FORWARD directive (3.1.2).
- **Reserved words:** The identifiers PACKED, CODE, and FORWARD are reserved words.
- **Real number conversion:** The real number conversion procedures FLOAT and TRUNC work with type INTEGER instead of type CARDINAL.
- **Cardinal division:** Cardinal division (via the DIV and MOD operators) does not work when either operand is greater than 32767.
- **Case labels:** Cardinal case label values cannot exceed MaxInt (32767) in case statements and record declarations.
- **Function results:** Function procedures can return results of any type. This differs from the current edition of **Programming in Modula-2**, which restricts function results to unstructured types. This restriction is expected to be removed from the language in the future.
- **VAR parameters:** Character array elements cannot be passed to variable parameters. (This includes the standard procedures INC and DEC.)
- **Compiler limits:** The maximum procedure size is 1200 bytes of object code. The maximum number of procedures per module is 100. The maximum level of lexical nesting is 32.
- **Run-time error checking:** The Modula-2 system does not detect integer/cardinal overflows, use of uninitialized variables, or NIL pointer references.

3.3 Compile Options

Compile options control the operation of the compiler. Compile options are controlled by directives embedded in comments in the source program. Compiler directives can appear anywhere within non-nested comments; directives in nested comments are ignored. Any number of directives can be placed in a single comment. Here is the syntax (in EBNF as used in the Report) for compiler directives:

```
Directive   =  "$"Identifier [Parameter].
Identifier  =  <option identifier>.
Parameter   =  Declare | String | Assign | Condition | Set.
Declare     =  ";".
String      =  <Modula string>.
Assign      =  ":=" <expression> ";".
Condition   =  <expression> "THEN".
Set         =  [String] Identifier.
```

The "$" character marks the beginning of a directive; blanks cannot appear between the "$" and the subsequent identifier. Comment text not associated with a directive is ignored, and thus may serve as comments describing the directive.

Example of compiler directives:

```
(* $TO "LIST.TEXT"  ... make a compiled listing *)
(* $STANDARD:=FALSE; $RECYCLE:=TRUE; $SEG:=45; *)
(* $SET "Debug output?" debug *)
(* $IF debug THEN *)
(* $PUSH RANGE  save state of range checking *)
```

Option identifiers are either compiler commands or compile-time variables. Compiler commands cause the compiler to perform a specific action. Compile-time variables are variables whose values control the compiler's operating mode. Directives are used to declare and assign values to compile-time variables.

There are two types of compile-time variables: **cardinal variables** and **Boolean variables.** Cardinal variables assume cardinal numbers as values. Boolean variables assume the values TRUE or FALSE.

NOTE- Compile-time variables are distinct from program variables and can only be used within compiler directives.

WARNING- Invalid compiler directives cause compiler syntax errors. If the error occurs in an expression, the compiler generates a suitable error message (as if it were compiling a Modula-2 expression); otherwise, it generates syntax error 16 ("Compiler directive error").

3.3.1 Interactive Compile Options

Compile-time variables are usually set by directives embedded in the source text; however, the compiler command SET requests directive values from the keyboard each time a program is compiled.

SET accepts a string and a variable name as parameters; the string parameter is optional. SET writes a console prompt and reads a value into the variable. If a string parameter is included, it is written to the console as a promptline; otherwise, the variable name is written to the console followed by a question mark. Typing 'T' or 'Y' sets the variable to TRUE. Typing 'F' or 'N' sets the variable to FALSE. Typing a cardinal number sets the variable to the specified value (backspaces are allowed). Compilation resumes after typing a valid response.

Example of SET:

```
(* $SET "Range checking? " RANGE *)
```

The compiler command TYPE accepts a string parameter and causes the compiler to write the string to the console. The string is followed by a carriage return. (Passing an empty string is equivalent to "WriteLn".) TYPE is used to precede occurrences of the SET command with informative messages.

Example of TYPE:

```
(* $TYPE "Set only one of these options:" *)
(* $TYPE "" empty string writes a blank line *)
(* $SET Apple2 *)
(* $SET Apple3 *)
(* $SET IBMPC  *)
```

3.3.2 Stacked Options

The values assumed by Boolean variables can be stacked. Stacking is useful for setting a Boolean variable in a small part of a program and then restoring its previous value. The compiler command PUSH saves the current value of a Boolean variable. The compiler command POP restores a Boolean variable to its previous value. PUSH and POP accept Boolean variables as arguments. Boolean values can be stacked to 15 levels deep,

Example of option stacking:

```
PROCEDURE RiskyIndex;
VAR IntArray: ARRAY [1..100] OF INTEGER;
    I,X: INTEGER;

BEGIN
  ...
  (*$PUSH RANGE $RANGE:=FALSE;  range checking off *)
  I := IntArray[X];
  (*$POP RANGE *)          (* restore range checking *)
  ...
END RiskyIndex;
```

Definition modules are assigned segment numbers with the predeclared cardinal variable SEG. The identifier is assigned a cardinal value indicating the desired segment number. The directive must appear immediately after the compilation unit's module heading.

Example of module segment assignment:

```
DEFINITION MODULE SegDemo;
(*$SEG := 43;*)      (*SegDemo is assigned segment 43*)
  FROM SYSTEM IMPORT WORD;
  ...

END SegDemo.
```

3.3.3 Standard Language

The predeclared Boolean variable STANDARD controls the use of nonstandard language constructs. The compiler accepts extensions only if STANDARD is set to FALSE at the top of a compilation unit. The default setting is TRUE.

Example of nonstandard language use:

```
(* $STANDARD := FALSE; *)
DEFINITION MODULE NonStandard;
  ...

  TYPE BitString = PACKED ARRAY [1..77] OF BOOLEAN;

  PROCEDURE ProcessorHalt;
  CODE
    0FFH
  END ProcessorHalt;

END NonStandard.
```

3.3.4 Include Files

Text files can be "included" into a compilation unit with the compiler command IN. The string parameter contains the name of the text file to be included. The file suffix ".TEXT" is optional. Compilation terminates (with error 10) if an included text file cannot be opened. Include files cannot be nested.

Example of include files:

```
DEFINITION MODULE VeryLowLevel;
  ...

  (* Compile declarations
     contained in UCSDOPS.TEXT *)

  (* $IN "UCSDOPS" — get IND from UCSDOPS.TEXT *)

  PROCEDURE Peek(a: ADDRESS): CARDINAL;
  CODE
    IND; 0
  END Peek;

END VeryLowLevel.
```

3.3.5 Compiled Listings

Compiled listing are produced with the compiler command TO. The string parameter contains the name of the listing file. The directive must appear at the top of the compilation unit.

The predeclared Boolean variable LIST controls the generation of a listing; it is used to selectively include or exclude parts of a program from the listing. Setting LIST to FALSE disables listing; setting LIST to TRUE enables listing. The TO command automatically enables listing. The LIST command is ignored if a TO command has not been specified.

Sample compiled listing:

```
 1   7    1:D  0 (* $TO "stuff.text" *)
 2   7    1:D  1 MODULE test;
 3   7    1:D  1
 4   7    2:D  1   PROCEDURE testproc;
 5   7    2:D  1   VAR i,j,k: INTEGER;
 6   7    2:C  0   BEGIN
 7   7    2:C  0    FOR i := 1 TO 10 DO
 8   7    2:C  3     FOR k := 1 TO 10 DO
 9   7    2:C  6      FOR j := 1 TO 10 DO
10   7    2:C  9      END;
11   7    2:C 16     END;
12   7    2:C 23    END;
13   7    2:C 30   END testproc;
14   7    2:C 48
15   7    1:C  0 BEGIN
16   7    3:C  0   testproc;
17   7    1:C  2 END test.
```

The first column in the listing displays the line number in the listing. The second column is the segment number. The third column is the procedure number. If the character after the colon is a "C", the line is a statement, and the value in the last column is the code offset of the beginning of the statement. If the character is a "D", the line is a declaration, and the value in the last column is the data offset of the first variable on the line.

Compiled listings are used to debug programs; in particular, for locating execution errors. See 4.4.6 for details.

Example of listing directives:

```
(*  $TO "PRINTER:"  *)
IMPLEMENTATION MODULE Classified;
  ...
  (* $PUSH LIST $LIST := FALSE; *)
  TopSecret := Truth[Beauty];
  (* $POP LIST *)
  ...

END Classified.
```

3.3.6 Run-time Checks

The generation of code for performing run-time checks is controlled by the predeclared Boolean variable RANGE. Setting RANGE to TRUE enables run-time checking; setting RANGE to FALSE disables run-time checking. The default setting is TRUE.

Compiler-controlled checks protect the following operations:

- integer/cardinal assignment
- assignment to subranges (including value parameters)
- array indexes
- FOR loop subranges
- INCL, EXCL, set construction
- reaching the end of a function procedure without executing a RETURN statement

Example of range check suppression:

```
PROCEDURE RiskyIndex;
VAR IntArray: ARRAY [1..100] OF INTEGER;
    I,X: INTEGER;

BEGIN
  ...
  (*$PUSH RANGE $RANGE:=FALSE;  range checking off *)
  I := IntArray[X];
  (*$POP RANGE *)         (* restore range checking *)
  ...
END RiskyIndex;
```

3.3.7 Quiet Compile

The predeclared Boolean variable QUIET controls the compiler's console display. The compiler can be operated in the so-called "quiet" mode by setting QUIET to TRUE at the top of a compilation unit. In quiet mode, the compiler suppresses its normal console display (4.1.2) and does not stop when a syntax error is discovered. The default setting is FALSE.

Example of specifying quiet compilation:

```
(* $QUIET:=TRUE; *)
MODULE Silence;
  ...

END Silence.
```

3.3.8 Copyright Notices

Copyright notices are placed near the front of symbol, object, or code files with the compiler command NOT. The string parameter contains the textual message to be embedded in the output file. Copyright directives must appear after the initial module heading, but before any procedure or module bodies.

Example of copyright notices:

```
MODULE Business;
(*$NOT "Copyright 1977, by Dee Ltd." *)
  ...

END Business.
```

3.3.9 Half-ASCII Terminals

Some popular microcomputers do not support the full ASCII character set; in particular, lower-case alphabetic characters, braces ("{" and "}"), and the vertical bar "|" are missing. Unfortunately, these characters are used as symbols in Modula-2. To avoid this problem, the predeclared Boolean variable UPCASE alters Modula-2's vocabulary to accommodate half-ASCII keyboards. The predeclared Boolean variable SPECIAL is provided for upper/lower case terminals lacking braces and bars.

Setting UPCASE to TRUE causes the following changes to the Modula-2 vocabulary. The exclamation point "!" can be substituted for the vertical bar "|" in case statements and record variants. Square brackets "[" and "]" can be substituted for braces in set constants. (Note that this matches Pascal's syntax.) Finally, lower and upper case alphabetic characters are considered equivalent. (Note that this matches Pascal's case insensitivity.)

The UPCASE directive must appear at the top of a program. The default setting is FALSE.

The variable SPECIAL is used identically to UPCASE. Setting SPECIAL to TRUE provides the special character substitution, but retains case significance of identifiers.

NOTE- Programs that make use of these options are nonportable, as they are not standard Modula-2 programs.

Example of half-ASCII Modula-2:

```
(* $UPCASE := TRUE; *)
MODULE PrAgMaTiCs;
 ...

CONST SETK = bitset[0,1,4,9];

TYPE RECTYPE = RECORD
                 CASE INTEGER OF
                   0:  i: INTEGER !
                   1:  r: REAL    !
                   2:  c: char
                 END;
               END;

end pRaGmAtIcS.
```

3.3.10 Extra Compile Space

The compiler may run out of symbol table space when compiling large modules; it does so by ungracefully expiring with a "stack overflow". The predeclared Boolean variable RECYCLE is provided to gain extra symbol table space at the expense of slower compilation. Setting RECYCLE to TRUE causes the compiler to recycle all storage consumed by unimported library module declarations; this often amounts to more than 2000 extra words of compile-time space. The directive must appear at the top of a program. The default setting is FALSE.

NOTE- If the compiler determines that it needs the storage, it may automatically enable recycling. (This only occurs when compiling programs which import large numbers of library module identifiers.) For more information on using the recycle option, see 4.1.2 and 4.4.1.

Example of specifying extra space:

```
(* $RECYCLE:=TRUE;  recycle symbol table *)
MODULE Big;
  ...

END Big.
```

3.3.11 Byte Flipping

The compiler generates byte-flipped code files by setting the predeclared Boolean variable FLIP to TRUE at the top of a program. Byte-flipped code files are executable only on processors of the opposite byte sex from the host processor. The directive must appear at the top of the program. The default setting is FALSE.

Example of byte-flipping option:

```
(* $FLIP:=TRUE; *)
MODULE A;
  ...

END A.
```

3.3.12 Conditional Compilation

The compiler commands IF, ELSIF, ELSE, and END allow selective inclusion or exclusion of sections of a source program. Selection is controlled by expressions consisting of compile-time variables.

User-defined compile-time variables are declared when they first appear as an option identifier in a compiler directive (or as an argument to the SET command). An uninitialized variable can be declared with the directive "$<identifier>;". Note that variable identifiers must be longer than one character.

Example of compile-time variable declaration:

```
(* $NOBUGS;   set it later with SET *)

(* $Stripped := FALSE; *)

(* $SET "Include Screen Module?" UseLocalScreen *)
```

NOTE- User-defined variables must be declared at the top of a program.

The type of a variable (either Boolean or cardinal) is determined by the type of its initial value; after the initial assignment, variables cannot be assigned values of another type.

Variables must be assigned values before they can be used in expressions. The full expression syntax is allowed (e.g. complex expressions, logical and arithmetic operators); however, only compile-time variables and integer or Boolean constants may be used as operands.

The IF command conditionally causes all source text to be skipped up to the subsequent ELSIF, ELSE, or END command. Expressions in IF and ELSIF must be of type BOOLEAN and must be terminated by the symbol THEN.

NOTE- Skipped text is treated as a continuation of the original comment containing the directive; thus, the compiler ignores all compiler directives within the skipped source. Conditional compilation directives cannot be nested. Comments containing conditional compilation directives should be closed immediately after the command.

WARNING- Conditional compilation directives render programs nonportable, as their non-interpretation by different Modula-2 compilers may change the semantics of a program.

Example of conditional compilation:

```
(* $SET "Compile what version?" Version *)

(* $IF (Version = 2) AND Apple2 THEN *)

  CONST BufSize  = 8500;
        GotPool  = FALSE;
        NumSegs  = 64;
  ...

(* $ELSIF Version = 4 THEN *)

  CONST BufSize  = 3000;
        GotPool  = TRUE;
        NumSegs  = Infinity-1;
  ...

(* $ELSE *)

  CONST BufSize  = 10000;
        GotPool  = FALSE;
        NumSegs  = 16;
  ...

(* $END *)
```

Another example of conditional compilation:

```
(* $IF DEBUG AND InternalRelease THEN *)

  IF BuffInx > MaxBuff THEN
    WriteString("Call Roger immediately");
    WriteHex(CompilerVersion, 4);
    HALT;
  END;

(* $END *)
```

3.3.13 Symbolic Execution Error Messages

The compiler generates debugging information when the predeclared Boolean variable DEBUG is set to TRUE. DEBUG controls the appearance of symbolic procedure names in execution error messages (see 4.4.6 for details). Note that setting DEBUG to TRUE results in larger code files, as the identifier of each procedure in the module is embedded in the code file (along with two bytes of overhead per procedure). To minimize code file expansion, DEBUG can be turned on and off within a program to select only a few procedures for symbolic error displays. The default setting of DEBUG is FALSE.

Example of debug option:

```
(* $DEBUG := TRUE; *)
MODULE Buggy;
  ...

END Buggy.
```

4 How To Use The System

This chapter describes the operation of the Modula-2 system. It covers the following topics:

- Compiling programs
- Executing programs
- Library management .
- Programming techniques

Section 4.1 explains how to compile programs: how to invoke the compiler, what the compiler's console display means, and how to correct syntax errors.

Section 4.2 explains how to execute programs.

Section 4.3 explains library management: how to add modules to the system library, and how to bind a program module and all its library and subprogram modules into a single executable code file.

Section 4.4 describes some useful programming techniques: how to structure large programs to compile and run in limited storage, how to locate execution errors, and tips on using the library effectively.

4.1 Compiling Programs

The Modula-2 compiler is a one-pass recursive descent compiler for the Modula-2 language. It is written in UCSD Pascal, and can be operated either as the "system compiler" or as a user program.

Unlike most UCSD Pascal compilers, the Modula-2 compiler is non-swapping; the compiler code remains resident in memory, allowing faster compilations.

Modula-2 source programs are translated into executable p-Code files. No linking is necessary, as separately compiled modules are automatically bound together at run time.

4.1.1 Invoking the Compiler

The Modula-2 compiler is a code file named SYSTEM.COMPILER. It is invoked by typing "C" at the system prompt.

Input and output file prompts are similar to those of the UCSD Pascal compiler. If a work text file exists, the compiler immediately begins compiling it; otherwise, the following prompt appears on the screen:

```
Compile what file?
```

After the input file name is entered, the output file prompt appears:

```
To what file?
```

The proper response to this prompt depends on the type of module being compiled. If the compilation unit is a program module, the output file prompt works as in UCSD Pascal; <cr> specifies the work file, "$" denotes the same file title as the input file, and a file title creates an explicitly named code file.

When compiling a definition module or implementation module, the compiler ignores the output prompt response and names the output file according to the module name and type; in these cases, responding with a carriage return is sufficient. The output file is written to the prefixed volume.

Compiler-generated output file names consist of a module name (truncated to a maximum of ten characters) followed by a file type suffix. The suffix ".SYM" is appended to the module name if the compilation unit is a definition module. The suffix ".MOD" is appended if the compilation unit is an implementation module. For example, compiling a definition module named "Foon" produces a disk file named "FOON.SYM". This naming convention is required by both the compiler and loader (see 2.1.5 for details). The Modula-2 compiler can also be executed as a user program. Its file name is usually M2.CODE. When executed, the compiler's initial console display appears first, followed by an input file prompt. The work file is ignored. An output file prompt does not appear; the compiler automatically names the output file as if '$' had been typed to an output prompt.

4.1.2 Console Display

Here is an example of a console display:

```
Modula-2 Compiler by Volition Systems
Version p-Code II.2  0.3a  11 Sep 82
Copyright 1982.  All rights reserved.

MODULE Test;

*     2  InOut                 [6221]
*     3  SystemTypes           [5804]
*     3  Program               [5722]
*     4  Files                 [5188]
*****   2247
      9  Proc1                 [3267]
     18    Mod1              3 [3132]
     20    Proc2             5 [3175]
     22    Proc3             4 [3233]
     24  Test                6 [3296]

27 lines, 3132 words left
82 bytes generated
```

The first line indicates the name and type of the compilation unit.

(If Test were a definition module, the line would appear as "DEFINITION MODULE Test"; if an implementation module, "IMPLEMENTATION MODULE Test".)

The compiler prints a line whenever it finishes importing a libary module or encounters the beginning of a procedure or module body. An asterisk in the leftmost column indicates an imported library module. The second column displays the current line in the source program. The third column displays the name of the procedure or module; the indentation indicates the lexical level of the procedure or module. (Note that nested procedures increase the lexical level, but nested modules do not.) The fourth column displays the procedure numbers assigned to procedures and module bodies; these are useful for locating execution errors (see 4.4.6 for details). The rightmost column [enclosed in brackets] displays the number of words left in memory.

At the end, the compiler prints the total number of source lines compiled, the minimum number of memory words available during compilation, and the number of bytes in the produced code segment.

NOTE- When the recycling option is enabled (3.3.10), the compiler prints a row of five asterisks on the screen: each asterisk indicates the completion of a recycling phase. Following the last asterisk is an integer value indicating the number of words reclaimed by recycling. The normal console display resumes when recycling is completed.

4.1.3 Error Handling

The compiler handles two kinds of errors: syntax errors and library errors. Syntax errors are handled as in UCSD Pascal; the user is prompted to either abort the compiler, resume compilation, or invoke the editor. Library errors are always fatal; the compiler prints a message and then terminates.

When the compiler discovers a syntax error, a prompt appears on the console display:

```
IMPORT a,b
EXPORT <<<<
Line 77, ';' expected
Type <escape>, E(dit, <sp>(continue)
```

The first two lines display the source text where the error occurred; the arrows point at the last symbol compiled. The third line displays the current line number and a textual error message describing what is wrong. The last line is a prompt line indicating your available options.

Typing <space> resumes compilation; the compiler usually recovers from the syntax error, and proceeds either to the next error or to the end of the program.

Typing <escape> terminates compilation and returns control to the system prompt line.

Typing "E" terminates compilation and automatically invokes the editor. When the proper input file is specified, the editor positions the cursor at the location of the error and again prints the message describing the syntax error.

Textual error messages are displayed only when the file "SYSTEM.SYNTAX" is on the system (boot) disk volume; otherwise, the compiler only displays the error number. Appendix 3 contains a list of compiler syntax error messages and their assigned numbers.

NOTE- The Advanced System Editor (ASE) works with both Modula-2 and UCSD Pascal "SYSTEM.SYNTAX" files; however, the standard editor does not work correctly with the Modula-2 SYSTEM.SYNTAX. If you are still using the standard editor (or if you regularly use both compilers), change the name of the Modula-2 error file to "MODULA.SYNTAX". The Modula-2 compiler will still display textual error messages, but the editor will display only error numbers.

NOTE- On fatal errors, the <space> option is not available; the only options are to enter the editor or terminate compilation.

Library error messages are handled differently than syntax error messages and are displayed in a variety of formats. Common library errors include:

- Missing library files.
- Invalid library files (usually an improperly updated system library file).
- Module version errors.
- Failure to assign a segment number to a definition module.
- Occurrences of variable or procedure declarations in "compile time" (segment 1) definition modules.

NOTE- All syntax errors become "fatal" errors if they occur while compiling a symbol file. This will not happen under normal circumstances.

NOTE- When the compiler is operated as a user program, the E(dit option is not available. To locate a syntax error in this case, note the line number and error; next, enter the editor with the source file, move the cursor down by the right number of lines, and you should end up right next to the syntax error.

4.2 Executing Programs

Modula-2 programs are executed identically to UCSD Pascal programs. Type "X" from the system prompt. The following prompt appears:

```
Execute what file?
```

Type the file title of the Modula-2 program's code file, then a carriage return. After a few disk accesses, the program is loaded and begins executing. Modula-2 programs generally take longer to load than Pascal programs, as library modules must be located in the library and then loaded into memory along with the program module code.

Modula-2 programs may fail to execute because of a load error; in this case, the loader prints an error message explaining the problem and returns control to the system prompt.

NOTE- Section 4.4.6 describes execution errors and how to locate them.

NOTE- The execution of Modula-2 programs from the system prompt is implemented as a three-step process. The compiler automatically writes a 2-block header on the front of every program module's code file; the header contains a small bootstrap program which is executable from the system prompt. When a Modula-2 program is executed, the system loads and executes the bootstrap program, whose primary task is to find the loader module in the system library, load it, and call it. The loader then goes back to the executed code file, loads the program module (and all of its imported library modules), and calls the program module.

4.3 Library Management

This section describes how to manage the Modula-2 library. Topics covered include operation of the library manager utility, how to use the library manager to update the system library file and create program libraries.

4.3.1 Using the Library Manager

The library manager is a utility program which is used to manipulate library files. Common operations on library files include inserting and deleting library modules from the system library file, and combining user library files into a single program library file.

The library manager also has the ability to "hide" modules stored in a library file. Hiding a module is equivalent to deleting it from the library, but holds a couple of advantages over outright deletion. The entire library file need not be updated to hide a module; hiding is a faster operation than deletion. Also, a hidden module is only temporarily deleted; the module can be subsequently restored by "unhiding" it. Hiding unused system modules improves system efficiency by reducing the amount of memory-resident library information. Hiding is also a useful program development tool (see 4.4.4 for details).

WARNING- The rest of this section is best read in front of the terminal with the library manager program running; otherwise, it is tedious!

The library manager is invoked by X(ecuting the code file named LIB. After a few disk accesses, the following prompt appears:

Lib: U(pdate C(reate S(tatus Q(uit <esc> [8427]

This prompt marks the outer level of the library manager. Q(uit and <esc> return you to the system prompt. S(tatus displays the modules in a library file and is used to hide and unhide modules. C(reate makes a new library file by copying modules from existing library files. U(pdate inserts and deletes modules in an existing library file.

The commands Q(uit and <esc> appear on most every prompt in the library manager. Q(uit terminates the current command level and updates the library file to reflect the changes made at that level. Typing <esc> terminates the current command level without updating the file.

<esc> always issues a prompt when it is typed:

Are you sure?

Typing "Y" escapes the current command level; typing any other character returns you to the current command level.

The integer value enclosed in brackets also appears on many library manager prompts; it displays the number of available words left in memory.

NOTE- Whenever a file name prompt appears in the library manager, you can type '*' to specify the system library file. The library manager recognizes '*' as a shorthand form of the system library file — the file name "MODULA.LIBRARY" is searched for on the prefixed volume, then on the system volume.

S(tatus

Type "S", and the following prompt appears:

Name of file for Status?

Type the complete file name of the library file you wish to modify, then type <return>.

A new prompt line appears across the top of the screen:

```
Status: H(ide U(nhide L(ib Q(uit <esc> P(rev N(ext  [8244]

                         Output file: *MODULA.LIBRARY
# Mod Name Seg               DLen  ILen
1 Program    2   D    I   674  1628
1 SubProgram 3        I        2304
1 Storage    4   D    I   234   854
2 SystemType 1   D    I   205     0
3 UnitIO     1   D    I   843     0
4 BlockIO    1   D        717
5 ASCII      1        I           0
6 MyProg    10        I        1066
7 InOut     54   DH   IH  574   930
8 MathLib0  50   DH       282
```

The output file name is identical to the input file name you typed earlier, as only the library file's attributes are being modified.

The table starting on the next line is known as the **module display**; it displays all the modules contained in the library file. The numbers beneath the pound sign "#" are used to select modules in various library manager commands. The module names are truncated to ten characters.

The "Seg" field displays the module segment number. Segment numbers worthy of note here are 1 (indicating compile-time modules), 2 and 3 (assigned to the loader and its subprogram), 4 (storage manager), and 7 (assigned only to program modules, which are treated as implementation modules by the module display).

The next two fields indicate whether the displayed library module includes a definition module, implementation module, or both. The letter "D" appears if the definition module is in the library, while "I" denotes the presence of an implementation module. If either of these letters is followed by "H", the indicated module is currently hidden (i.e. inaccessible by the compiler and loader).

The last two fields indicate the module sizes (in bytes). An implementation module size indicates the size of the module's code segment. A definition module size indicates the number of bytes in the symbol file. Values are displayed as (decimal) integers. Note that the implementation module size is 0 for "compile-time" modules.

Up to 40 module entries can be displayed on the screen at one time. If the library contains more than 40 module entries, use the N(ext and P(rev commands to move between module displays. N(ext displays the next 40 module entries in the library, while P(rev displays the previous 40 module entries. Note that P(rev and N(ext have no effect unless the current library file contains more than 40 modules.

NOTE- Be sure you understand the module display (and its associated commands), as it appears throughout the library manager.

S(tatus commands

H(ide and U(nhide produce the following prompt:

Which module #, A(ll, <esc>?

Typing a module entry number followed by <return> specifies a single module entry. "A" specifies all module entries in the current module display.

Once one (or all) module entries have been specified, this prompt appears:

D(ef, I(mp, B(oth, <esc>?

D(ef (un)hides the module entry's definition module, I(mp (un)hides the implementation module. B(oth (un)hides both modules. After you respond to this prompt, the module display is updated to reflect the new status of the specified module(s); the "H" character either appears or disappears from the

"D" or "I" indicating the specified module(s).

L(ib assigns new library numbers to system library files. The following prompt appears:

Current Lib # = 1957
New Library # ?

The value displayed on the first line is the library number stored in the current system library file. The new library number is entered and then terminated by typing <return>.

Library numbers are used to uniquely identify system library files originating from different systems; they can be used as "system identifiers" when program development is distributed across a number of different systems. The library number becomes a part of the module key assigned to every definition module compiled on a given system.

NOTE- Any file created for use as the system library file MODULA.LIBRARY **must** have the L(ib command performed on it. The compiler and loader will not open a system library file if has not been assigned a library number.

C(reate

Type "C" from the library manager's outermost prompt line, and the following prompt appears:

Name of file to Create?

Type the complete file name of the library file you wish to create, then type <return>.

A new prompt line appears across the top of the screen, and the module display (described above) is displayed below it:

Create: C(opy L(ib Q(uit <esc> P(rev N(ext [7777]

The output file name displayed on the screen is the name of the file you wish to create. The module display is initially empty; the C(opy modules command fills up the module display with modules copied from other library files.

The C(opy modules command produces the following prompt:

Name of file to Copy from?

Type the complete file name of the library file you wish to copy modules from, then type <return>. A new prompt line appears across the top of the screen:

Copy: S(elect D(eselect H(ide Un(hide Q(uit <esc> P(rev N(ext

The current module display is replaced with the module display of the library file just specified. Modules are selected from this module display for copying into the new library file's module display.

Modules are selected for copying with the S(elect and D(eselect commands. S(elect and D(eselect produce the following prompt:

Which module #, A(ll, <esc>?

Typing a module entry number followed by <return> specifies a single module entry. "A" specifies all module entries in the current module display.

Once one (or all) module entries have been specified, this prompt appears:

D(ef, I(mp, B(oth, <esc>?

D(ef (de)selects the module entry's definition module, I(mp (de)selects the implementation module. B(oth (de)selects both modules. After you respond to this prompt, the module display is updated to reflect the modules that have been selected for copying; the presence of an asterisk "*" preceding a "D" or "I" in the module display indicates that the corresponding module has been selected for copying.

NOTE- The S(elect command automatically selects the library module contained in a one-module library file; the additional prompts do not appear.

Typing Q(uit from the Copy prompt copies the selected modules into the new library file. The new library file's module display reappears on the screen; it now includes the modules copied from the last library file. The Create prompt line also reappears, allowing you to use the C(opy modules command to copy additional modules from other library files.

Typing <esc> returns to the Create prompt without copying any modules.

NOTE- The library manager performs some library integrity checks normally performed by the compiler and loader; in particular, it prevents the following situations from occurring:

- Inclusion of duplicate module names into a library file.
- Inclusion of a definition or implementation module when its companion module has an incompatible module key.
- Inclusion of a module that references other modules in the library file when the module keys do not match.

NOTE- The available memory (as displayed on the prompt lines) shrinks as additional modules are copied into the new library file. The library manager can build libraries containing arbitrary numbers of modules, but limits the total size of a library file to what can fit in memory at one time. The memory available value indicates when this limit is approached; if it gets close to 0, do not copy any more modules into the library!

To leave the C(reate command, type Q(uit or <esc>. Q(uit finishes writing the output file and saves it on disk. <esc> exits the C(reate command, but purges the output file. After both commands, control is returned to the library manager's outer prompt.

U(pdate

The U(pdate command is similar to C(reate, but is used to change the contents of an existing library file. U(pdate first prompts for the input and output file names; typing a "$" as the output file name causes the output file to have the same name as the input file. After the output file is specified, U(pdate automatically performs a C(opy and S(elect of all modules in the input file. Modules are then removed with D(eselect or added by C(opying modules from other library files. U(pdate also performs an automatic L(ib operation on the output file, assigning it the library number from the input file; thus, L(ib need not (and in fact should not) be invoked.

4.3.2 Updating the System Library

The system library file is usually updated with the library manager commands S(tatus and U(pdate. (C(reate is generally used only for building program libraries.)

S(tatus is used to hide modules or change library numbers. Note that S(tatus is a much faster operation than U(pdate. S(tatus does not copy the existing library to a new file; it merely updates information in the current library file.

U(pdate is used to insert or delete modules from the system library. The output file is named "MODULA.LIBRARY"; the output file thus replaces the old system library file when it is written out to disk.

A system library file must contain the modules Program, SubProgram, and Storage — the L(ib command will not work without them, and a system library file is unusable if not assigned a library number. Note that hiding has no effect on their implementation modules.

WARNING- Do not hide or delete library modules used by the library manager: Terminal, BlockIO, Screen, ASCII, Conversions, UnitIO, and Standards. If you do, the library manager cannot be invoked again, making it impossible to subsequently update the library (in particular, unhiding the unwisely hidden modules). If this does happen, you will have to transfer a new copy of the system library file from your backup disks.

4.3.3 Creating Stand-alone Programs

Program files (2.1.4) are code files that contain a program module and all of its subsidiary library modules (possibly including subprogram modules). Program files are not truly stand-alone programs; the system library file must still reside separately on disk (in the file MODULA.LIBRARY).

C(reate is used to collect modules from several user library files into a single program file. Program files are usually — but not necessarily — named after the (main) program module contained within. Remember to add the ".CODE" suffix to the output file name.

In order for a program file to work correctly, the "main" program module must be the **first** module copied into the output library file. The loader assumes that the last program module in the file — i.e. the first module copied into the library file — is the one to be called.

NOTE- See 4.4.2 for details on structuring large programs into program libraries.

4.4 Programming Techniques

This section describes techniques which improve the effectiveness of the Modula-2 system. It covers the following topics:

- Maximizing compile-time space
- Maximizing run-time space
- File naming conventions
- Using the library
- Accessing machine-level operations
- Locating execution errors

Sections 4.4.1 and 4.4.2 describe techniques for making programs compile and execute as efficiently as possible.

Section 4.4.3 suggests some naming conventions for organizing the source files of library modules.

Section 4.4.4 explains how to make efficient use of the library system.

Section 4.4.5 explains how to take advantage of set operators and type transfer functions to obtain some useful low-level operations.

Finally, section 4.4.6 explains how to use execution error messages and compiled listings to track down execution errors.

4.4.1 Maximizing Compile-time Space

Modula-2's separate compilation facilities play an important role in the development of large programs on limited-resource machines. On such systems, the compiler tends to occupy most of available memory, leaving relatively little space for symbol table storage and thus limiting the size of a compilation unit. Languages such as Pascal equate compilation units with programs, thereby imposing strict limits on the size of a program. Modula-2's ability to construct programs from separate compilation units allows the development of much larger programs than can be written in Pascal.

NOTE- It is worth noting that the use of modules to restrict the visibility of identifiers — presented earlier in this manual as a technique for improving program understandability — proves to be an implementation asset by reducing the demands on symbol table storage.

The Modula-2 language offers the possibility of significantly increasing the amount of available symbol table space. First, a compiler can recycle symbol table storage allocated for a local module's private variables once the module has been compiled. Second (and more important), after compiling the symbol file of an imported definition module, a compiler can recycle the storage allocated for symbols not actually imported by the client module. The Modula-2 compiler provided with this system is capable of performing the latter space optimization.

Storage reclamation is controlled by the compile option $RECYCLE (3.3.10). Recycling can create up to 2000 extra words of compile space at the expense of slightly slower compilations (due to the extra time spent in the recycling phase).

Definition module recycling can significantly affect the design of large compilation units which import identifiers from many different library modules. A definition's modules symbols are recycled **only** if the module is subjected to unqualifying import. In unqualifying import, the compiler retains the symbol records of identifiers actually imported from the module; the remaining definition module identifiers are disposed of. If a library module is imported by name, then all of its (exported) identifiers are potentially accessible and therefore cannot be recycled. In short, recycling is most effective when unqualifying import is used on all library modules.

Another factor affecting compile-time symbol table space is the number of modules in the system library. The compiler maintains information in memory describing all modules in the system library (2.1.5); thus, very large system libraries may consume nontrivial amounts of compile-time storage. The recycle option reclaims library information along with unused library identifiers, so this is ultimately not a serious problem; however, the need for recycling can be minimized by limiting the system library to only the modules needed for a particular system configuration; this not only saves compile space on (nonrecycling) compilations, but makes system library updating more efficient (because of the smaller library file).

4.4.2 Maximizing Run-time Space

Large programs usually consist of a number of code "overlays" in order to reduce the amount of memory occupied by program code. For instance, UCSD Pascal provides "segment" procedures whose code remains disk-resident until they are called. Code management in Modula-2 is performed by dividing a large program into a collection of subprograms, and using the loader to execute subprograms as procedures. Unlike segment procedures, Modula-2 subprograms are complete programs; they can in fact be executable programs themselves. Note that subprograms may share the system's dynamic storage, thus allowing a subprogram to build dynamic data structures that are available to the main program after the subprogram terminates. Subprograms communicate with calling programs by importing the same library modules and sharing their variables.

The subprogram call concept has a number of advantages over the use of segment procedures. Subprograms can be written and tested as executable programs before being incorporated into their host program, thus making possible a true "building block" approach to the design of large software systems. Shell/menu programs can be written which prompt for the name of a Modula-2 program module (or Pascal program), execute it, and then redisplay the command prompt.

Because programs are independent with respect to segment assignment, a single program can call arbitrarily many subprograms (eliminating UCSD Pascal's "ran out of segments" syndrome). In very large programs, the subprogram call concept can be more efficient than than having a single program with many segments. A program containing large numbers of segments requires segment information to be memory-resident for the life of the program, whereas subprogram calls accumulate memory-resident segment information only on nested subprogram calls (when existing segment entries are temporarily displaced by a subprogram's segments).

The optimal structure for large Modula-2 systems is a base program which serves as a global environment for its subprograms (and as a "shell" or "menu" to the user) and a large number of independent subprograms, each representing a different function within the system.

The primary disadvantage of subprogram calls is that they usually require more disk accesses than a corresponding segment procedure call. The library system is designed to minimize this problem; in particular, it allows for differing uses of subprograms. Subprograms are generally used in one of two ways: either as **segment programs** (purposely similar to "segment procedures"), or as **subsystems.**

Subsystems have the following properties: they perform well-defined functions, they are substantial programs with a number of subsidiary library modules, and they are called relatively infrequently. A typical example would be the role played by the library manager. From the system's point of view, the extra disk accesses required to load a subsystem are acceptable losses in system performance given the advantages of the subsystem concept.

The proper way to structure a subsystem is to bind the program module and all of its subsidiary modules into a single program file. The structure of the software system is reflected in the way it is stored in the library: a central code file executed as a "shell", and a collection of program files each of which represents a different subsystem. On subsystem calls, the loader searches the disk only once for the subprogram file; since subsidiary modules are contained in the same file, they load relatively quickly.

Segment programs have the following properties: they are localized to a single program, and they are called relatively frequently. Segment programs do not have any subsidiary library modules, as they usually serve subsidiary roles themselves; e.g. initializing the global data structures of a host program. From the system's point of view, it is desirable that they are loaded and unloaded as quickly as possible to minimize the (inevitable) degradation in system performance caused by loading a code file from disk.

The proper place for a segment program is in the program file of its host subsystem; here it serves its role as a subsidiary module. Because the library information describing subsidiary modules is memory-resident during the life of the host subsystem, the subprogram call requires no disk accesses other than reading in the code segment itself — it thus loads as quickly as a UCSD Pascal segment procedure.

The Modula-2 system also provides a feature for simplifying the storage management in large systems. Because the system provides true dynamic storage management, it is necessary to explicitly deallocate all dynamically allocated variables. This requirement can be a severe limitation when it is necessary to reclaim storage allocated by subprograms that have to construct complex dynamic structures. To address this problem, subprogram calls have the option of controlling whether the called program shares dynamic storage with the calling program. On shared subprogram calls, all structures created by the subprogram are retained when the subprogram terminates. On unshared calls, all storage allocated by the subprogram is automatically deallocated upon subprogram termination. See **Standard Library** for details.

Another factor affecting run-time storage space is the presence of in-memory library information maintained by the loader. The loader keeps information in memory describing all modules in the system library; thus, very large system libraries may consume nontrivial amounts of run-time storage. In most cases, it is desirable to pare the system library down so it contains only the modules used by a particular system configuration. Library information is also minimized by maintaining the subprograms of a very large program in separate program files rather than lumping them together into a single (and possibly huge!) program file.

4.4.3 File Naming Conventions

A library module X consists of up to four disk files: two text files (containing definition and implementation module sources), a symbol file, and an object file. The compiler automatically assigns the file names "X.SYM" and "X.MOD" to the symbol and object files; however, the user is responsible for naming the text files. A problem arises because there are two text files meriting the file name "X.TEXT". A standard convention is to name the definition module source file "XD.TEXT" and the implementation module source file "X.TEXT". These names are distinct, but similar enough to relate the files (and to manipulate them in the filer as the wildcard entity "X=TEXT").

NOTE- Compiler-assigned output file names (i.e. files with the suffixes ".SYM", ".MOD", and ".CODE") should not be changed, as they are an essential part of the library access scheme.

NOTE- Given the proliferation of disk files resulting from library modules, the system library file is seen to serve as a facility for reducing the number of disk directory entries used up by collections of library modules.

4.4.4 Using the Library

The system library provided with the Modula-2 system contains all library modules provided with the system. As a result, the library file is rather large and unwieldy: it consumes a lot of disk space, slows down the compiler and loader, and reduces the amount of compile-time and run-time space available. Fortunately, you can make the whole system more efficient and easier to use by removing unused library modules from the system library; the smaller the library is, the better the system runs.

Here is how to tailor the Modula-2 system to your needs:

- Save a copy of the original system library file on an archive disk (to ensure that you never accidentally delete the only copy of some library module).
- Learn how to use the library manager.
- Determine which library modules you will be using for your next programming project. (e.g. "Do I really need decimal arithmetic in my real-time turboincabulating system?")
- Check the module hierarchy (in the **Implementation Guide**) to see if the library modules you have chosen happen to import some other library modules — you will need these, too.
- Use the library manager to construct a new system library file containing only the library modules you need. Put the new system library file on your system disks and start programming!
- When your program is done, remember that definition modules can be deleted from the library without affecting program execution; they are needed only for compilation.

The library manager allows modules stored in the system library file to be "hidden" so they cannot be accessed by the compiler or loader. You can take advantage of the fact that hiding a module in the system library is a much faster operation than updating the system library with a new library module. If a module stored in the system library is found to be incorrect (and thus requires updating and testing), it is faster to hide the existing module and perform all updates and testing with the user library. Only after the module is performing correctly need it be inserted into the system library to replace the old (hidden) version. Section 2.1.6 explains how to make efficient use of the library during program development.

The compiler and loader check for a system library file on the prefixed volume before checking the system volume. If your system is configured to operate with the system library file on the system volume, you can exploit this property of the library search algorithm (2.1.5) to test new versions of the system library without having to disturb the existing system library file. When the system library file is updated, write it to the prefixed volume with the standard name "MODULA.LIBRARY". When the next Modula program is executed, it will access the system library file on the prefixed volume. If the new system library is correct, delete the old system library file and transfer the new file to the system volume. If the new system library is nonfunctional, just delete it — the system is now back to its original state.

NOTE- Programs load slightly faster if the system library file resides on the same disk as the user library and program library files; thus, it is usually preferable to keep the system library file on the prefixed disk along with the rest of the library.

4.4.5 Accessing Low-level Machine Operations

The type conversion functions allow access to a few useful machine-level operations. The full-word logical operations AND, OR, XOR, and bit masks are achieved with Modula-2's set operators. Operands must be converted to type BITSET.

set operator	machine operation
union "+"	logical OR
intersection "*"	logical AND
symmetric difference "/"	logical XOR
difference "-"	bit mask

Multi-word comparisons for equality and inequality are achieved by converting the operands to sets (of corresponding length) and using the set operators "=" and "#".

Before using sets for multi-word operations, be aware that the maximum set size varies across implementations. On 6502's, the maximum set size is 32 words; the remaining processors allow full 255 word sets. Multi-word comparisons are not advisable on packed records and arrays, as most packed structures contain unused (and thus uninitialized) bit fields which prevent accurate comparisons.

Character variables declared at fixed addresses are accessed as byte quantities. This allows access to individual bytes without disturbing adjacent memory — a necessary attribute for accessing such objects as bank switches and I/O registers.

4.4.6 Locating Execution Errors

Execution errors are handled either by the calling program or (in the default case) by the system. Since program-controlled execution error handling indicates only the occurrence of an error and not its actual location, execution errors are best handled by the system.

On a system-controlled execution error, the system terminates the program and displays a multi-line error message describing the error. The first line of the message indicates the location of execution error. Successive lines display the procedure **call chain**; i.e. the series of procedure calls leading from the error back to the outer block of the program. (Procedure call chains are commonly known as "walkbacks".) Following the call chain is a textual error message and a prompt line.

Example of an execution error message:

```
>M=MyModule, P=23, I=432
^M=BasicMod, P=5,  I=18
^M=MainProg, P=12, I=174
^M=MainProg, P=2,  I=291
Range Error, type <space>
```

Typing the space bar terminates the current program and returns control to the calling program.

Note that the messages describing the execution error location and call chain have similar form. Each consists of a module (compilation unit) name, procedure number within the module, and code offset within the procedure. The character '>' marks the message describing the execution error location. The character '^' marks messages describing the location of each procedure call in the call chain.

To find the locations of error and procedure calls within your source programs, you must match the procedure numbers and code offsets in the error message with the values displayed in compiled listings (see 3.3.5 for details).

Thus, the first thing to do is find (or create) up-to-date compiled listings of the named modules. Procedures are located by matching the procedure number with the procedure number on the listing; similarly for the code offset. Note that in many cases the displayed code offset does not match any value displayed in the listing; this is because each source statement produces arbitrary numbers of bytes of code. To track down the erroneous source line, find the largest code offset value which is smaller than the displayed offset value.

To simplify the pursuit of execution errors, the compiler provides the compile option $DEBUG (3.2.13). When a module is compiled with DEBUG set to TRUE, the execution error message displays the procedure name in place of the procedure number. With symbolic procedure names in the call chain, it becomes possible to trace execution errors without using compiled listings.

Example of a symbolic execution error message:

```
>M=MyModule, P=BigProc, I=432
^M=BasicMod, P=StartSort, I=18
^M=MainProg, P=InitFiles, I=174
^M=MainProg, P=MainProg, I=291
Range Error, type <space>
```

Note that an execution error message may include both procedure names and procedure numbers. Because symbolic procedure information can be specified on a per-module (or even per-procedure) basis, error messages display procedure names only if they are available in the code file.

NOTE- The maximum call chain depth displayed is 10 calls.

NOTE- Sometimes an execution error may occur within a standard or utility module; although compiled listings of these modules are unavailable, the call chain should lead back to one of your own modules. The most likely causes of such errors are invalid parameters passed to system routines.

NOTE- Execution error messages do not appear (in system-controlled mode) if a subprogram terminates with program results NormalReturn or ProgramHalted. The standard procedure HALT terminates a program with program result ProgramHalted.

NOTE- If an execution error occurs in a called Pascal program, the error message displays segment numbers in place of module names. Note that Pascal program outer blocks are not displayed in the call chain.

Appendix 1 Module Segment Numbers

The following table indicates the segment numbers assigned to modules contained in the system library. Segments are assigned so that the most frequently used library modules reside in the highest numbered segments.

Segments 0 through 6 are reserved for the Modula-2 system. Program modules always reside in segment 7. Segments 8 through 47 are available for user-defined library modules. Segments 48 through 63 are provisionally reserved for library modules provided with the system.

Many of the standard library modules are interdependent; importing one of these modules implies the importation of other modules, resulting in the use of extra segments. **Standard Library** describes library module dependencies.

The utility module ASCII and all system-dependent modules are compile-time modules, and thus are assigned segment 1.

NOTE- Segments 59 through 63 are reserved for certain implementations. See the **Implementation Guide** for details.

Library module segment assignment:

Program	2	Reals	52
SubProgram	3	Strings	53
Storage	4	InOut	54
Decimal	48	Conversions	55
Processes	49	Texts	56
MathLib0	50	Files	57
RealInOut	51	Terminal	58

Appendix 2 Compiler Directives

$DEBUG	Setting DEBUG to TRUE causes the compiler to emit code file information which enables execution error messages to display symbolic procedure names. The default setting is FALSE.
$ELSIF	"$ELSIF <expression> THEN" marks an alternate choice of a conditionally compiled section of program text. The following text is compiled only if the previous section of text was not selected and the expression evaluates to TRUE.
$ELSE	Marks the default part of a conditionally compiled section of program text. The following text is compiled only if the previous section(s) of text was not selected.
$END	Marks the end of a conditionally compiled section of program text.
$FLIP	Setting FLIP to TRUE causes the compiler to generate code files for processors of the opposite byte-sex. The default setting is FALSE. Set at top of program.
$IF	"$IF <expression> THEN" marks the start of a conditionally compiled section of program text. The following text is compiled only if the expression evaluates to TRUE.
$IN	The following string contains the name of a text file to be included into the program text.
$LIST	Setting LIST to TRUE generates a compiled listing. FALSE suppresses listing. The default setting is FALSE. The TO option must be used for LIST to have any effect.
$NOT	The following string is embedded in the object or symbol file as a copyright notice. NOT must be set after the initial module heading.
$POP	Sets the specified Boolean variable to its previously stacked value.
$PUSH	Stacks the current value of the specified Boolean variable.

$QUIET	Setting QUIET to TRUE suppresses the compiler's console display. The default setting is FALSE.
$RANGE	Setting RANGE to TRUE generates runtime range checks on array and subrange references. FALSE suppresses checks. The default setting is TRUE.
$RECYCLE	Setting RECYCLE to TRUE enables the symbol table recycling phase in the compiler. The default setting is FALSE. Set at top of program.
$SEG	Assign the module segment number to the definition module. Set after module heading.
$SET	If a string parameter is provided, it is written to the screen as a prompt; otherwise, the compile-time variable name is printed on the screen followed by a '?'. Typing 'y' or 't' sets the variable to TRUE; 'n' or 'f' sets it to FALSE. Cardinal numbers may also be entered.
$SPECIAL	Setting SPECIAL to TRUE alters the Modula-2 vocabulary for systems with half ASCII character sets. The default setting is FALSE. Set at top of program.
$STANDARD	Setting STANDARD to FALSE allows the use of Modula-2 language extensions. The default setting is TRUE. Set at top of program.
$TO	Writes a compiled listing to the file named by the string parameter. Sets LIST to TRUE. Set at top of program.
$TYPE	The string parameter is written to the screen.
$UPCASE	Setting UPCASE to TRUE sets SPECIAL to TRUE and translates all lower case characters to upper case. The default setting is FALSE. Set at top of program.

Appendix 3 Compiler Error Messages

1: Non-standard construct
2: Constant out of range
3: Open comment at end of file
4: String terminator not on this line
5: Too many errors
6: String too long
7: Copyright must appear after MODULE and before PROCEDUREs
8: IO error on output code file
9: Unable to open include file
10: IO error opening output code file
11: Illegal character in text
12: Unexpected end of source file
13: IO error reading System Library
14: Illegal System Library
15: IO error on listing file
16: Error in $ directive
17: $ options must be more than 1 letter long
18: $IF not closed with $END at end of file
19: $ELSIF, $ELSE, or $END without previous $IF
20: Identifier expected
21: Integer constant expected
22: ']' expected
23: ';' expected
24: Block name at the END does not match
25: Illegal declaration or block terminator
26: ':=' expected
27: Error in expression
28: 'THEN' expected
29: Error in LOOP statement
30: Constant must not be CARDINAL
31: Error in REPEAT statement
32: 'UNTIL' expected
33: Error in WHILE statement
34: 'DO' expected
35: Error in CASE statement
36: 'OF' expected
37: ':' expected
38: 'BEGIN' expected
39: Error in WITH statement
40: 'END' expected
41: ')' expected
42: Illegal component in constant
43: '=' expected
44: Error in TYPE declaration
45: '(' expected
46: 'MODULE' expected

47: 'QUALIFIED' expected
48: Illegal expression component
49: Illegal simple type
50: ',' expected
51: Illegal formal parameter type
52: Illegal statement starter or missing END
53: '.' expected
54: Export at global level not allowed
55: Body in definition module not allowed
56: 'TO' expected
57: Nested module in definition module not allowed
58: '}' expected
59: '..' expected
60: Error in FOR statement
61: 'IMPORT' expected
62: 'DEFINITION', 'IMPLEMENTATION', or 'MODULE' expected
63: Error reading source file
64: IMPLEMENTATION not allowed for Pascal ($CODE) modules
70: Identifier specified twice in importlist
71: Identifier not exported from qualifying module
72: Identifier declared twice
73: Identifier not declared or incorrect class
74: Type not declared
75: Identifier already declared in module environment
78: Value of absolute address must be of type CARDINAL
79: Scope table overflow in compiler
80: Illegal priority
81: Definition module not found
82: Structure not allowed for implementation of hidden type
83: Procedure implementation different from definition
84: Not all defined procedures or hidden types implemented
86: Incompatible versions of symbolic modules
88: Function type is not scalar or basic type
90: Pointer-referenced type not declared
91: Tagfieldtype expected
92: Incompatible type of variant-constant
93: Constant used twice
94: Arithmetic error in evaluation of constant expression
95: Range not correct
96: Range only with scalar types
97: Type-incompatible constructor element
98: Element value out of bounds
99: Set-type identifier expected
101: Exported items were never defined
102: Forward procedures were never defined
103: Wrong class of identifier
104: No such module name found
105: Module name expected
106: Scalar type expected
107: Set too large
108: Type must not be INTEGER or CARDINAL

109: Scalar or subrange type expected
110: Variant value out of bounds
111: Illegal export from program module
120: Incompatible types in conversion
121: This type is not expected
122: Variable expected
123: Incorrect constant
124: No procedure found for substitution
125: Incorrect procedure call terminator
126: Set constant out of range
127: Error in standard procedure parameters
128: Type incompatibility
129: Type identifier expected
130: Type impossible to index
131: Field not belonging to a record variable
132: Too many parameters
134: Reference not to a variable
135: Illegal parameter substitution
136: Constant expected
137: Expected parameters
138: BOOLEAN type expected
139: Scalar types expected
140: Operation with incompatible type
141: Only global procedure or function
142: Incompatible element type
143: Type incompatible operands
144: No selectors allowed for procedures
145: Only function call allowed in expression
146: Arrow not belonging to a pointer variable
147: Standard function or procedure must not be assigned
148: Constant not allowed as variant
149: SET type expected
150: Illegal substitution to WORD parameter
151: EXIT only in LOOP
152: Incorrect use of RETURN statement
153: Expression expected
154: Expression not allowed
155: Type of function expected
156: Integer constant expected
157: Procedure call expected
158: Identifier not exported from qualifying module
161: Call of procedure with lower priority not allowed
300: Index out of range
301: Division by zero
303: Case label defined twice
404: Too many globals, externals and calls
405: Procedure too long (codetable overflow)
990: IMPORT not allowed in Pascal ($CODE) modules
991: Cardinal divisor too large (> 100000B)
992: FOR control variable must not have byte size
993: Illegal use of byte variable

994: Too many nested procedures
995: FOR step too large (> 77777B)
996: CASE label too large (> 77777B)
997: Illegal parameter substitution
999: Identifier referenced before this definition

Index

Modula-2

Implementation Guide

for the IBM PC

Release:	**0.3**
Date:	**26 August 1983**
Author:	**Richard Gleaves**

Table Of Contents

1 Introduction

This document describes Volition Systems' implementation of Modula-2 for the IBM Personal Computer. It covers the following topics:

- Installation guide
- System configuration
- Formatting new disks
- Machine-dependent modules
- Interrupt system
- Machine-level data representations
- Library module hierarchy

Section 2 explains how to install Modula-2 on your PC. (The most important step here is to back up your distribution disks.)

Section 3 explains how to configure the Modula-2 system to take full advantage of your PC. This step involves deciding how to use available memory and which peripherals (joy sticks, printer, etc.) to support.

Section 4 explains how to initialize new disks for use with the Modula-2 system.

Section 5 describes modules which are specific to the PC implementation of Modula-2. The library modules **IBMStuff** and **SYSTEM86** provide access to low-level system facilities (interrupt system, peripherals, and extended memory access).

Section 6 describes the Modula-2 interrupt system on the PC, including IOTRANSFER vector numbers, module priorities, and how to write interrupt handlers.

Section 7 describes the machine-level data representation of various data types. This information is necessary for performing low-level operations involving type conversion.

Section 8 describes the library module hierarchy. This information is necessary for reconfiguring the library.

2 Installation Guide

This section describes how to install Modula-2 on your PC. Along with this you will need the **IBM Installation Notes** which provide additional installation and configuration details.

Section 2.1 explains the installation procedure.

Section 2.2 presents miscellaneous system information.

Section 2.3 describes the Modula operating system files.

Section 2.4 describes the Modula-2 system files.

Section 2.5 describes the interpreter files.

Modula-2 on the PC is a complete software system based on Volition's Modula operating system. The Modula operating system includes a file manager, text editor, Pascal compiler, and many utility programs. The Modula-2 system includes a Modula-2 compiler, module library, and a library manager.

NOTE- The Modula operating system is described in the **Modula Operating System Manual.** The ASE text editor is described in the **ASE User's Manual.** The **IBM Installation Notes** contain details on installing the system.

2.1 Installation Procedure

Modula-2 for the IBM PC is distributed on four single-sided floppy disks in UCSD p-System format. The four disks are named SYS, LIB, UTIL, and PROGS.

Here is an overview of the installation procedure:

- To start you need five blank diskettes.
- Copy the distribution disks onto four of the blank disks.
- Store the originals in a safe place.
- Initialize the fifth blank disk and name it INTERP.
- Rearrange the files so they are stored on the proper disks.
- The Modula-2 system is now ready for use. SYS and LIB serve as your system and work disks.
- Reconfigure the system to improve performance.

Copy the Distribution Disks

The first and most important thing to do is to copy the distribution disks onto some blank disks. If your distribution disks do not contain write-protect tabs, now is a good time to add them. Insert the SYS disk label side up into the left hand disk drive, close the drive door, and type Ctrl-Alt-Del to start the system. After a few moments and some disk action, the Modula-2 system should display its startup message in the center of the screen and a command promptline across the top.

You have to format the blank disks before you can copy the distribution disks onto them. Execute the program named FORMAT; this is the disk formatter utility program. The prompts it displays are generally self-explanatory, but if you need more information, section 4 explains how to run the disk formatter. Use the right hand disk drive to format the blank disks. Note that the left hand disk drive is called "unit 4" and the right hand drive "unit 5". Note also that you need to know whether your blank disks are single or double sided. Be sure to format all five of the blank disks before proceeding to the next step. The disk formatter assigns the name BLANK to each newly formatted disk.

To copy the distribution disks, execute the program named BACKUP; this is the disk copier utility program. Section 10.1.2 in the **Modula Operating System Manual** explains how to run the disk copier. Note that once the disk copier program is running, you can take the SYS disk out of unit 4. Copy all four of the distribution disks onto blank disks. Be sure to preserve the original disk names; when the disk copier program asks if you want to rename the disks as BACKUP, type 'n'.

When you finish copying the distribution disks, store the originals in a safe place. The disk copies will be used to build your system disk set.

Using Double Sided Disks

If you have double sided disk drives in your IBM PC, continue with these steps.

After you have copied the distribution disks onto double sided disks, you should execute the disk size utility program. This program is named DISKSIZE; it is stored on the UTIL disk. The disk size changes information stored in the disk directory to match the storage capacity of double sided disks.

To execute DISKSIZE, place the UTIL disk in unit 5. The SYS disk is assumed to be in unit 4. Execute the file #5:DISKSIZE — the "#5:" informs the system that the program is stored on the disk in unit 5. When the disk size program asks how many blocks are on the disk, type in 640 — this is the proper size of double sided disks. Be sure to change the size of all the Modula-2 system disks (including SYS itself).

Rearrange Files on the Disks

After creating the system disk set, you must rearrange the files so that related disk files are stored on the same disk. This process converts the system disk set from four to five disks, freeing up space on the SYS and LIB disks for your own files.

With the SYS disk in unit 4, type "F" to start the filer program. When the filer promptline appears across the top of the screen, you can take out the SYS disk.

Put the fifth (formatted) blank disk into unit 5 and use the filer command Change to change its name to INTERP. <ret> denotes the return key:

Change what file? #5:, INTERP:<ret>

Next, put the UTIL disk in unit 4 and use the command Listdir to display the disk file directory:

Dir of what volume? #4:<ret>

The files that appear below the line of dashes are interpreter files. Use the Transfer command to copy the interpreter files to the INTERP disk:

Transfer what file? UTIL:?, INTERP:$<ret>

The question mark causes the filer to ask for each file whether you want to transfer it to INTERP. Type 'N' for all files above (and including) the dashed line, but type 'Y' for the rest. This transfers all the interpreter files to the INTERP disk.

Once you have transferred the interpreter files, you can remove them from the UTIL disk. Use the Remove command to remove all files below (and including) the line of dashes:

Remove what file? UTIL:?<ret>

Using this same procedure, move the utility program files first from the SYS disk to UTIL, then from the LIB disk to UTIL, and finally from the PROGS disk to UTIL.

NOTE- If you are using single sided disks, you will not be able to fit all the utility files onto the UTIL disk. You can either keep the leftover files on another disk (perhaps named UTIL2) or just copy them from the distribution disks when you need them.

The system disks are now completed and ready for use. If you can spare the blank disks, back up the reorganized system disk set; in case anything goes wrong, this saves you the effort of having to reconstruct them from the distribution disks.

Run the Installed System

The Modula-2 system is now ready to use. The normal system configuration is to run with the SYS disk in unit 4 and the LIB disk in unit 5; SYS contains the system files, and LIB is used to store library and program files. **Note that with this configuration the prefixed volume must always be set to unit 5 in order to compile or execute Modula-2 programs.** The filer command Prefix sets the prefixed volume:

Set prefix to ? #5:<ret>

The extra disk space on SYS and LIB can be used either to store program files or frequently used utility programs.

The PROGS disk contains sample programs that demonstrate how to use the Modula-2 language and library modules.

The UTIL disk is used to store infrequently used utility programs.

The INTERP disk is used only when you are reconfiguring the system.

The system is preconfigured to operate within a single 64K byte space, with any remaining memory automatically allocated as a RAM disk. See section 3 for details on how to reconfigure the system to work with larger code spaces and/or different peripheral devices.

NOTE- Perhaps the most rewarding reconfiguration is to run the system off the RAM disk. See 3.4 for details.

NOTE- Section 2.2 contains important operating information.

2.2 System Notes

The Modula-2 system library file MODULA.LIBRARY must reside on either the prefixed disk volume or the system boot volume in order to compile or execute Modula-2 programs.

When using a RAM disk for development work, be sure to occasionally back up your files onto a regular disk. If for some reason the system requires rebooting, the contents of RAM are cleared and any files stored in the RAM disk are lost.

The system and utility programs differ slightly from their standard UCSD Pascal counterparts in a few cases. The biggest difference is that the filer allows you to have two wildcards in a file name — this can be very useful at times. See the **Modula Operating System Manual** for details.

Segment numbers 59 through 62 are available for user-defined library modules on the PC. Segment 63 is used by the library module IBMStuff which controls the screen, joysticks, and calendar clock. These segments are formally reserved for implementation-dependent modules, but they are generally available on all UCSD Pascal based Modula-2 implementations except the Apple ///, which uses them to access the SOS operating system.

SYSTEM.BATCH and SYSTEM.SHELL are utility programs that are invoked by the system commands Batch and Shell. If you aren't using them, you can leave these files off the disk; if you are, they can be stored on any online disk volume and still be invoked from the command promptline.

The shell implements program pipes as intermediate files written to disk. Pipe files are opened with the name "*temp"; therefore, the system boot volume must contain enough free disk space to contain whatever intermediate file the shell generates. This should be accounted for if you plan to use the shell extensively. (If so, running the system from the RAM disk is strongly advised.)

The shell and and the utility program Teletalk are both Modula-2 programs; in order to run them, the Modula-2 system library file must reside on the prefixed or system disk volume.

Process work spaces often have to be larger than expected when the system is configured for separate code and data spaces. The separate code and data interpreter maintains on the stack a pool of all recently assigned string constants. Process work spaces must be large enough to contain this constant pool along with whatever local variables and procedure calls it performs. (Note that the constant pool is cut back on a procedural basis.)

2.3 Modula Operating System Files

This section describes the files that make up the Modula operating system. These files are contained on the release disk set.

SYSTEM.INTERP is the preconfigured p-code interpreter supplied with the system. It must reside on the system boot disk.

SYSTEM.PASCAL is the operating system file. It must reside on the system boot disk. If you plan to operate your PC from an external terminal, the Binder utility BINDER.CODE is used to bind new gotoxy procedures into the operating system.

SYSTEM.MISCINFO is the system information file. It must reside on the system boot disk. This file is preconfigured to work with a graphics card or monochrome display. If you plan to operate your PC from an external terminal, the utility programs SETUP.CODE and CONFIG.CODE are used to reconfigure the system information file for your terminal. Note that SYSTEM.MISCINFO must be properly configured for ASE to work.

SYSTEM.FILER is similar to the standard UCSD Pascal file manager, but offers a more powerful "wildcard" facility. See the **Modula Operating System Manual** for details.

SYSTEM.EDITOR is the ASE text editor.

YALOE.CODE is a line-oriented text editor.

SYSTEM.BATCH is the command interpreter program invoked by the system command Batch. The command file B.DEMO.TEXT demonstrates the use of the command file interpreter.

SYSTEM.SHELL is the "p-NIX" command shell which implements many of the Unix operating system commands and features (ls, grep, cat, pipes, I/O redirection, etc). Note that any program can be named SYSTEM.SHELL and invoked by Shell.

PC.CODE is the Pascal compiler. Note that this file can be changed to SYSTEM.COMPILER if you wish to use the Pascal compiler as the "system" compiler. PASCAL.SYNTAX is the syntax error file for the Pascal compiler. This file should be changed to SYSTEM.SYNTAX (and the existing SYSTEM.SYNTAX to MODULA.SYNTAX) if you plan to use both Pascal and Modula-2 compilers.

BACKUP.CODE is a disk copier utility program. It provides a safe and reliable way to create backup copies of a floppy disk. FCOPY.CODE performs the same function for individual files.

SERTALK.CODE transfers disk files from machine to machine via an RS232 serial line.

TELETALK.CODE is used to send and record text files during electronic mail sessions. The text files TELETALK, SYS.PARM, RAWCON, and REMOTE are the source files for this program. Note that it is written in Modula-2.

PATCH.CODE is a byte-level disk file editor.

FLIPDIR.CODE flips the disk directories of disks created on byte-flipped machines.

COMPARE.CODE compares two text files and reports on any differences. COMPCODE.CODE compares disk files of any type.

COPYDUPDIR.CODE copies the duplicate disk directory onto the primary disk directory (in case of disk crashes).

LIBRARY.CODE manipulates code segments in Pascal code and library files.

BOOTER.CODE copies bootstrap information from one disk to another.

GLOBALS.TEXT contains the Pascal declarations for the Modula operating system globals. M2.GLOBALD.TEXT contains the Modula-2 equivalent. The global declarations are used by experienced UCSD Pascal system programmers to access system information.

FORMAT.CODE is the disk formatter utility program. FORMAT.INFO contains primary bootstrap code which FORMAT writes to new disks.

DISKSIZE.CODE is the disk size utility program. It is used to change the volume size stored in a disk directory.

INITDATE.CODE is a program which sets the current date on all online disk volumes. (Its normal use is as a "startup" program.)

COPYBOOT.CODE is a program which copies disk files from the boot disk onto another disk volume and specifies the volume as the system volume. (Its normal use is as a "startup" program for running the system from a RAM disk.)

CALC.CODE is a simple desktop calculator simulation. Note that it requires floating point support.

2.4 Modula-2 System Files

This section describes the files that provide the Modula-2 system environment. These files are contained on the release disk set.

SYSTEM.COMPILER is the Modula-2 compiler.

SYSTEM.SYNTAX is the Modula-2 syntax error file.

MODULA.LIBRARY is the system library for the Modula-2 system.

LIB.CODE is the library manager utility program.

The files on the PROGS disk contain sample Modula-2 programs which are provided for your edification and enlightenment.

2.5 Interpreter Files

This section describes the files that are used to build new interpreters. The separate interpreter "skeleton" file, bootstrap, and peripheral device driver files are linked and configured to form new interpreters for different system configurations. See section 3 for details.

128K.INTERP is a linked interpreter configured to run in 128K bytes of memory.

86.SEP.CODE is the interpreter "skeleton" file that provides separate code and data spaces. 86.NONSEP.CODE is the interpreter skeleton file that allocates code and data within a single 64K space. 86.NOFP.CODE is equivalent to 86.NONSEP but does not contain floating point support software (yielding about 500 words of system memory). 86.BOOT.CODE is the secondary bootstrap file which is linked to an interpreter file.

Files beginning with the letters "IO" contain peripheral device driver code. IOSCREEN.CODE contains the screen driver. IODISKS.CODE the floppy disk driver, IORAMDSK.CODE the RAM disk driver, IOCOMS.CODE the serial port driver, IOLPTS.CODE the printer driver, and IOGAME.CODE the joystick driver. IOQUDCLK.CODE contains a calendar clock driver for the Quadram QuadBoard peripheral card.

LINKER.CODE is the interpreter linker program.

IBMUTIL.CODE is the interpreter configuration utility program.

3 System Configuration

This section explains how to configure the system for your IBM PC. System configuration is necessary because a PC can be equipped in so many different ways. The main factors influencing system configuration are the amount of memory available and the number of peripheral devices installed:

- The Modula-2 system can be used on PC's with anywhere from 64K to 640K bytes of memory. Memory is divided up into three areas: code space, data space, and RAM disk. On a 64K PC, available memory serves as a combined code and data space, with no memory left over for a RAM disk. When more memory is available, it can be used as a separate code space or RAM disk (or both).

- Software drivers are provided for the screen, disk drives, serial port, printer, RAM disk, joysticks, and calendar clock; these are linked with one of the interpreter "skeleton" files to create an interpreter configured for your PC. Including a driver with the interpreter provides you with access to the corresponding peripheral device. Not including a driver provides you with more system memory due to a smaller interpreter. Therefore, you will want to make an interpreter that contains only the drivers you need.

The system includes two preconfigured interpreters; with these, you can use the system immediately without having to perform any configuration procedures. One of these interpreters — named SYSTEM.INTERP — is used to boot the system the first time. The preconfigured interpreters are described in 3.1.

The utility program IBMUtil is used to set various configuration parameters: I/O unit number assignments, RAM disk and code space size, serial port baud rates, and so on. IBMUtil modifies parameter values stored in an existing interpreter file. IBMUtil is described in 3.2.

The utility program Linker is used to link together an interpreter "skeleton" and the appropriate drivers into a complete interpreter file. (IBMUtil must then be used to configure the newly linked interpreter.) The linker is described in 3.3.

The utility programs CopyBoot and InitDate offer RAM disk and calendar clock support. CopyBoot turns the RAM disk into the system disk. Benefits of running the system from the RAM disk include improved performance and freeing the system disk drive for your own use. InitDate uses the calendar clock to set the system date so you don't have to set it manually. Both of these programs are used as "startup" programs which are automatically executed when the system starts up. They are described in 3.4.

3.1 Preconfigured Interpreters

The file SYSTEM.INTERP supplied on the release disk is configured to operate in 64K bytes of memory and contains drivers for the screen, disk drives, serial port, printer, and RAM disk (allocated in all available memory beyond 64K). With this interpreter, the system should boot on any PC meeting the minimum hardware requirements.

The file 128K.INTERP is identical to SYSTEM.INTERP except that it is configured for separate 64K code and data spaces; it requires at least 128K bytes of memory to successfully boot the system. To use 128K.INTERP as the system interpreter, change the name of the existing SYSTEM.INTERP to 64K.INTERP, change 128K.INTERP to SYSTEM.INTERP, then reboot.

NOTE- The IBMUtil utility can be used to change the configuration parameters in these interpreters.

3.2 IBMUtil

The IBMUtil utility (IBMUTIL.CODE on the disk) is used to set configuration parameters in the interpreter. Note that these changes alter the interpreter file only and do not take effect until the system is rebooted.

After you X(ecute IBMUTIL, the following prompt appears:

Interp file?

Type in the name of the interpreter file you wish to modify. It is a good idea to reconfigure a copy of the current system interpreter file so you can fall back to the original if the reconfigured copy doesn't work properly.

The following prompt appears after you enter the file name:

INTERPUTIL: A(ttach drivers C(onfigure drivers Q(uit

A(ttach is used to assign unit numbers to the drivers. C(onfigure is used to change the configuration parameters of each driver. Q(uit exits the program.

NOTE- A(ttach is described later in this section.

After typing C(onfigure, the following prompt appears:

CONFIGURE: I(nterp D(isks R(amdisk S(erial P(rinter C(lock Q(uit

Note that the prompt may not display all of these commands; IBMUtil displays a command only if the corresponding driver is linked into the interpreter file being configured.

I(nterp

The I(nterp command is used to specify the code and data spaces. It displays the following prompt:

```
INTERP: D(ata range C(ode range Q(uit
```

A message appears below the prompt indicating whether the interpreter code and data spaces are separate or not:

```
Separate code and data
```

or...

```
Code and data in same space
```

IBMUtil also displays the current setting of the code and data spaces:

```
Data space segment base  = 0060H
Data space segment size  = 1000H
Code space segment base  = 1060H
Code space segment size  = 0FA0H
```

If the interpreter code space is nonseparate, IBMUtil prompts for and displays the data space only.

There are a few things worth noting about these settings. First, the values entered and displayed in this prompt are assumed to have an implicit trailing 0 digit. For instance, "1000H" denotes the hex value "10000", which is 65536 in decimal.

NOTE- The trailing zero digit is implicit because these values are assigned to the 8086 segment registers, which constitute twenty bit byte addresses with the low four bits implicitly set to zero.

The values shown above are suitable for a 128K PC with separate code and data. The data space starts at hex address 600 and extends for a full 64K to hex address 105FF. The code space starts at hex address 10600 and extends to the end of memory; it is slightly less than 64K because of the 600 hex offset of the data space.

On a 64K PC with nonseparate code and data, the code and data spaces are coincident, with a base address of 600 hex and size of 0FA00 hex.

After typing C(ode or D(ata, the following prompt appears:

CODE/DATA: B(ase S(ize Q(uit

Typing B(ase displays this message:

Code/Data space segment base in hex?

Typing S(ize displays this message:

Code/Data space segment size in hex?

In both cases, enter the appropriate hex value. Remember to omit the trailing 0 digit. (You do not have to type in a trailing 'H'.)

NOTE- The lowest address for a code or data space must be at least 600 hex. Memory between 0 and 400 hex are reserved for use as 8086 interrupt vector addresses. Memory between 400 and 600 hex is reserved for use by the PC's ROM software.

NOTE- The interpreter always resides within the data space.

NOTE- With separate code and data, the maximum data space size is 10000 hex. This limitation is imposed by the interpreter's use of 16-bit data pointers.

NOTE- Code and data spaces can cross 64K physical segment boundaries without causing any problems.

WARNING- Code, data, and RAM disk spaces must not overlap each other.

R(amdisk

The R(amdisk command is used to specify the RAM disk space; i.e., an area of memory that is used as a virtual disk. It displays the following prompt:

RAMDISK: B(ase address S(ize Q(uit

Below this appear the current settings of the code and data spaces:

```
Base segment address = 2000H
Size of disk         = 100 blocks
```

Type B(ase to set the base address. Note that it must be entered as a hex value with no trailing zero digit or 'H'.

Type S(ize to set the RAM disk size. The following prompt appears:

B(locks A(ll available Q(uit

Type A(ll to specify that all addressable memory above the the base address will be allocated for the RAM disk. (This setting appears in the setting display as "All available".) Type B(locks to assign a fixed amount of memory for the RAM disk. Note that this value is expected in terms of blocks (where a block is 512 bytes).

D(isks

The D(isks command is used to set various parameters controlling the floppy disk drives. It displays the following prompt:

DISKS: S(eek U(nload L(oad H(ead settle M(otor start O(ff motor B(uffer Q(uit

Below this appear the current drive settings:

```
Seek rate              = 6ms
Unload time            = 0ms
Load time              = 8ms
Head settle time       = 0ms
Motor start time       = 250ms
Motor off time         = 1850ms
Buffer segment address = 0020H
```

Each of the disk commands issues a prompt indicating acceptable values. (The individual prompts are straightforward and thus are not listed here.)

The B(uffer command is notable as it accepts yet another "segment address" with implicit trailing zero hex digit. The disk buffer is 512 bytes (200 hex) in length. It is used for partial sector reads: where the I/O system has requested to read a few bytes from the disk, while the low-level disk routines can only read a full sector (512 bytes) at a time.

NOTE- The standard disk buffer address of 200 hex takes advantage of some normally unused memory at the high end of the interrupt vector table (0-400 hex). The use of this memory as a buffer assumes that no interrupt vector above 80 hex is in use; otherwise, overlapping occurs.

NOTE- The disk parameter values shown above differ in some instances from IBM's recommended values. The standard seek rate is 8ms, but 6ms works on most machines and is faster and quieter. IBM suggests a head settle time of 25ms, but most everyone uses 0ms (which is twice as fast). Likewise, the standard motor start time is 500ms, but 250ms works well and is faster.

S(erial

The S(erial command is used to set various parameters controlling the serial port. It displays the following prompt:

SERIAL[COM1]: 1..4(port B(aud S(top bits P(arity W(ord size I(nterrupts Q(uit

Each of the serial port commands issues a prompt indicating acceptable values. (The individual prompts are straightforward and thus are not listed here.)

Typing the digits 1 through 4 selects the serial port to be configured (most PC's only have serial ports 1 and 2). The promptline displays the current serial port; for instance, "COM1" denotes serial port 1, "COM2" serial port 2, and so on.

The I(nterrupts command is used to enable serial port interrupts. A serial port can be configured to generate interrupts on input, output, error, or modem. (See the IBM Technical Reference for details.) The current interrupt status is indicated by the letters 'E', 'I', 'O', and 'M', which are displayed if the corresponding interrupt is enabled. If none are enabled, "None" is displayed.

Serial port interrupts are mapped to interrupt vector numbers in Modula-2's IOTRANSFER facility. See section 6 for details.

P(rinter

The P(rinter command is used to set various parameters controlling the printer port. It displays the following prompt:

PRINTER[LPT1]: 1..3(port I(nterrupts A(uto line feed Q(uit

Typing the digits 1 through 3 selects the printer port to be configured (most PC's only have printer port 1). The promptline displays the current printer port; for instance, "LPT1" denotes printer port 1.

The I(nterrupt command is used to control printer port interrupts: it toggles the current interrupt status, which is displayed below the promptline. Printer port interrupts are mapped to interrupt vector numbers in Modula-2's IOTRANSFER facility. See section 6 for details.

The A(uto line feed command controls a line connected to the standard IBM printer which tells the printer whether to auto line feed.

C(lock

The C(lock command is used to set the time and date in the calendar clock. It displays the following prompt:

CLOCK: D(ate T(ime Q(uit

The D(ate command is used to set the calendar date. It displays the following prompt:

DATE: M(onth D(ay Y(ear S(et A(bort

The month, day, and year are entered as integer values in the usual range. S(et establishes the entered values as the new date; A(bort exits without disturbing the current date.

The T(ime command is used to set the time. It displays the following prompt:

TIME: H(our M(inute S(et A(bort

The hour and minute are entered as integer values (the hour value assumes a 24 hour clock). S(et and A(bort work as described in the D(ate command above; however, using S(et here is like setting the time on a watch. The hour and minute values are set right when you type S(et; the seconds value is simultaneously set to zero.

NOTE- The time and date prompts appear only if the calendar clock card is installed and responding to the current calendar clock driver.

A(ttach

A(ttach is the companion command to C(onfigure. A(ttach is used to assign I/O unit numbers (and other attributes) to the drivers contained in an interpreter file. It displays the following prompt:

ATTACH:[0] U(nit D(river R(ead W(rite I(nit S(tat Q(uit

Two tables appear below the promptline; the table on the right displays the drivers linked into the interpreter, while the table on the left displays the I/O unit numbers (and whatever drivers have been assigned to them). To configure an interpreter, you must assign the drivers displayed in the right hand table to the unit table displayed on the left.

The driver table looks like this:

```
A) IOSCREEN    - C 1   - B 0
B) IODISKS     - C 0   - B 4
C) IOCOMS      - C 4   - B 0
D) IOLPTS      - C 3   - B 0
E) IORAMDSK    - C 0   - B 1
```

The letters on the left are used to select the drivers. Following the driver name are the driver attributes: 'C' denotes a character-oriented driver (e.g. console, printer, serial port), while 'B' denotes a block-oriented driver (e.g. disks).

The numbers following the 'B' and 'C' characters indicate the number of devices (of that type) that the driver supports. For instance, the IODISKS driver displayed above can control up to four byte-oriented devices, but no character-oriented devices.

The unit table looks like this:

```
#0 none
#1 IOSCREEN    - C 0   RWIS
#2 IOSCREEN    - C 0   RWIS
#3 none
#4 IODISKS     - B 0   RWI
#5 IODISKS     - B 1   RWI
#6 IOLPTS      - C 0   WIS
```

The numbers on the left are the available I/O unit numbers. Next is the name of the driver assigned to that unit. Following the name are the unit attributes: the type and "local driver number" of the assigned device. Local driver numbers are used to distinguish the devices supported by a single driver; for instance, if the driver IODISKS supports four disk devices, then the devices are identified by the local driver numbers zero through three. Each local driver number is assigned to a different unit number.

Finally, the characters 'R', 'W', 'I', and 'S' indicate the I/O operations available on the device via its unit number. 'R' indicates that read operations are allowed, 'W' write operations, 'I' initialization, and 'S' status.

NOTE- These I/O operations correspond to the UCSD Pascal intrinsics UNITREAD, UNITWRITE, UNITCLEAR, and UNITSTATUS.

Here is the attach promptline again:

ATTACH:[0] U(nit D(river R(ead W(rite I(nit S(tat Q(uit

All commands operate on the current unit, which is displayed on the promptline enclosed in square brackets. (Unit 0 is the current unit in the above promptline.) To change the current unit number, type U(nit followed by a unit number.

The D(river command is used to add, delete, or replace the driver assigned to the current unit. It displays the following prompt:

Driver letter ['A'..'F'], <SP> to remove, or <ESC>?

Type <sp> to delete the currently assigned driver or <esc> to exit the D(river command. The letters enclosed in brackets are used to assign drivers to a unit number; they correspond to the letters displayed in the driver table. When you select a driver by typing its letter, this prompt appears:

Local driver number?

Type in the local driver number you wish to assign.

When you first assign a driver to a unit number, all I/O operations allowable on the device are enabled. To disable any of these operations, use the R(ead, W(rite, I(nit, and S(tat commands; they toggle the current settings.

NOTE- The driver itself can control what operations can be enabled. If you attempt to enable an operation that is not provided by the driver, the response will be "Operation not allowed on this device".

There is one restriction and a number of conventions governing the assignment of drivers to unit numbers. The restriction is that character drivers **must** be assigned to units 1 and 2. The conventions are standard unit number-device assignments that originate from the UCSD Pascal system environment; they are recommended because some higher-level software might depend on them.

Standard unit number assignments and attributes:

Unit #	**Driver**	**Local Driver #**	**Status**
1-2	IOSCREEN	0	RWIS
4-5	IODISKS	0-1	RWI
6	IOLPTS	0	WIS
7	IOCOMS	0	R IS
8	IOCOMS	0	RWIS
9-10	IODISKS	2-3	RWI
11	IORAMDSK	0	RWI
17-19	IOCOMS	1-3	RWIS
20-21	IOLPTS	1-2	WIS
30	IOGAME	0	R
31	IOCALCLK	0	RWI

3.3 Linker

The Linker utility (LINKER.CODE on the disk) is used to link together an interpreter skeleton and drivers into a complete interpreter file.

After you X(ecute LINKER, the following prompt appears:

I(nterp L(ink A(ll code P(roc info Q(uit

Type 'I' to link together an interpreter. The following prompt appears:

Link file?

Type in the name of the appropriate interpreter skeleton file. (The ".CODE" suffix is automatically appended if omitted.)

Three interpreter skeleton files are provided with the system: 86.NONSEP.CODE, 86.NOFP.CODE, and 86.SEP.CODE. 86.NONSEP is used for interpreters where the code and data reside in the same 64K byte memory space. 86.NOFP is equivalent to 86.NONSEP, but does not contain support code for the 8087 math coprocessor; using this interpreter gains about 500 words of system memory. 86.SEP is used for interpreters that maintain code and data in separate memory spaces (64Kb max data, unlimited code).

After you type in the interpreter skeleton file, the original prompt appears again on the next line:

Link file?

Type in the name of a driver you wish to include. Each time you enter a driver name, the original prompt reappears, allowing you to enter the name of another driver.

Here are the names of the driver files:

IOSCREEN	- display driver
IODISKS	- floppy disk driver
IOCOMS	- serial port driver
IOLPTS	- printer driver
IOQUDCLK	- Quadboard calendar clock driver
IORAMDSK	- RAM disk driver
IOGAME	- joy stick driver

After you have entered all of the drivers you want, type in the name of the secondary bootstrap file: 86.BOOT.CODE. When the next "Link file?" prompt reappears, just type <return> — this indicates that all files have been entered and that linking can proceed.

NOTE- The interpreter skeleton must always be the first file entered for linking. The bootstrap must always be the last file entered. Drivers may be entered in any order.

After pausing for a while to read all of the files into memory, the linker then displays the following prompt:

Output file name?

Type in the name you want for the linked interpreter file. ("NEW.INTERP" is a favorite.) The linker then proceeds to link the files together; it displays the name, address range (in hex), and size (in decimal) of each file linked:

```
Proc INTERP86   Addr=0000-1EC1  Size=7874
Proc IODISKS    Addr=1EC2-207D  Size=444
Proc BOOT86     Addr=207E-22A3  Size=550
```

When linking is finished, the linker writes the linked interpreter out to disk and then terminates, returning control to the system prompt.

NOTE- The linked interpreter must be configured with the IBMutil program before it can be used as a system interpreter.

NOTE- The commands L(ink, A(ll code, and P(roc info are not documented here because they perform functions beyond the scope of this manual.

3.4 RAM Disk and Calendar Clock Support

The utility programs CopyBoot and InitDate offer RAM disk and calendar clock support.

CopyBoot

CopyBoot (COPYBOOT.CODE on the disk) converts the RAM disk into the system volume. It copies files from the system disk to the RAM disk and then specifies the RAM disk as the new system volume. Running the system from the RAM disk offers these benefits:

- System performance is improved, as oft-used programs such as the filer and editor are read from the RAM disk instead of the floppy disk.
- An extra disk drive is freed for your own use, as the system disk no longer needs to be online (because the system files reside on the RAM disk).

Files are copied in the order in which they appear on the system disk; CopyBoot displays the file names on the console as the files are copied. Copying continues until all of the files are copied or the RAM disk runs out of space.

CopyBoot has a feature which lets you limit the number of files copied onto the RAM disk. Starting with the first file in the system disk directory, CopyBoot copies files to the RAM disk until it finds a file named ENDBOOT. ENDBOOT and all subsequent files are not copied across. You can use the filer command Make to create a one-block data file named ENDBOOT.

Example of ENDBOOT in a system disk directory:

```
BOOT:
SYSTEM.MISCINFO        — the following files are copied
SYSTEM.PASCAL
SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.COMPILER
MODULA.LIBRARY
ENDBOOT                — the following files are not
LIB.CODE
PATCH.CODE
```

CopyBoot is intended for use as a "startup" program which is automatically executed when the system is booted. It should be renamed SYSTEM.STARTUP and stored on the system disk.

CopyBoot allows you to specify a "startup" program which is automatically executed after CopyBoot finishes. If the system disk contains the file SYSTEM.NEWSTART, CopyBoot copies it to the RAM disk and changes its name to SYSTEM.STARTUP. When CopyBoot finishes, the SYSTEM.STARTUP on the RAM disk is automatically executed.

NOTE- Both SYSTEM.STARTUP (CopyBoot in disguise) and SYSTEM.INTERP can be kept on the system disk ahead of ENDBOOT without their being copied to the RAM disk. This is because CopyBoot refuses to copy files named SYSTEM.STARTUP or SYSTEM.INTERP.

NOTE- CopyBoot assumes that the RAM disk has been assigned to unit 11. See 3.2 for details.

InitDate

InitDate (INITDATE.CODE on the disk) uses the calendar clock to set the system date so you don't have to set it manually.

InitDate reads the current date from the calendar clock card and sets the system date both in memory and on the system disk.

InitDate is intended for use as a "startup" program which is automatically executed when the system is booted. It should be renamed SYSTEM.STARTUP and stored on the system disk.

NOTE- InitDate assumes that the calendar clock has been assigned to unit 31. See 3.2 for details.

4 Formatting New Disks

The Format utility (FORMAT.CODE on the disk) is used to initialize floppy disks so they can be used with the system.

After you X(ecute FORMAT, the following prompt appears:

S(ingle sided D(ouble sided Q(uit

Type 's' if you are formatting a single-sided disk, or 'd' if a double-sided disk. (Typing 'q' exits the program.) A single-sided disk stores 320 blocks of data, while a double-sided disk stores double the amount: 640 blocks.

NOTE- Single-sided disks can be formatted as double-sided, but this is a risky practice. Disks formatted this way store twice as much data as normal, but tend to be unreliable.

Another prompt appears:

Format what unit? [4,5] (type return to exit)

Type in the unit number of the disk drive to be used for formatting the disk followed by <return>. (Typing just <return> exits the program.)

This prompt appears next:

Insert disk to be formatted and press F(ormat or Q(uit

Put the disk in the proper disk drive and type 'f'. If the disk has already been formatted, this prompt appears:

Destroy all information on BLANK?

Type 'y' to proceed with formatting; type 'n' to chicken out.

Format writes a period to the screen after formatting each track on the disk. If the disk is double-sided, colons appear instead of periods.

The following message appears when Format finishes formatting:

320 block disk formatted successfully

This is displayed when you are formatting single-sided disks; for double-sided disks, it will say "640 block disk".

Next, if the file FORMAT.INFO is present, Format automatically writes its contents onto blocks 0 and 1 of the newly formatted disk. FORMAT.INFO contains bootstrap code which is necessary if the disk is to be used as a system boot disk. (Blocks 0 and 1 of every disk are reserved for bootstrap code.)

The original prompt then reappears, allowing you to format another disk:

Insert disk to be formatted and press F(ormat or Q(uit

NOTE- Format automatically writes an empty disk directory onto disks it formats. Disks are named BLANK.

5 Machine-dependent Modules

This section describes machine-dependent modules provided with the Modula-2 system.

The library module IBMStuff (5.1) provides access to the display, calendar clock, and joysticks.

The module SYSTEM86 (5.2) provides access to low-level system facilities.

5.1 IBMStuff

The library module IBMstuff provides operations for controlling the joysticks, calendar clock, and monitor display.

The procedure GameIO returns the current settings of an analog input device attached to the PC. The most common analog input device is the joystick, which is most commonly used to control computer games. The parameters axis0 through axis3 are set to values in the range 0..99. The parameters button0 through button3 are set to TRUE if the corresponding button is depressed; otherwise, they are set to FALSE. The exact interpretation of these values depends on the specific peripheral device; for details, see the device documentation.

NOTE- If the period between successive calls to GameIO is too short, the subsequent call may not return correct values. Programs should generally allow 10 to 20 milliseconds to elapse between GameIO calls.

NOTE- GameIO assumes that an analog input device is properly connected to the PC and that the IOGAME driver has been assigned I/O unit 30 in the interpreter.

The procedures ReadTime and WriteTime are used to read and set the time and date from the calendar clock card. Time/date values are passed and returned as characters strings in variables of type TimeString. The character string always has the following format:

mm/dd/yy hh:mm:ss

The alphabetic letters denote digits. Leading zero digits and the space in the middle are significant. Hours are set according to 24-hour "military" time. Strings used to set the time must match this syntax exactly. Note that TimeString is declared so that the string always end with a null character.

NOTE- ReadTime and WriteTime assume that a calendar clock card is properly connected to the PC and that the IOCALCLK driver has been assigned I/O unit 31 in the interpreter.

The procedures ScreenPage, ScreenMode, and ScreenCharColor control the PC display.

The procedure ScreenMode sets the display to the specified mode: 40/80 columns, monochrome/color, or highres graphics as specified by the type VideoMode.

The PC dedicates 16K bytes of memory for its screen display. In highres graphics mode, the display uses all 16K, but in 40 or 80 column mode, the display uses only part of the display memory. In modes requiring only portions of the display memory, the remaining memory is available as screen "pages" which can be alternately displayed on the screen. In 25x40 mode, 8 pages are available; in 24x80 mode, 4 pages are available.

The procedure ScreenPage is used to switch between screen pages. To establish a new screen page, call ScreenPage with the appropriate page number; you can then write whatever you want onto the new page. When you switch back to the original page, the new page disappears, but it still contains whatever you wrote onto it. When you again switch to the new page, its contents are automatically displayed on the screen.

The procedure ScreenCharColor is used to set the color and intensity of the display. Setting the parameter blink to TRUE causes the display to blink. Setting the parameter intensity to TRUE causes the display to appear in high intensity. (Note that the high intensity option is not available on all monitors.) The parameters foreground and background control the display color; foreground indicates the the color of the characters. Foreground and background colors are specified by the type CharColor.

```
DEFINITION MODULE IBMStuff; (* $SEG := 63; *)

EXPORT QUALIFIED
  GameIO, VideoMode, ScreenMode, PageRange, ScreenPage,
  CharColor, ScreenCharColor, TimeString, ReadTime, WriteTime;

  PROCEDURE GameIO   (VAR axis0,axis1,axis2,axis3: CARDINAL;
                      VAR button0,button1,button2,button3: BOOLEAN);

  TYPE VideoMode =    (Video40X25BW,
                       Video40X25Color,
                       Video80X25BW,
                       Video80X25Color,
                       Video320X200Color,
                       Video320X200BW,
                       Video640X200BW);

  PROCEDURE ScreenMode (mode: VideoMode);

  TYPE PageRange = [0..7];

  PROCEDURE ScreenPage (pagenum: PageRange);

  TYPE CharColor = SET OF (ColorR,ColorG,ColorB);

  PROCEDURE ScreenCharColor(blink,intensity: BOOLEAN;
                            foreground,background: CharColor);

  TYPE TimeString = ARRAY [0..17] OF CHAR;
                           (* digits: "mm/dd/yy hh:mm:ss" *)

  PROCEDURE ReadTime (VAR time: TimeString);

  PROCEDURE WriteTime (time: TimeString);

END IBMStuff.
```

5.2 SYSTEM86

The library module SYSTEM86 provides access to various low-level system facilities.

The procedures Peek and Poke provide access to arbitrary locations in 8086 memory. Because of the 8086's segmented memory architecture, a memory address is specified by two parameters: seg and offset. The address mapping is as follows:

logical address = seg*16 + offset

The procedures InByte, OutByte, InWord, and OutWord correspond to the 8086 I/O port instructions. On the byte I/O operations, the high order byte of the word quantity is either ignored (OutByte) or zeroed (InByte).

XUnitRead and XUnitWrite are equivalent to the UCSD intrinsics UnitRead and UnitWrite except that they use the 8086's segmented addresses to specify the buffer area.

ClearVector, Raise, and SetPriority provide access to the interrupt system. Section 6 describes the interrupt system.

ClearVector is always passed NILs as address parameters. It is used to disassociate a process from an interrupt vector which it has performed IOTRANSFERs to.

Raise generates an interrupt through the specified interrupt vector.

NOTE- SYSTEM86 is a compile-time module and does not occupy any memory at run time. All operations expand to "inline" code sequences rather than actual Modula-2 procedure calls.

```
DEFINITION MODULE SYSTEM86;  (* $SEG := 1; *)

FROM SYSTEM IMPORT WORD, ADDRESS;

EXPORT QUALIFIED
  Peek, Poke, InByte, InWord, OutByte, OutWord,
  Raise, ClearVector, SetPriority,
  XUnitRead, XUnitWrite;

  PROCEDURE Peek (seg, offset: CARDINAL): CARDINAL;

  PROCEDURE Poke (w: WORD; seg,offset: CARDINAL);

  PROCEDURE InByte (portnum: CARDINAL): CARDINAL;

  PROCEDURE InWord (portnum: CARDINAL): CARDINAL;

  PROCEDURE OutByte (value: WORD; portnum: CARDINAL);

  PROCEDURE OutWord (value: WORD; portnum: CARDINAL);

  PROCEDURE Raise (vector: CARDINAL);

  PROCEDURE ClearVector (a,b: ADDRESS; vector: CARDINAL);

  PROCEDURE SetPriority (NewPriority: CARDINAL): CARDINAL;

  PROCEDURE XUnitRead (unit,seg,offset,bytes,block: CARDINAL;
                       control: BITSET);

  PROCEDURE XUnitWrite (unit,seg,offset,bytes,block: CARDINAL;
                        control: BITSET);

END SYSTEM86.
```

6 Interrupt System

The interrupt system on the PC provides interrupt vectors for the keyboard, serial and parallel ports, timer, vertical retrace, and program break.

Modula-2 programs are interruptable only between the execution of P-codes. Thus, if an interrupt occurs in the middle of a P-code (this includes low-level I/O operations), the corresponding IOTRANSFER cannot occur until the interpreter fetches the next P-code.

A process is connected to an interrupt only while an IOTRANSFER call is pending. If an interrupt occurs through a vector and there is no IOTRANSFER pending, the interrupt is queued for the next IOTRANSFER call; any subsequent interrupts through the vector are ignored. If IOTRANSFER is called on a vector where an interrupt is pending, the transfer takes place immediately.

One ramification of this scheme is that when a program calls IOTRANSFER for the first time, and an interrupt is pending on the vector, an I/O transfer occurs immediately. To clear any pending interrupts, a program must call ClearVector. before its first IOTRANSFER call (see below for details).

Nine interrupt vectors are defined for the PC; they are numbered 0 through 8. Vector assignments are as follows:

vector	**device**
0	timer (18.2/sec)
1	keyboard (press & release)
2	vertical retrace
3	unused
4	serial port
5	unused
6	unused
7	parallel port
8	program break

Vector 0 is preprogrammed to interrupt 18.2 times per second. It is useful for programming time-slicing into a process scheduler by using the vector as a timer interrupt to switch processes.

Vector 1 interrupts whenever a key changes position. Note that this includes both pressing and releasing a key; thus, every keystroke generates two interrupts. This feature can be useful for detecting when a key is being held down. The procedure UnitIO.UnitBusy can be used to determine whether an interrupt was caused by pressing or releasing a key: if UnitBusy(1) returns FALSE, the interrupt was caused by pressing a key. (Unit 1 is the console I/O unit.)

NOTE- If a key is pressed and held down, the keyboard's auto-repeat feature generates an interrupt for each "virtual" key press, but not for the corresponding virtual release.

Vector 2 is known as the vertical retrace interrupt. "Vertical retrace" refers to the period of time when the display monitor's electron scanning gun returns to the top left hand of the screen. This vector is less commonly used than the others, being limited to programming high speed flicker-free graphics.

Vector 4 is for serial port interrupts. Note that an interpreter can be configured to generate serial port interrupts on input, output, error, or dial-up; see 3.2 for details.

Vector 7 is for parallel port interrupts. Note that an interpreter can be configured with parallel port interrupts either enabled or disabled; see 3.2 for details.

Vector 8 interrupts when the program break key is typed. If a process is waiting (via IOTRANSFER) on vector 8 when the break key is typed, the IOTRANSFER occurs; otherwise, the usual execution error occurs.

Interrupt priorities are treated as mask values rather than ordinal values. The lower eight bits of a module priority number specify which devices are prevented from interrupting a module. Bits 0 through 7 in a module priority number correspond to interrupt vectors 0 through 7; if a bit is set to 0, the corresponding vector cannot interrupt the module. For instance: priority value 0 would prevent all vectors from interrupting a module; 2 would prevent the keyboard (vector 1) from interrupting; 0FFH would allow all vectors to interrupt a module. The default interrupt priority is 0.

NOTE- Interrupt vector 8 is nonmaskable; module priority numbers have no effect upon it.

The PC interrupt system provides the procedures Raise, SetPriority, and ClearVector to support the use of IOTRANSFER. These are exported from the module SYSTEM86 — see 5.2 for details.

Raise causes an interrupt through the specified vector.

SetPriority sets the interrupt priority to the specified value and returns the current priority as a function result. SetPriority should be used only when necessary; module priority numbers should be used whenever possible.

ClearVector terminates any IOTRANSFERs pending on a vector. Before terminating, programs using IOTRANSFER must call ClearVector(NIL,NIL,x) for every interrupt vector that they use; otherwise, an interrupt occuring after the program has terminated is likely to crash the system.

NOTE- Only one IOTRANSFER can be pending on an interrupt vector at any one time; otherwise, execution error 17 occurs.

WARNING- The program break key can terminate a program without allowing it to execute its ClearVector calls, resulting in subsequent system crashes. Unless you can guarantee that they will not be interrupted by program breaks, all programs using IOTRANSFER should create special handler processes which wait for program breaks and call ClearVector before terminating the program.

7 Machine-level Data Representation

This chapter describes machine-level data representation on the PC. The basic unit of storage is a 16-bit word. Bits in a word are numbered 0 to 15; bit 0 is the least significant bit. All machine addresses are byte addresses.

- The procedures SIZE and TSIZE return results in units of bytes.
- Words consist of two 8-bit bytes. The lower-addressed byte always contains the **least** significant byte of a word quantity.
- Integers are stored in one word as 16-bit two's complement values. The minimum integer value is -32768. The maximum integer value is 32767.
- Cardinals are stored in one word as unsigned 16-bit integers. The minimum cardinal value is 0. The maximum cardinal value is 65535.
- Booleans are stored in one word. The cardinal value of FALSE is 0. The cardinal value of TRUE is 1.
- Characters are stored in one word (except in character arrays and fixed address variables). The least significant byte contains an ASCII character value. The most significant byte contains 0.
- Reals are stored in four words in IEEE 64-bit format.
- Enumerations are stored in one word. Enumerated constants are assigned cardinal values 0, 1, 2, ... in the order they are declared.
- Subranges assume the data representation of their base type.
- Arrays are stored in integral numbers of words. Character arrays in Modula-2 are stored as byte arrays; the first character is stored in the lowest-addressed byte, and remaining characters are stored in consecutive bytes.
- Records are stored in integral numbers of words. Record fields are allocated in the order they are declared.
- Sets are stored in integral numbers of words (unless they are sub-word fields of packed records or arrays). The number of words allocated for a set is 1 + ORD(high element) DIV 16. Sets contain up to 4080 elements (255 words). Sets of negative integers are not allowed.

- Pointers are stored in one word and contain absolute byte addresses. The integer value of NIL is -1.
- Procedure types are stored in one word. The lower-addressed byte contains the segment number; the higher-addressed byte the procedure number.
- Opaque types are stored in one word.
- The types Decimals.DECIMAL, Wides.WIDE, and Files.FilePos are actually multiword records — they are documented as opaque types merely to discourage access to their internal representations.
- Decimal numbers (of type DECIMAL) are stored in 5 words. The five words are treated as an array of 20 4-bit "nibbles" — the first nibble contains the decimal sign (of type DecState); the remaining 19 nibbles contain decimal digits (0..9).
- File positions (of type FilePos) are stored in 2 words. The first word contains a block number. The second word contains a byte offset in the range 1..512.
- Character variables declared at fixed addresses are stored in a single byte.
- Packed data representation is described in 3.1.1 of **The Modula-2 System.**

The compiler assigns data offsets to variables and parameters in the order they are declared. For example, given the declaration

```
VAR I,J: INTEGER;
      K: BOOLEAN;
```

The following is always true: ADR(I) < ADR (J) < ADR(K).

8 Library Module Hierarchy

This section describes intermodule dependencies of all library modules provided with the system. For more information on module dependencies and the module hierarchy, see section 3 in **Standard Library.**

- **RealInOut** -> InOut, Reals, Standards
- **InOut** -> Texts, Files, Conversions, Standards
- **Reals** -> Texts, Standards
- **Texts** -> Conversions, Files, Storage, Program, Standards, Bits, FileDef, UCSDGlobals
- **Files** -> Storage, Program, SystemTypes, UnitIO, Standards, Bits, FileDef, BlockIO, UCSDGlobals
- **Terminal** -> UCSDGlobals
- **Strings** -> Program, Standards
- **Processes** -> Storage
- **Storage** -> Program, Standards
- **Program** -> Storage, SubProgram, LibDef, Standards, BlockIO, UnitIO, Bits

Index

Modula

Operating System

Release:	**0.3**
Date:	**26 August 1983**
Author:	**Richard Gleaves**

Table Of Contents

1 Introduction

The Modula operating system is an interactive single-user system for developing Modula-2 and Pascal programs. It is compatible with the version II UCSD Pascal system.

The Modula operating system provides the following facilities:

- **Batch Command Interpreter** — Reads a series of system commands and data from a command file. It is used to automate repetitive system tasks.
- **Shell Command Interpreter** — Creates a "command shell" programming environment where individual programs can be linked together to perform complex tasks.
- **File Manager** — Manages disk files and volumes.
- **Line-oriented Editor** — Offers basic text editing capabilities on unconfigured systems.
- **Pascal Compiler** — A fast one-pass compiler which includes many of the UCSD Pascal language extensions.
- **Utility Programs** — A large collection of utilities which aid system configuration and software development.

1.1 Scope of This Manual

This manual describes Volition Systems' Modula operating system. It provides a complete description of the system commands and features.

This manual is not a tutorial — it assumes you are familiar with the UCSD Pascal language and the UCSD Pascal system. If you have not used UCSD Pascal, the following books are recommended as tutorials:

Introduction to the UCSD p-system
Charles W. Grant and Jon Butah
Sybex, Berkeley, California, 1982.

Introduction to Pascal (Including UCSD Pascal)
Rodnay Zaks
Sybex, Berkeley, California, 1981.

1.2 Notation

This section describes the notation used in this manual.

A variant of Backus-Naur form (BNF) is used as a notation for describing the form of promptlines, input data, and text file items. 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 can be substituted with the right-hand side. The vertical bar '|' separates possible choices for substitution; the example above indicates that 'trout', 'salmon', or 'tuna' can be substituted for '<fish>'.

An item enclosed in brackets ('[' and ']') can be optionally substituted into a textual expression; for instance, '[micro]computer' can represent the text strings 'computer' and 'microcomputer'.

An item enclosed in braces ('{' and '}') can 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}

Literal occurrences of the characters used above are delimited by quotes to avoid confusing them with notational definitions (e.g. <left-bracket> ::= '<' | '{' | '[').

BNF notation is often used informally to describe the appearance of a promptline or the required form of some input data. Here are some typical examples:

Typing <cr> completes the input prompt.
President <surname> should be [<expletive>] impeached!

The syntax for Pascal's IF statement is:

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

NOTE- Paragraphs beginning with the word **NOTE** contain interesting or useful information related to the current topic.

WARNING- Paragraphs beginning with the word **WARNING** point out potential problems associated with the current topic.

Section references have the form 'x.y.z. ...', where x, y, and z denote digits. The first digit indicates the chapter; subsequent digits indicate sections within the chapter. For instance, the phrase 'see 3.4' refers to section 4 in chapter 3.

This manual defines a number of terms for describing system features and commands. When new terms are introduced, they appear in **boldface** and are followed by either a definition or a reference to the defining section.

1.3 Terminology

This section describes the terminology used in this manual.

The following terms are used to describe the file I/O system: **file name, file block, block number, unit,** and **unit number.**

A **file name** is a character string that conforms to the file naming conventions of the UCSD Pascal file system.

File names usually consist of a **file title** and a **file suffix.** For instance, the file name 'LIB.TEXT' contains the file title 'LIB' and the file suffix '.TEXT'.

A **file block** is the basic unit of disk file storage; a block contains 512 bytes. A **block number** is a number specifying a file block within a disk file; the first block of a disk file is block 0.

A **unit** corresponds to a physical I/O device. Each unit is identified by a unique **unit number.** For instance, unit 1 is the system console, unit 6 is the printer, and units 4 and 5 are the disk drives.

This manual upholds the tradition in UCSD Pascal system documents of describing key commands as metawords. **<cr>** and **<return>** denote the carriage return key. **<spacebar>** and **<space>** denote the space key. **<etx>** and **<esc>** denote the keys defined as the accept and escape commands respectively. Vector keys are denoted by **<left>**, **<right>**, **<up>**, and **<down>**. Many of these commands are defined using the Setup utility (10.5.1).

2 Basic Concepts

This chapter describes some basic concepts used throughout the system:

- Promptlines
- Prompts
- Key commands

Promptlines display the available commands across the top of the screen. Promptlines are described in section 2.1.

Prompts ask you for specific information: a file name, a number, or sometimes just a 'yes' or 'no'. Prompts are described in section 2.2.

Key commands are certain terminal keys which are defined as system commands. Key commands are described in section 2.3.

2.1 Promptlines

Promptlines are used to display the available commands in many parts of the system. Promptlines appear across the top of the screen. Here is an example:

```
Command: E(dit, R(un, F(ile, C(omp, B(atch, S(hell, X(ecute
```

Commands are invoked by typing one of the **command characters.** The command characters in a promptline are capitalized and (usually) separated from the command abbreviation with a left parenthesis; for instance, typing 'e' to the above promptline invokes the editor. Note that command characters can be typed in either as upper or lower case letters.

Some promptlines contain more commands than can be displayed across the screen. With these promptlines, typing '?' as a command character displays a promptline containing a second set of commands.

Many system promptlines display a version number in their promptlines. This indicates what release version of that program you are using.

2.2 Prompts

The system displays a **prompt** when it needs some information. Prompts usually appear in the form of a question:

Compile what file?

Are you sure (y/n)?

Scan for how many blocks?

Prompts fall into two classes: the 'yes/no' prompt or the input prompt.

'Yes/no' prompts are the simplest to answer: merely type 'y' or 'n'. ('Y' and 'N' also work.)

Input prompts require a string of characters followed by <return>. Typing <return> signals the end of the character string. You can use <backspace> to erase any typing mistakes. You can also erase everything you've typed by hitting <delete>.

Most input prompts offer a way to 'escape' from the prompt. For instance, typing just <return> to the prompt

Compile what file?

... returns you to the system promptline. Most (but not all) input prompts in the system recognize <return> as an escape.

2.3 Key Commands

This section describes key commands recognized by the system.

The <accept> and <escape> keys are used throughout the system as command terminators. <accept> is used mostly in ASE, where it indicates a command is to be completed. <escape> is recognized as a way to escape from (undo) a command. The actual keys that invoke <accept> and <escape> are 'soft'; that is, they are defined when the terminal is configured for the system.

Cursor movement is important in a few parts of the system — mainly the editor. <space> and <backspace> move the cursor in the expected direction. The vector keys <left>, <right>, <up>, and <down> also work in the usual sense.

The <eof> key is used to terminate character streams when the console is accessed as an input file. <eof> is usually defined (during system configuration) as ctrl-C.

The stop/start keys are used to stop console output. This is useful for stopping programs which are spewing out data faster than you can read it — typing the stop key halts the program, freezing the console display. Console output (and thus program execution) are resumed by typing the start key. ctrl-S usually serves as the stop/start toggle, but some systems define ctrl-S as the stop key and ctrl-Q as the start key.

The flush key causes the system to discard all console output until the next console read operation is completed. Console output can also be restored by typing the flush key again. The flush key is useful for speeding up programs which are bogged down because of their voluminous console output. The flush key is usually defined as ctrl-F.

3 Operating System

The Modula operating system provides basic facilities for program execution, error handling, and Pascal runtime support. It also offers a number of useful features:

- You can configure the system to automatically execute a user program when it first starts up. This lets you create turnkey applications.
- Programs can be kept in the **work file** to reduce the number of commands needed to compile and execute them. This speeds up program development.
- When a syntax error occurs during compilation, the system can automatically invoke the editor to let you fix the error. This simplifies debugging.
- The file system implements high level concepts such as removable disk volumes and device-independent file I/O.
- The batch command interpreter lets you automate repetitive system tasks. System commands and data are read from a command file instead of the keyboard. The system can be configured to automatically execute a command file when it first starts up.

Section 3.1 explains how to start the system.

The work file is described in 3.2.

Syntax errors and editor invocation are described in 3.3.

Runtime error handling is described in 3.4.

Disk volume swapping is described in 3.5.

Operating system commands are described in 3.6.

NOTE- The file system is described in chapter 4.

3.1 Starting the System

The operating system is started up by placing the system disk in the proper disk drive and pressing the 'boot' button. (Details on this procedure may vary across different machines.)

After a few moments and a few disk accesses, the welcome message appears on the screen:

```
                         SYSTEM:



                        29 Mar 83

             Volition Systems Modula-2  0.3m
```

The welcome message displays the names of all online disk volumes and the current system date. The system promptline then appears across the top of the screen:

```
   Xecute, Batch, Shell, Run, File, Edit, Comp, UsrRst, Init
```

The system is now ready for you to type one of the system commands. See 3.6 for details on the system commands.

If a program named SYSTEM.STARTUP resides on the system disk, the operating system immediately executes it instead of displaying the welcome message and system prompt. This feature lets you create stand-alone application systems where the operating system is never visible to the end user.

3.2 The Work File

The **work file** is a special file which is used as a 'work' area for developing programs. The work file speeds up program development by reducing the number of commands you have to type in order to edit, compile, and execute your program.

Here is how the work file works:

- If you type E(dit and a work file exists, the editor selects it as the input file.
- If you type C(ompile and a work file exists, the compiler automatically begins compiling the work file.
- If you type R(un and a compiled work file exists, the operating system automatically executes the work file.
- If you type R(un and an uncompiled work file exists, the system automatically compiles and executes the work file.

The work file actually consists of two parts: the **work text file**, and the **work code file.** The work code file is usually the compiled form of the work text file.

To create a new work file, type <return> to the editor's input file prompt. At the end of the edit session, the editor's U(pdate command writes the newly created work file to a disk file named SYSTEM.WRK.TEXT. Compiling this file results in the file SYSTEM.WRK.CODE.

Because the work file SYSTEM.WRK is only a temporary 'scratch pad', you will probably want to save the results as a regular disk file. The filer command S(ave lets you permanently save the current work file as a regular file.

Another way to create a work file is to select an existing disk file as the new work file. The filer command G(et specifies an existing named disk file as the work file.

Finally, to get rid of the current work file, the filer command N(ew removes any SYSTEM.WRK files and disassociates any existing regular files from being the work file.

3.3 Syntax Errors and the Editor

When the compiler finds a syntax error in your program, you have the choice of continuing compiling, stopping the compiler, or invoking the editor to fix the error. If you choose to edit your program, the operating system automatically starts the editor, which moves the cursor to the location of the error and displays a message describing the error.

NOTE- If you are not compiling the work file, the editor has to ask for the name of the file being compiled before it can jump to the syntax error location.

3.4 Runtime Errors

The operating system is responsible for handling software and hardware errors that occur during Pascal program execution. When an error occurs, the system displays a message describing the error and then terminates the program.

Note that programs may be terminated either by an execution error or by explicit user interruption of the program (i.e. typing the <break> key).

Execution error messages consist of an error description and the location of the error in the program code.

Example of an execution error message:

```
Divide by zero
S# 1, P# 5, I# 20
Type <space> to continue
```

The error description is usually a textual message (e.g. 'Value Range Error'), but sometimes only an execution error number is displayed. Execution error numbers are described in Appendix 3.

When an execution error is caused by an I/O error, a message describing the I/O error is printed next to the execution error message. As with execution errors, sometimes only an I/O error number is displayed. I/O error numbers are described in Appendix 2.

The execution error location is specified in terms of the code file structure; the 'S', 'P', and 'I' fields indicate the segment, procedure, and code offset of the instruction that caused the error. If you have a compiled listing of the program, you can use these offsets to trace the error to a single statement within the source program.

After you type the space bar, the system terminates the program, reinitializes itself, and redisplays the system prompt.

A **stack overflow** occurs when there is not enough memory for a program to continue executing. The program is terminated, and the following message appears on the screen:

```
*STK OFLOW*
```

The system then reinitializes itself and redisplays the system prompt.

3.5 Disk Swapping

The Modula operating system allows you to mount and dismount different disk volumes during normal system operation. For instance, to transfer files from one disk to another, you can enter the filer, replace the system disk with the disks you wish to transfer to, then transfer the files. By detecting disk swapping, the system is able to keep track of what disk volumes are currently online (without crashing like some other popular operating systems do).

During program execution, however, disk swapping can be risky. if a system or user program requires a code segment from a disk volume and the volume is no longer online, the system will crash. The system addresses this problem in two ways.

First, the file handler and most utility programs do not contain segment procedures; their code remains resident throughout execution. (Your programs will have to do the same in order to be immune to disk swapping.)

Second, the operating system tries to protect itself from crashes caused by random disk swapping. If the operating system determines that the system disk is not mounted, the following message appears after the program terminates:

Replace SYSTEM:

The system then waits until the system disk is remounted in the proper drive; when it is, the system prompt is redisplayed.

WARNING- The operating system detects disk swapping by checking the disk volume name whenever a disk is accessed by a directory operation (e.g. Open or Close). If the system disk is replaced by a disk volume that is never accessed, program termination will halt the system with an unrecoverable execution error.

3.6 System Commands

This section describes the commands available from the system promptline.

3.6.1 Clear Screen

If you type a character which is not a system command, the screen is cleared and the system promptline is redisplayed.

3.6.2 C(ompile

C(ompile invokes the Pascal compiler.

The compiler is the code file SYSTEM.COMPILER.

The Pascal compiler is described in chapter 8.

3.6.3 E(dit

E(dit invokes the editor.

The editor is a code file named SYSTEM.EDITOR.

If a work text file exists, it is automatically entered as the input file name.

The editor is described in the **ASE User's Manual.**

3.6.4 F(ile

F(ile invokes the filer.

The filer is the code file SYSTEM.FILER.

The filer is described in chapter 5.

NOTE- The filer can be operated without having the disk containing the filer code file online.

3.6.5 H(alt

H(alt stops the system.

The only way to restart the system after H(alt is to reboot.

3.6.6 I(nitialize

I(nitialize causes the system to reinitialize its state information.

The operating system clears all online I/O devices, rebuilds the system data structures, and searches all online disk volumes to locate the system programs.

Most execution errors automatically invoke I(nitialize.

3.6.7 R(un

R(un executes the work file.

If the work code file does not exist, the compiler is automatically invoked. If a work text file does not exist, the compiler input prompt appears.

3.6.8 B(atch

B(atch invokes the batch command interpreter.

The batch command interpreter is the code file SYSTEM.BATCH on the system volume.

The batch command interpreter is described in chapter 6.

3.6.9 S(hell

S(hell invokes the shell command interpreter.

The shell command interpreter is the code file SYSTEM.SHELL on the system volume.

The shell command interpreter is described in chapter 7.

3.6.10 U(ser restart

U(ser restart reexecutes the last program executed. U(ser restart does not work if the system has been reinitialized.

3.6.11 X(ecute

X(ecute executes the specified code file.

The following prompt appears:

> **Execute what file?**

Type in the title of the code file to be executed. The file suffix '.CODE' is automatically appended to the file name (unless it ends with a period).

4 File System

This chapter describes the Modula file system. Note that it is identical to the UCSD Pascal file system.

There are two good reasons for learning the file system in detail:

- Much of the file name syntax involves simplifying the specification of a file. Because most of your time in the system is spent creating, modifying, or deleting files, you can save a lot of time and effort by learning the file naming conventions.

- The Modula file system requires more a bit more user attention than other file systems; in particular, it is susceptible to running out of disk space if disk files and disk volumes are not properly managed. This problem can be minimized if your programs are designed with a firm understanding of how the file system works.

Files and disk file attributes are described in 4.1.

Disk volumes are described in 4.2.

Disk directories are described in 4.3.

File names and file name syntax are described in 4.4.

4.1 Files

A **file** is a collection of information which is usually stored on a disk. A disk file is referred to by its file name. Each disk has a directory which contains the file names and disk locations of each file on the disk. Files are accessed by programs and by the filer.

4.1.1 File Attributes

Each file is assigned a number of **file attributes.**

One of the attributes of a file is the **file date.** The current system date (5.3.3) is assigned to a file when it is created or modified.

Another file attribute is called the **file type.** The type of a file determines the way it can be used; for instance, the system does not let you edit code files or execute text files. File types are assigned based on part of the file name known as the **file suffix.** Note that file types are assigned when the file is first created; the file name can be changed later without affecting the file type.

Here are the reserved file suffixes:

.TEXT .BACK	Human-readable text.
.CODE	Machine-executable code.
.DATA	Data. (The default file type)
.BAD	A worn-out area on the disk.

4.1.2 File Lengths

When a disk file is created, the file system allocates a fixed area on disk where the file is to reside. When the newly created file is closed, the file system releases any disk space that was not actually used by the file; however, while the file is open, it reserves all of its allocated disk space for growing room. A file's allocated disk space is known as its **length attribute.**

By default, newly created files are allocated the largest free space on the volume; this minimize the chances of their running out of disk space as they are written to. However, this scheme causes problems when a program attempts to create more than one file on a disk volume having only one free space; though the free space might easily contain all of the finished files, the first file created is allocated all available disk space, preventing the other files from being created.

To avoid this problem, a file's length attribute can be explicitly specified by appending a **length specifier** to the file name. The length specifier value indicates the estimated maximum file size (in blocks). The file system then allocates the disk file in the first free space large enough to contain it. For instance, with the file name 'roger.file[77]', the file system allocates the file 'roger.file' 77 blocks of disk space in the first free space large enough to contain it.

The file length specifier '[*]' is useful when creating more than one file on a volume; it allocates the file in either half the largest free space on the disk or the second largest free space — whichever is largest.

NOTE- If a growing file reaches the end of its allocated disk space, one of two things happens. If the disk space following the file's allocated space is already occupied by an existing file, the file system reports the error 'No room on volume'. If the following disk space is part of a free space, the file system extends the file's length attribute into the adjacent free space.

NOTE- Over time, disk free spaces tend to increase in number and decrease in size, making it more difficult to create new files (or extend existing ones). Free spaces can be consolidated with the file manager's K(runch command (5.3.6).

4.2 Volumes

A **volume** is any I/O device: the printer, the console, a serial port, or a disk.

A **serial device** cannot store information — it can only produce or consume a stream of data. The console, printer, and serial ports are serial devices.

A **block-structured device** is one that can store a directory and files — most commonly a disk drive. Block-structured devices are divided into a fixed number of 512 byte storage areas known as **blocks.** Blocks are randomly accessible by block number. Disks are usually the only block-structured devices.

Volumes are addressed either by their **volume name** or **unit number.** Each I/O device is assigned a unit number. Each serial device is assigned a fixed volume name. The volume name of a block-structured device is whatever disk volume is mounted in the device.

Volume Name	**Unit Number**	**Description**
CONSOLE:	1	screen & keyboard with echo
SYSTERM:	2	screen & keyboard without echo
<vol name>	4	the system 'boot' disk
<vol name>	5	the alternate disk
PRINTER:	6	the line printer
REMIN:	7	serial input line
REMOUT:	8	serial output line
<vol names>	9 - 12	additional disk drives

4.3 Directories

Directories are stored on a disk along with the disk files. The directory is 4 blocks longs and is kept on blocks 2 thru 5 of a disk.

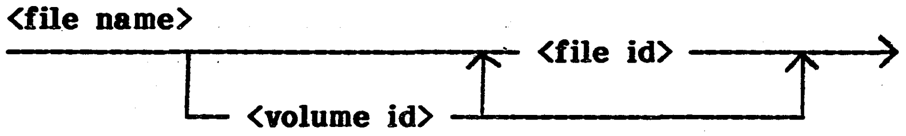
Directories contain the volume name and up to 77 directory entries. Each directory entry contains the name, disk location, and attributes of a disk file. A directory cannot contain two permanent files with the same name; saving a file on a disk volume already containing a file with the same name deletes the existing file. Note that program-generated temporary disk files can coexist with files having the same name — a temporary file only becomes permanent when it is closed and locked (8.3.1).

The filer automatically initializes disk volumes to contain two disk directories. The second directory is called a **duplicate directory** — it serves as a backup copy of the main directory.

The duplicate directory is 4 blocks longs and is kept on blocks 6 thru 9 of the disk. If something happens to the main directory (e.g. a bad block), it can be restored by using the information stored in the backup directory. The utility program CopyDupDir copies the duplicate directory onto the space occupied by the main directory.

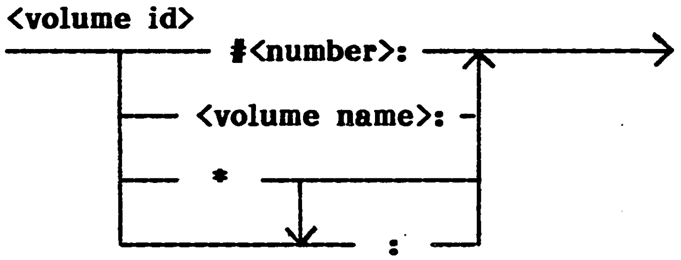
4.4 File Names

File names are used to address files and volumes. A file's attributes, maximum size, and disk location are all controlled by its initial file name. The system commands and the program I/O intrinsics both use the same file name syntax.



4.4.1 Volume Identifiers

Volume identifiers are used to specify volumes or files contained in volumes.



A volume name can include any characters except '#' or ':' and can be up seven characters long.

Using '*' as a volume identifier specifies the system volume. This is provided as a handy abbreviation for addressing files stored on the system disk. Note that unlike other volume identifiers, '*' does not require a colon to appear after it. The entire system volume can be addressed with the file name '*'.

The default volume name is called the **prefixed volume.** Disk file names lacking a volume identifier are assumed to reside on the prefixed volume; thus, you can save time by setting the most frequently used disk volume as the prefixed volume. The prefixed volume is set with the filer command P(refix.

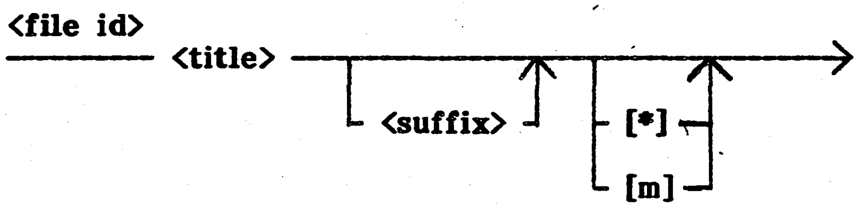
Examples of volume names:

```
#5:
XFRDISK:
*
:
*:
```

WARNING- Beware of using disk volume names as normal files. Treating a disk volume as a file exposes its directory to harm, because if you write to this 'file', the data gets written over the directory and all files on the disk get lost. The only place volume names should be used as file names is in the filer when it prompts for a volume name or when you C(hange a volume name.

4.4.2 File Identifiers

File identifiers are used to specify disk files stored on volumes.



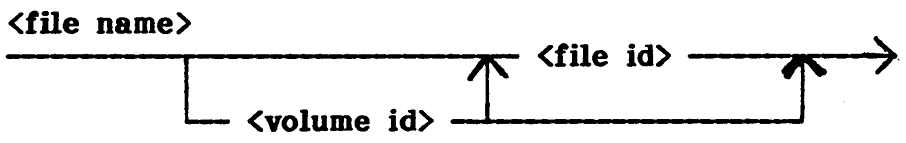
A **file identifier** consists of a title followed by an optional suffix and **length specifier.** The title and suffix may be up to fifteen characters long.

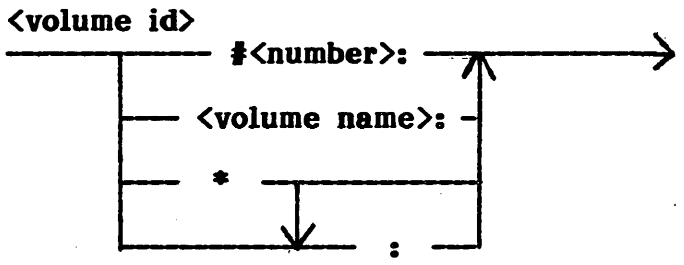
A file length specifier is delimited by square brackets. The symbol 'm' denotes a positive integer.

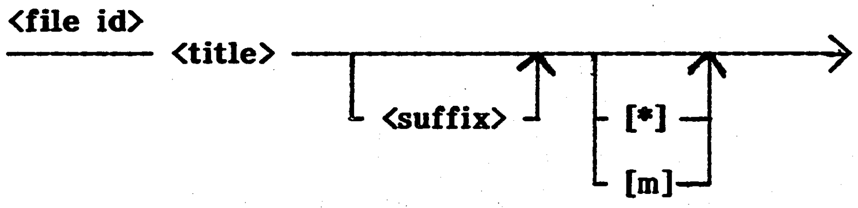
Examples of file names:

```
FORZ.TEXT
SHELL.1.TEXT[*]
*SYSTEM.WRK.TEXT
SYSTEM.MODULA
MACRO:PEST.FOTO[27]
STUFF.DATA
```

4.4.3 File Name Syntax







All spaces and control characters are ignored, and all lower case characters are mapped to upper case. The following characters should not be used in a file name: '$', '=', '?', '[', and ','. The filer uses them as wildcard characters.

A volume name may contain any characters except '#' or ':' and can be up seven characters long. A file identifier may contain any character (except '[') and can be up to fifteen characters long.

5 File Manager

The **file manager** ('filer' for short) is used to perform the following tasks:

- Managing disk files.
- Displaying the files stored on a disk volume.
- Managing work files.
- Managing disk volumes.
- Detecting and fixing damaged disks.

Section 5.1 describes the filer promptline and explains how to respond to certain filer commands.

Section 5.2 describes file names in the filer. File names may contain 'wildcard' characters which allow a single file name to specify several disk files. Wildcards let you manipulate a number of disk files with a single filer command.

Section 5.3 describes the filer commands.

NOTE- Before you use the filer, you should understand how the Modula file system works. The file system is described in chapter 4.

5.1 Filer Prompts

The filer promptline appears as:

L(dir, T(rans, R(em, C(hng, D(ate, P(refix, V(ols, Z(ero, Q(uit ?

This promptline does not display all the filer commands. The other commands appear if you type '?':

K(rnch, M(ake, B(adBlk, E(xdir, G(et, S(ave, W(hat, N(ew, X(amin ?

Most filer commands accept one or two file names as arguments; however, you can type in as many files names as you want, and the filer command will operate on each file specified. Lists of file names are separated by commas. Filer commands that accept a single file name read file names from the list one at a time. Filer commands that accept file name pairs (e.g. T(ransfer) read file names from the list two at a time.

5.2 File Names

File names are entered in the usual UCSD Pascal file name syntax. The G(et and S(ave commands do not require file suffixes (e.g. 'MYFILE'); all other commands expect complete file names (e.g. 'MYFILE.TEXT').

The character '=' is allowed to appear in file names. It is called a **wildcard** because it allows a single file name to match a number of disk files. For example, removing the file name 'F=.TEXT' causes the filer to remove all text files whose names begin with 'F'. Removing the file name '=' removes all the files on your disk! (But not without a warning prompt.)

NOTE- A file name may contain more than one wildcard (e.g. '=STUFF=').

An occurrence of '=' matches any string of characters in a file name — including the empty string. In commands that accept file name pairs, an '=' in the second file name is replaced with the character string matched by the wildcard in the first file name.

Example of '=' wildcard:

Transfer what file? BACKUP:=.CODE,BOOT:SYSTEM.=

All the code files on the BACKUP volume are transfered to the BOOT volume. The code files on BACKUP become system files on BOOT; for instance, MODULA.CODE is renamed SYSTEM.MODULA.

The character '?' works like '=', but causes the filer to write a prompt for each file that matches the file name. '?' allows you to skip some of the files that match the wildcard file name.

Example of '?' wildcard:

Remove what file? ?.CODE

The R(emove command then writes a series of prompts:

'Remove <file name>.CODE?'

... for every code file on the volume. Typing 'y' or 'Y' removes the file and generates the next 'Remove?' prompt; 'n' or 'N' preserves the named file. <escape> terminates the entire R(emove command.

The '$' character is used in commands that accept file name pairs. When used as the second file name, '$' denotes the first file name (which can contain wildcards).

Transfer what file? FILER.CODE,#5:$

5.3 Commands

This section presents a command overview; commands are grouped according to their function. Sections 5.3.1 through 5.3.18 describe each command in detail; commands are ordered alphabetically for ease of reference.

Disk File Commands

C(hange	Change file names.
T(ransfer	Transfer files from one disk to another.
R(emove	Remove files.
M(ake	Create new files.

Work File Commands

G(et	Establish a new work file.
S(ave	Save the current work file.
N(ew	Remove the current work file.
W(hat	Display work file status.

Disk Volume Commands

L(dir	List files on a disk volume.
E(x dir	Extended listing of files on a disk volume.
D(ate	Change the system date on all disk volumes.
K(runch	Merge all free spaces on a disk volume.
P(refix	Change the prefixed volume name.
V(olumes	List online volumes.
Z(ero	Initialize a disk volume.

Disk Repair Commands

B(ad blocks	Check for damaged disk blocks.
X(amine	Repair damaged disk blocks.

5.3.1 B(ad blocks

The B(ad blocks command checks disk volumes for blocks that can no longer store information reliably.

The following prompt appears:

Bad block scan of what volume?

Type in the name of the volume to to be scanned.

B(ad blocks check each block on the disk volume. If a block is bad, a message appears identifying it. At the end of the scan, the number of bad blocks is printed.

NOTE- The X(amine command is used to repair bad blocks (or to mark them 'bad' if they cannot be repaired).

5.3.2 C(hange

The C(hange command changes the name of a disk file or volume.

C(hange requires a pair of file names: the name to be changed followed by the new name.

If a wildcard is used in the first file name, then it also must appear in the second; strings matched by the first wildcard are substituted in the second.

Examples of C(hange:

Change what file? =.BACK,=.TEXT

Changes all backup files on the volume to text files.

Change what file? MYDISK:,ARCHIVE:

Renames the disk volume MYDISK to ARCHIVE.

5.3.3 D(ate

The D(ate command sets the system date.

```
DATE SET: <1..31>-<JAN..DEC>-<00..99>
Today is 5-Dec-82
New date?
```

To save the current date, type <return>. To set a new date, type it in the indicated form. You do not have to type in all of the date fields; for instance, typing '6-Dec' followed by <return> preserves the current year.

The system assigns the current date to newly created or modified disk files. File dates can be viewed with the filer commands L(dir and E(x dir.

5.3.4 E(x dir

The E(x dir command displays an extended directory listing of the specified disk volume.

The following prompt appears:

Dir of what volume?

Type in the name of the disk volume to to be listed.

E(x dir displays all files and free spaces on a volume. The listing displays the following information:

- file name
- file size (in blocks)
- file date
- starting block number
- # valid data bytes in last block
- file type

NOTE- See the L(dir command for more information on E(x dir.

Sample of extended directory listing:

```
MANUAL:
STARTDOC.TEXT     12  29-Sep-82 716    512   Textfile
FILER.TEXT         4   3-Dec-82 728    512   Textfile
< UNUSED >       190            732
FILER.1.BACK      62   3-Dec-82 922    512   Textfile
FILER.1.TEXT      58   3-Dec-82 984    512   Textfile
< UNUSED >      1238           1042
6/6 files<listed/in-dir>, 52 blocks used, 28 unused, 38 in largest
```

5.3.5 G(et

The G(et command specifies an existing disk file as the work file.

If an unsaved work file exists, the following prompt appears first:

> **Throw away current workfile?**

Type 'y' to discard the existing work file; type anything else to escape.

The following prompt appears:

> **Get what file?**

The file name is entered without a file suffix.

NOTE- G(et does not actually create a new disk file; it merely specifies an existing file as the source of the new work file. See 3.2 for more information on work files.

5.3.6 K(runch

The K(runch command merges the free spaces on a disk volume into one contiguous area.

The following prompt appears:

Crunch what volume?

Type in the name of the volume to be crunched. The next prompt appears:

From the end of disk, block 420? (Y/N)

Type something other than 'y' or 'n' to escape. Type 'y' to commence normal crunching — all disk files will be moved forward on the disk volume, leaving one large free space at the end of the disk.

If you want to crunch the files towards the end of the disk, type 'n'. The following prompt appears:

Starting at block #?

Type a non-number to escape. Typing a block number causes all files beyond the specified block number to be moved to the end of the disk. This so-called 'backwards crunch' is used to open free spaces between existing files; it is useful for inserting new files in front of the existing files on a volume.

NOTE- Be sure to do a B(ad blocks scan before crunching a disk volume; files can be lost by writing them over (unmarked) bad blocks. K(runch avoids overwriting disk blocks already marked 'bad'.

WARNING- K(runch is a critical operation. if the disk volume is removed (or the system fails) during crunching, disk files on the volume may be ruined.

5.3.7 L(dir

The L(dir command lists the disk files contained on the specified disk volume. The directory listing can be written to the console or to a text file.

The following prompt appears:

Dir of what volume?

Type in the name of the volume to be listed.

Wildcards can be used to list subsets of the files on a volume; the wildcard file name is appended to the volume name. For example, '=.TEXT' lists all the text files on the prefixed volume. 'BOOT:SYSTEM.=' lists all the system files on the volume named BOOT.

Directory listings display the following information:

- file name
- file length (# blocks)
- file date

Directory listings normally go to the console, but they can also be written to text files by appending ',<text file name>' to the prompt response. For example, 'SYSTEM.=,LISTING.TEXT' lists all system files on the prefix volume and writes the listing to the text file LISTING.TEXT. '#5:,printer:' prints a directory listing of the disk volume mounted in unit 5.

5.3.8 M(ake

The M(ake command creates disk files.

The following prompt appears:

Make what file?

Type in the name of the file to be created.

File length specifiers are used to control the size and location of created files. For instance, making 'JUNK[25]' creates a file named JUNK in the first free space larger than 24 blocks.

NOTE- M(ake is useful for recovering accidentally removed files. Just M(ake the proper size (and type) of file on top of the free space where the file used to reside on the disk.

5.3.9 N(ew

The N(ew command removes the current work file.

If an unsaved work file exists, the following prompt appears:

Throw away current workfile?

Type 'y' to discard the current work file. Type any other character to escape.

NOTE- N(ew removes the work file SYSTEM.WRK if it exists. See 3.2 for more information on work files.

5.3.10 P(refix

The P(refix command sets the prefixed volume name.

The following prompt appears:

> **Prefix is MYDISK:**
> **Set prefix to?**

Type a volume name to set the prefixed volume. Type <return> to escape. Note that the specified volume does not have to be online.

If you type a unit number, the disk volume in that unit becomes the prefixed volume. If the unit does not contain a disk volume, the prefixed volume is defined to be whatever disk volume is subsequently mounted in that unit.

5.3.11 Q(uit

The Q(uit command exits the filer.

NOTE- Be sure to replace the system disk before Q(uitting.

5.3.12 R(emove

The R(emove command removes files from a disk volume.

The following prompt appears:

> **Remove what file?**

Type in the name of the file(s) to be removed.

The following prompt then appears:

> **Update directory?**

Type 'y' to permanently remove the files. Type anything else to preserve them.

5.3.13 S(ave

The S(ave command saves the current work file in a disk file.

If the work file was obtained from an existing disk file, the following prompt appears:

Save as MYFILE?

Type 'y' to write the work file to the indicated disk file name (thus removing the old version). Type 'n' to save it under a different name. Type anything else to escape.

If the work file is newly created (or if you typed 'n' to the last prompt), the following prompt appears:

Save as what file?

Type in the title that the work file should be saved as. Be sure to leave off the file suffix.

See 3.2 for more information on work files.

5.3.14 T(ransfer

The T(ransfer command copies disk files from one disk to another.

T(ransfer accepts a file name pair: the first specifies the source file, the second specifies the destination file name.

T(ransfer performs the following tasks:

- Copying disk files onto different disk volumes.
- Transferring files to the console or printer.
- Moving files around on a disk volume.
- Copying disk volumes onto different disks (though the Backup utility is more reliable).

Examples of T(ransfer:

> **Transfer what file?** junk.data,$[25]

Moves the file JUNK.DATA to the first free space larger than 24 blocks.

> **Transfer what file?** *=.CODE,#5:$

Copies all code files on the boot volume to the disk volume in unit 5.

> **Transfer what file?** WORK:,#5:

Copies the contents of the disk volume WORK onto the disk in unit 5.

5.3.15 V(olumes

The V(olumes command displays all online volumes and indicates the prefixed and system volumes.

Example of V(olumes:

```
Vols on-line:
  1   CONSOLE:
  2   SYSTERM:
  4 * BOOT:      [2280]
  5 p MANUAL:    [ 494]
  6   PRINTER:
  7   REMIN:
  8   REMOUT:
System vol - BOOT
Prefix vol - MANUAL
```

Disk volumes display their size (in blocks). '*' indicates the system boot volume. 'p' indicates the prefixed volume. '#' indicates other online disk volumes.

5.3.16 W(hat

The W(hat command displays the name of the current work file.

5.3.17 X(amine

The X(amine command examines and repairs damaged disk blocks.

The following prompt appears:

> **Examine blocks on what volume?**

Type in the volume to examine. X(amine then lets you check a single block or a range of blocks. The following prompt appears:

> **From block?**

Type in the lower block number of the block range to be examined. The next prompt is:

> **To block?**

Type in the higher block number of the range to be examined. To check a single block, the 'from' and the 'to' block numbers can be the same.

If you are scanning over an existing disk file, the following prompt appears:

> **File(s) endangered:**
> **MYFILE.DATA**
> **Do you want to fix them?**

Type 'y' to repair the indicated blocks. Type anything else to escape. If a block is succesfully repaired, the following message appears:

> **Block 253 may be ok**

If a block cannot be repaired, the following message appears:

> **Block 253 is bad**

If a disk block is unrepairable, X(amine asks if you want to mark it as a 'bad block'. Bad blocks appear in directory listings as files of type 'Bad'. Note that the K(runch command moves files around bad blocks to prevent them from being written over the damaged areas on a disk.

WARNING- X(amine cannot restore the data lost when a block goes bad. Repairing a block ensures only that it works reliably the next time it is written to.

5.3.18 Z(ero

The Z(ero command initializes new disk volumes.

Z(ero works differently depending on whether you are initializing a brand new disk or zapping an existing disk volume.

If you are zeroing a brand new disk, the following prompt appears:

Zero dir of what vol?

Type in the unit number containing the disk. The next prompt appears:

# of blocks on this disk?

Type in the number of blocks on the disk. (This number depends on the size and format of the disks you are using.) The next prompt is:

New volume name?

Type in the name of the new disk volume. Z(ero then asks you if the volume name is correct. Type 'y' to zero the disk. Type anything else to escape.

NOTE- New disks may require formatting before they can be Z(ero'd.

If you are zeroing an existing disk volume, the following prompt appears:

Destroy ASEMAN?

Type 'y' to continue. Type anything else to escape. The next prompt is:

Are there 494 blocks on this disk?

This is the size of the disk volume about to be zapped. Type 'y' to use it as the size of new volume. Typing anything else causes Z(ero to ask for a block size. Z(ero then proceeds to ask for the new volume name (as described above).

6 Batch Command Interpreter

Command files are sequences of system commmands and data stored in text files. When a command file is submitted for execution, the system automatically performs the operations specified by the command file. Command files are useful for automating repetitious tasks (such as recompiling a suite of related programs).

Section 6.1 explains how to submit command files for execution.

Section 6.2 describes how command files work.

Section 6.3 explains how command files can be automatically invoked.

Section 6.4 describes the command file syntax.

Section 6.5 presents some example command files.

NOTE- The system disk includes the command file 'BATCH.DEMO.TEXT'. When submitted for execution, it demonstrates the use of command files.

6.1 Submitting Command Files

Command files are submitted for execution with the B(atch command. B(atch invokes the command interpreter program SYSTEM.BATCH which resides on the system boot volume. The following prompt appears:

Filename?

Type in the name of the command file to execute. (Be sure to leave off the file suffix '.TEXT'.) Type <return> to escape.

The system normally begins executing the commands at the front of a command file. You can specify command file execution to begin at other locations in the file by typing a command file **target.** A target consists of a command file name followed by the name of a label in the command file where you want execution to start. See 6.4.2 for more information on targets.

You can also pass string parameters to a command file. See 6.4.4 for details.

6.2 Command File Execution

The command interpreter reads through the command file, translating it into a series of system commands and input data. If the command file contains an error, the command interpreter terminates without submitting it for execution.

NOTE- If a command file goes into an endless loop, type the break key to stop the command interpreter.

When the command interpreter finishes translating a command file, the resulting commands and data are stored into the keyboard type-ahead buffer. The command interpreter returns control to the system prompt, which proceeds to read the queued characters as if they were typed in by hand.

NOTE- The keyboard type-ahead buffer holds up to 128 characters — a command file thus cannot generate more than 128 characters of commands and data at a time. (This is not a big problem — see the **b** command for details.)

WARNING- Command files can run amuck if the queued commands and data do not match the actual system prompts.

6.3 Automatic Command File Execution

Command files are normally executed by invoking the B(atch command and typing in a command file name. You can also define command files which are automatically executed when the system is booted or when the B(atch command is invoked.

A command file named 'PROFILE.TEXT' is automatically executed when the system is booted.

A command file named 'EXEC.TEXT' is automatically executed when B(atch is invoked. Note that automatic execution does not occur if any keyboard input is queued.

NOTE- The command files 'PROFILE' and 'EXEC' must reside on the prefixed volume in order to be automatically executed.

6.4 Command Files

A command file is a text file which contains a series of commands and labels. Each text line contains a command or label (which must appear as the first word on the line). Text lines that do not start with a command or label are treated as comments. Commands are described in 6.4.1.

Commands accept **targets** or **textlines** as arguments. The flow-of-control commands use targets as labels to jump to within the command file. Targets are described in 6.4.2. Textlines contain text that is either written to the console or queued as system input. Textlines are described in 6.4.3. Parameters may be passed to a command file when it is invoked. Parameters are described in 6.4.4.

NOTE- The command interpreter ignores blanks except in parameter lists (where they serve as parameter separators) and after the commands **read, write, writeln**, and **t.** Also, blanks should not appear in targets.

6.4.1 Commands

stk <textline>

Example: stk fe#5|n

Saves the text for queuing in the type-ahead buffer.

b <target>

Example: b myfile/nextsub

Saves a B(atch command to the specified target for queueing in the type-ahead buffer. This command allows command files to regain control of the system after submitting a set of commands and data. If the target does not contain a file name, control returns to the current command file.

run

Example: run

If a **call** command is outstanding, the command interpreter returns to the command following the **call.** If no **call** is outstanding, all saved text is written to the keyboard type-ahead buffer and the command interpreter terminates.

write <textline>

Example: write Hi, there!

Writes a message (but no carriage return) to the console.

writeln <textline>

Example: writeln I am a line of text.

Writes a message and carriage return to the console.

t <textline>

Example: t Yet another line of text...

t is equivalent to **writeln**, but can print longer textlines.

read <textline>

Example: read Enter file name:

Writes a message to the console, then reads from the keyboard until <return> is typed. The keyboard input is accessible with the '?' command (see 6.4.3 for details).

goto <target>

Example: goto loopstart

Causes the command interpreter to jump to the indicated label.

call <target> p1..p9

Example: call startsub

Causes the command interpreter to jump to the indicated label. When **run** is executed, control returns to the command following the **call** command. Calls can be nested up to 18 levels deep. The parameters p1 thru p9 are described in 6.4.4. Note that the symbol '|?' (6.4.3) can be used to return values from calls.

set <digit> <str>

Example: set 3 mystring

Assigns a value to one of the string parameters. Parameters are addressed by the digits 1 thru 9. Note that '?' can be used to assign values to the symbol '|?' (6.4.3).

equ <a> <b> <target>

Example: equ |3 mystring startsub

If the string in <a> equals the string in <b>, the command interpreter jumps to the specified target. The other comparison operations are also available: **neq, les, leq, geq, gtr.**

verbose

Example: verbose

Verifies each command before executing it. The command name is written to the console. Type <return> to execute the command. Type <escape><return> to terminate the command interpreter. **verbose** is used for testing new command files.

quiet

Example: quiet

Disables the **verbose** command.

6.4.2 Targets & Labels

Targets are used as arguments to the **goto** and **call** commands and to the S(ubmit prompt. Targets indicate the location in a command file where command interpretation is to continue. Target locations are either **labels** or **line numbers.**

Line numbers in a command file are zero-based; thus, the third line in a command file is on line 2. Line numbers are specified in a target by a backslash ('\'). For example, the command '**goto** \12' jumps to the thirteenth line in the command file. Note that line numbers are intended for use by the command interpreter — people should use labels instead.

Labels are (non-command) names which appear at the front of a line. Labels are specified in a target by a slash ('/'). For example, the command '**goto** /loopstart' jumps to the label 'loopstart' in the command file.

Targets can specify locations in other command files. For example, the command '**goto** profile/subroutine' causes the command interpreter to jump to the label 'subroutine' in the command file 'PROFILE.TEXT'. The file suffix '.TEXT' must not appear in the file name. If only a file name is specified (e.g. 'goto profile'), the command interpreter jumps to the first line in the indicated command file.

NOTE- The default command file name in a target is the name of the host command file.

6.4.3 Text Lines

Text line parameters are defined to extend from the command name to the end of the line.

Text passed to the **stk** command is handled specially (because it is queued as system input). Nonprinting characters are represented by two-character sequences: the escape character '|' followed by a command character. Note that blanks are ignored — they can be specified with the character sequence '| '.

Command characters are defined as follows:

' '	<space>	'n'	<return>
'|'	| (single '|')	'b'	<backspace>
'u'	<up>	'!'	<escape>
'd'	<down>	'@'	<line-delete>
'l'	<left>	't'	<tab>
'r'	<right>	'?'	response to last **read**
'0'..'9'	<params>	'k'	redirect to keyboard
'e'	<etx>		

The command characters '?' and 'k' have special properties.

An occurrence of '|?' in a textline is replaced with the input from the last **read** command.

The command character 'k' should only be used in parameters to the **stk** command. When the system encounters an occurrence of '|k' while reading from the type-ahead buffer, it proceeds to read directly from the keyboard until ctrl-E is typed. When ctrl-E is typed, the system resumes reading from the type-ahead buffer.

Because '|k' allows intermixing of queued and direct keyboard input, command files can define automated tasks which also interact with the user. (See 6.5 for an example of '|k'.)

6.4.4 String Parameters

String parameters can be passed to command files or to subroutines invoked with the **call** command. Up to 9 parameters (numbered 1 thru 9) can be passed. Parameters are listed after the target. The default parameter value is the empty string. Parameters are referenced with the symbol '|x' (where 1 <= x <= 9). The number of parameters passed is contained in the symbol '|0'.

Example of parameter passing:

```
call foon/startsub do re mi fa so la te do
```

In the routine labelled by 'foon', occurrences of the symbol '|3' would be substituted with the string 'mi'.

6.5 Example Command Files

The first example is a listing of the command file 'BATCH.DEMO' provided with the system:

```
writeln line 0 executing
b /target
run

target
writeln target executing
writeln calling /t2
call /t2
writeln /t2 returned
writeln going to /t3
goto /t3

t2
writeln /t2 running
run

t3
writeln /t3 gone to
writeln
read Enter Text :
writeln You Typed '|?'
writeln
writeln end of test
run
```

Example of a directory lister:

```
t
t This command file runs forever...
t
loop
read List what volume?
stk f e |?,#1 |n q
b /loop
run
```

This command file repeatedly prompts for a volume and displays its directory on the console. Note that the target in the **b** command implicitly specifies the host command file.

Another example of a directory lister:

```
stk f e |k ,#1 |n q
b \0
run
```

In this example, the volume name is not specified until the filer's own directory listing prompt appears; the '|k' then redirects system input to the keyboard. Note however that the prompt response must be terminated by typing ctrl-E.

7 Shell Command Interpreter

The shell command interpreter is a collection of Modula-2 programs which provides a powerful "command shell" programming environment informally named "p-NIX". The shell offers the following features:

- Pipes
- I/O redirection
- Wildcards
- Various predefined shell commands

Section 7.1 explains how to use the shell.

Section 7.2 describes shell commands provided with the system.

Section 7.3 explains how to add new commands to the shell.

NOTE- I/O redirection is limited to the p-NIX commands and Modula-2 programs that perform standard I/O via the modules InOut or Texts. Other programs can be invoked from the shell, but cannot have their I/O redirected.

7.1 Using the Shell

The shell is invoked by typing the S(hell command on the system promptline. S(hell invokes the program SYSTEM.SHELL which resides on the system boot volume. The following prompt appears:

```
1>
```

The right arrow '>' indicates the shell is ready for a command. The number (0 in this case) indicates how many shell commands have been executed.

The simplest thing to do in the shell is to invoke a program; for instance, the utility program BACKUP.CODE can be executed by typing 'backup' and a carriage return.

The shell becomes more useful when you learn how to use the shell commands. Shell commands are actually programs which perform specific tasks; for instance, the shell command 'ls' lists the contents of the disk directory.

7.1.1 Program Results

If a program terminates abnormally, the shell displays a message on the screen explaining what happened. If you attempt to execute a program which does not exist, the shell displays the message 'Missing program'. Programs terminated by a HALT call display the message 'Program HALT'. Programs terminated abnormally display the message 'Error return x', where x denotes the ordinal value of the program result.

7.1.2 I/O Redirection

All shell commands read from the standard input file and write to the standard output file. Standard input and output defaults to the system console, but can be redirected to disk files. For instance, the 'ls' command mentioned above writes its output to the standard output file, so typing 'ls' writes the directory listing to the screen. However, if you type

```
1> ls >myfile
```

... the directory listing is written to a file named 'myfile'. The symbol '>' after the ls command redirects the standard output to the named file.

Similarly, the symbol '<' is used to redirect the standard input. For instance, the shell command 'cat' copies the standard input to the standard output. If you just type

```
2> cat
```

... the system merely waits for some characters to be typed (remember that the standard input is the keyboard), then echoes the characters to the screen (the standard output). However, typing

```
3> cat <myfile >newfile
```

... copies the contents of the file 'myfile' into the file 'newfile'.

NOTE- Standard input from the keyboard is terminated by typing the <eof> key.

7.1.3 Command Arguments

Many shell commands accept a list of arguments after the command. For instance, typing

```
4> cat a b c >big
```

... copies the contents of the files 'a', 'b', and 'c' to the file 'big'. Note that arguments are usually treated as input file names; they are always processed left to right.

7.1.4 Wildcards

Wildcards let you type in a single file name argument that matches many actual file names. For instance, typing

```
5> rm f=
```

... removes all files starting with the letter 'f'. The wildcard character '=' can be used more than once in a file name; for instance, '=s=t=' matches all files containing an 's' and a 't'.

7.1.5 Pipes

Shell commands can be linked together so the output of one command becomes the input of another. For instance, typing

```
6> ls | sort | more
```

... writes a directory listing sorted by file name to the screen. If the bottom of the screen is reached, the prompt 'More?' appears; typing 'y' continues the sorted directory listing at the top of the screen.

The symbol '|' is called a **pipe**; it is used to connect the shell commands together. In this example, the ls command writes a directory listing to the standard output. The sort command reads the directory listing as input and writes the sorted listing to its standard output. Finally, the more command echoes the sorted listing to the console, but inserts the 'More?' prompt every twenty-four lines and pages the screen.

NOTE- Pipes are implemented as anonymous intermediate files written to the system boot volume. When each command finishes writing to the intermediate file, the shell starts the next command using the intermediate file as the standard input. Pipe performance can be greatly enhanced by using a RAM disk for the system volume. Note that extremely large pipe files may exceed disk capacity.

7.2 Shell Commands

This section describes the commands provided with the shell command interpreter.

All commands except rm and cp work with text files only; rm and cp can be used on all files.

NOTE- All shell commands use the following naming convention: the file suffix '.TEXT' is automatically appended to all file names unless they end with a period (e.g. 'MYDATA.').

7.2.1 cat

The **cat** command copies the standard input to the standard output. The cat command may be followed by a list of file names; in this case, it writes the catenation of all the files to the standard output. For instance, "cat a b c" writes the catenation of the files a, b, and c to the standard output.

7.2.2 cp

The **cp** command copies the file named by the first argument to the file named by the second argument. Note that cp works with all types of files. For instance, "cp myfile #5:myfile" copies the file "myfile" to the volume mounted in unit 5.

7.2.3 date

The **date** command writes the current date to the standard output (e.g. 'Today is January 14, 1983').

7.2.4 echo

The **echo** command writes its command arguments to the standard output.

7.2.5 ed

The **ed** command invokes the editor. If an argument is listed, it is used as the file name to edit; for instance, 'ed stuff' edits the file 'stuff.text'.

7.2.6 f

The **f** command invokes the filer.

7.2.7 grep

The **grep** command searches the standard input for occurrences of the character string passed as the first argument and writes all lines containing the string to the standard output. The string argument may be followed by a list of file names; in this case, grep searches through all of the listed files and prefixes each output line with the name of the file from where it came. For instance, "grep MODULE test= >matches" searches for occurrences of the word "MODULE" in all files whose names begin with "test". Text lines containing "MODULE" are written to the file "matches".

7.2.8 ls

The **ls** command lists the files on the prefixed disk volume. The ls command has three options. **ls -l** lists file attributes along with the file names. **ls -e** lists the disk free spaces along with the files. **ls -el** does both. The ls command may be followed by a list of file or volume names. If a file name is listed, ls lists any files on the prefixed volume that match the name. If a volume name is listed, ls lists all files on that volume.

7.2.9 mc

The **mc** command invokes the compiler. If an argument is listed, it is used as the input file name; for instance, 'mc textsd' compiles the file 'textsd.text'.

7.2.10 mem

The **mem** command writes the number of words of memory available and the address of the heap top to the standard output (e.g. 'Memavail=58440, NP=5728').

7.2.11 more

The **more** command echoes the standard input to the terminal, and displays the prompt 'More?' when the output reaches the bottom of the screen. Typing 'y' or <return> clears the screen and redisplays the next 24 lines of output; typing 'n' terminates the more command, stopping the screen output.

7.2.12 mv

The **mv** command changes the name of a file. The file named by the first argument is changed to the file name passed as the second argument. For instance, "mv foon yeen" changes the name of the file FOON.TEXT to YEEN.TEXT.

7.2.13 rm

The **rm** command removes the specified files. Note that rm works with all types of files. For instance, "rm *system=" removes all the system files from the boot disk.

7.2.14 sh

The **sh** command invokes the shell (recursively).

7.2.15 sort

The **sort** command sorts the lines read from the standard input and writes them to the standard output. Lines are sorted lexicographically. The sort command may be followed by a list of file names; in this case, it writes the sorted catenation of all the files to the standard output. For instance, "sort list1 list2 >final" sorts the lines contained in the files list1 and list2, and writes the sorted output to the file "final".

7.2.16 wc

The **wc** command counts the number of words, lines, and characters in the standard output and writes the totals to the standard output (e.g. '2 lines, 4 words, 17 chars'). The wc command may be followed by a list of file names; in this case, wc prefixes the totals for each file with the name of the file they belong to and writes the totals out on separate lines. For instance, "wc chap= >lexdata" performs a word count on all files beginning with "chap" and writes the results to the file "lexdata".

7.3 Adding New Shell Commands

The shell commands provided with the system are individual Modula-2 programs bound into the program library file named SYSTEM.SHELL. The contents of SYSTEM.SHELL may be examined by running the Modula-2 library manager utility. Adding a new shell command merely requires writing the appropriate Modula-2 program and adding it to the program library file. (In fact, it does not have to be added to the library file to be called, but the library file is a nice place to keep commands.)

All shell commands perform their I/O through the text files Texts.input and Texts.output. The shell itself redirects these files.

Shell commands gain access to the command arguments by importing the library module Args which appears in SYSTEM.SHELL. When the shell processes a command, it stores the command arguments in individual string variables pointed at by the array variable ArgV. The number of arguments passed is stored in ArgC.

NOTE- Shell commands should conform to the file naming conventions described in 7.2.

The library module Args:

```
DEFINITION MODULE Args; (* $SEG := 44; *)

  FROM Strings IMPORT STRING;

  EXPORT QUALIFIED StringPtr, ArgC, ArgV;

  TYPE StringPtr = POINTER TO STRING;
       ArgRange = [0..255];

  VAR ArgC: ArgRange;
      ArgV: POINTER TO ARRAY ArgRange (* 0..ArgC *) OF StringPtr;

END Args.
```

8 Pascal Compiler

The Pascal compiler is a one-pass recursive descent compiler for the language VS Pascal. VS Pascal is a dialect of standard Pascal which provides many of the UCSD Pascal extensions.

NOTE- VS Pascal provides all version II UCSD Pascal features except UNITs, external procedures, long integers, and record comparison. Note also that it enforces standard Pascal's 'name' type equivalence rather than UCSD Pascal's weaker notion of 'structural' type equivalence.

Compiler operation is explained in 8.1.

Compile options are described in 8.2.

The VS Pascal intrinsics are presented in 8.3.

Differences from standard Pascal are described in 8.4.

8.1 Operation

The compiler is invoked by typing the C(omp command on the system promptline. (It can also be run as a user program.)

If a workfile exists, the compiler automatically begins compiling it; otherwise, the following prompt appears:

Compile what file?

Type in the name of the file to compile (don't type the .TEXT suffix). The output file prompt appears next:

To what file?

Type in the name of the output file (again, the .CODE suffix is unnecessary). If you type <return>, the output file is given the same name as the source file.

The compiler displays various information on the screen while it compiles a program: the name of each procedure, the line number on which it occurs, and the amount of memory left.

If a syntax error is detected, the compiler displays the source line where the error occured; the symbol where the error was detected is pointed at by '<<<<'. The error message also displays the line number in the source program where the error occurred, and the syntax error number (syntax error messages are listed in Appendix 4).

Three options are available at this point. Typing <space> causes the compiler to continue compiling. Typing <esc> terminates the compiler. Typing 'E' invokes the editor; when the file is read in, the editor automatically positions the cursor at the error location and displays the proper syntax error message.

NOTE- The editor displays textual syntax error messages only if the file SYSTEM.SYNTAX resides on the system disk. If the syntax file is missing, the editor displays only the syntax error number.

8.2 Compile Options

Compile options control both the compiler's operation and the nature of the produced code. Options appear as comments in the source program; they have the following form:

(*$<options>*)

Compile options consist of a capital letter followed either by a switch character ('+', '-', or '^') or a string parameter. The compile option letter must appear immediately after the dollar sign.

When followed by a '+', an option is said to be enabled; when followed by '-', it is disabled. Some options can be followed by '^'; this restores ('pops') the option to its previous setting.

More than one switch option can appear in a single comment; when they do, they are delimited by commas with no blanks in between (e.g. '(*$F-,R+*)'). Only one string option can appear in a comment.

NOTE- All compile options have default settings. These are described in the following sections.

8.2.1 I/O Checks

The generation of code for performing runtime I/O checks is controlled by the I compile option. Setting I- eliminates I/O checks; setting I+ generates I/O checks. Setting I^ restores the previous option setting. The default setting is I+.

I/O checks ensure that all I/O operations are successfully completed. If an I/O error occurs, the I/O check terminates the program with an I/O error. Programs compiled I- are expected to perform their own I/O checking using the IORESULT intrinsic.

Example of I/O check option:

(*$I-*)

8.2.2 Include Files

Text files can be 'included' into a source program with the I compile option. The string parameter contains the name of the text file to be included. Compilation terminates if an included file cannot be opened. Include files cannot be nested.

Example of include file option:

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

8.2.3 Compiled Listings

Compiled listings are produced with the L compile option. The string parameter contains the name of the listing file. The option must appear at the top of a program.

Example of list file option:

```
(*$L listfile.text *)
```

The L option can also be used as a switch option to selectively list parts of a program. Setting L- disables listing; setting L+ enables listing. The L switch option is ignored if an L string option has not been declared at the top of the program. Note that the L string option automatically sets L+.

The first column in a compiled listing displays the source line number. The second column is the segment number. The third column is the procedure number. If the character after the procedure number is a 'C', the line is a statement, and the value in the last column is the code offset of the beginning of the statement. If the character is a 'D', the line is a declaration, and the value in the last column is the data offset of the first variable on the line.

Compiled listings can be useful for debugging programs. See 3.4 for more information.

8.2.4 Quiet Compile

The Q compile option controls the compiler's console display. The compiler can be operated in the so-called 'quiet mode' by setting Q+ at the top of the program. In quiet mode, the compiler suppresses its normal console display and does not stop when a syntax error is discovered. The default setting is Q-.

Example of quiet compile option:

```
(*$Q-*)
```

8.2.5 Range Checks

The generation of code for performing range checks at runtime is controlled by the R compile option. Setting R- eliminates range checks; setting R+ generates range checks. Setting R^ restores the previous option setting. The default setting is R+.

Compiler-controlled range checks protect subrange assignment and array indexing.

Example of range check option:

```
(*$R-*)
```

8.2.6 System-level Compile

The U compile option controls whether a program is to be compiled at the system program lexical level or the user program lexical level. The U option is set at the top of a program. Setting U+ specifies the system level and also sets I- and R-. The default setting is U+.

NOTE- Programs compiled U- will execute properly only if they are structured to execute at the system level; otherwise, they will crash the system. A description of the system lexical level is beyond the scope of this manual.

Example of system-level option:

```
(*$U-*)
```

8.2.7 Separate Code & Data

The N compile option controls whether a program is to be run on a p-machine which stores its code and data on separate stacks. The N option is set at the top of a program. Setting N+ specifies separate code and data. Setting N- specifies mixed code and data. The default setting is whatever is appropriate for your machine.

WARNING- Programs compiled N- will crash the system if they are executed on a separate code and data machine. A description of the code file differences for separate code and data is beyond the scope of this manual.

Example of separate code and data option:

```
(*$N-*)
```

8.2.8 Byte Flipping

The compiler generates byte-flipped code files by setting the F compile option at the top of a program. Byte-flipped code files are executable only on processors of the opposite byte sex. Setting F+ causes byte-flipping. The default setting is F-.

NOTE- Programs compiled F+ on your system will crash if you try to execute them. A description of byte sex is beyond the scope of this manual.

Example of byte-flipping option:

```
(*$F+*)
```

8.3 VS Pascal Intrinsics

This section describes procedures that are predefined in VS Pascal: these are also known as **intrinsics.** VS Pascal intrinsics provide the following operations:

- Input and Output
- String Manipulation
- Byte Array Manipulation
- Miscellaneous

The following notation is used to describe the intrinsic syntax. Required parameters are listed along with the procedure identifier. Optional parameters are enclosed in brackets; default parameter values appear in braces on the line below.

The following terms are used in the intrinsic definitions:

ARRAY	PACKED ARRAY OF CHARacters
BLOCK	disk block (512 bytes)
BLOCKS	number of blocks (integer)
BLOCKNUMBER	disk block address (integer)
BOOLEAN	Boolean expression
CHARACTER	character expression
DESTINATION	packed character array (may be indexed)
EXPRESSION	part or all of an expression (specified below)

FILEID	file variable of type:
	FILE OF <type>;
	TEXT;
	INTERACTIVE;
	FILE;

INDEX	string index or packed character array index
NUMBER	expression of type INTEGER or REAL.
RELBLOCK	file-relative disk block address (0-based)

SIZE	number of bytes or characters (integer)
SOURCE	packed character array (may be indexed)
STRING	string expression (unless otherwise noted)
STRVAR	string variable
TITLE	file name (string)
UNITNUMBER	physical device number (integer)
VOLID	volume identifier (string)

8.3.1 Input and Output

```
PROCEDURE RESET (FILEID [,TITLE]);
PROCEDURE REWRITE (FILEID, TITLE);
```

REWRITE creates a new file for writing. RESET opens an existing file for reading and writing. The string parameter TITLE can specify either a disk file or serial volume for I/O.

RESET without the title parameter rewinds an (open) file. Calling RESET (with title) or REWRITE on an open file causes an I/O error.

```
PROCEDURE CLOSE (FILEID [,OPTION]);
```

CLOSE closes an open disk file. Four close options are available: 'LOCK', 'NORMAL', 'PURGE' and 'CRUNCH'.

CLOSE(F,NORMAL) closes the file. If F is a disk file opened with REWRITE, it is removed. NORMAL is the default option.

CLOSE(F,LOCK) closes the file. If F is a disk file opened with REWRITE, it is saved.

CLOSE(F,PURGE) closes the file. If F is a disk file, it is removed. If F is a serial volume, the volume will go offline.

CLOSE(F,CRUNCH) closes the file. If F is a disk file, the current file position becomes the end of the file; any file data beyond the current file position is deleted.

```
PROCEDURE READ[LN] ([FILEID,] STRVAR);
PROCEDURE WRITE[LN] ([FILEID,] STRING);
```

These procedures can be used only on files of type TEXT (FILE OF CHAR) or INTERACTIVE. The default files are INPUT (for READs) and OUTPUT (for WRITEs).

READ(F, STRVAR) reads all characters on a line except the carriage return and sets EOLN to TRUE. Note that any subsequent readstrings return the empty string until a READLN or READ (character) is called.

```
FUNCTION EOF [(FILEID)] : BOOLEAN;
FUNCTION EOLN [(FILEID)] : BOOLEAN;
```

EOF and EOLN return FALSE when a file is initially opened. The default file parameter is INPUT.

When writing to a file, EOF returns TRUE if there is no more room on the disk.

When a GET(F) or READ(F) call sets the file position to an EOLN or EOF character, EOLN(F) is set to TRUE and the character will be read as a blank.

When GET(F) or READ(F) sets the file position to an EOF character, EOF(F) is set to TRUE. When EOF(F) is TRUE, the contents of the file buffer are undefined.

EOF and EOLN work differently on files of type INTERACTIVE — see 8.4.10 for more information.

```
PROCEDURE GET (FILEID);
PROCEDURE PUT (FILEID);
```

GET and PUT are used only on files of type FILE OF <type>.

GET(F) reads the record from the current file position into the window variable F^ and increments the file pointer. PUT(F) writes the contents of the window variable F^ to the current file position and increments the file pointer.

PROCEDURE SEEK (FILEID, INTEGER);

SEEK(F,I) changes the file position so that the subsequent GET(F) or PUT(F) accesses the INTEGERth record of the file. File record numbers are 0-based. The contents of the file window are undefined after a SEEK. Note that a READ or WRITE must occur between successive SEEKs.

PROCEDURE PAGE (FILEID);

PAGE(F) writes a top-of-form character (ASCII ff) to the file F.

FUNCTION IORESULT : INTEGER;

IORESULT returns the I/O status result of the previous I/O operation. Appendix 2 describes the I/O result values.

FUNCTION BLOCKREAD (FILEID,ARRAY,BLOCKS,[RELBLOCK]) : INTEGER;
FUNCTION BLOCKWRITE (FILEID,ARRAY,BLOCKS,[RELBLOCK]) : INTEGER;
{SEQUENTIAL}

BLOCKREAD and BLOCKWRITE transfer integral numbers of blocks of data between a memory buffer and a file. The function result returns the number of blocks actually transferred. The number of blocks transferred comes back less than the number of blocks requested either when an I/O error occurs or the end of the file was read.

The file parameter must be of type FILE. The parameter ARRAY should be a multiple of 512 bytes and no smaller than the number of blocks requested for transfer. The parameter BLOCKS is the number of blocks to be transferred.

The optional parameter RELBLOCK specifies the file-relative block number where the transfer should start. (The first block in a file is block 0.) If the RELBLOCK parameter is omitted, block I/O is performed sequentially starting at the first block in the file. (Note that a random-access block I/O operation changes the file position.)

NOTE- BLOCKREAD and BLOCKWRITE do not perform any error checking, so IORESULT (or the number of blocks transfered) should be checked after each call.

PROCEDURE UNITCLEAR (UNITNUMBER);

UNITCLEAR resets the specified peripheral device to its initial state.

PROCEDURE UNITREAD (UNITNUMBER, ARRAY, LENGTH [,BLOCKNUMBER [,FLAGS]]);
PROCEDURE UNITWRITE (UNITNUMBER, ARRAY, LENGTH [,BLOCKNUMBER [,FLAGS]]);

UNITREAD and UNITWRITE perform low-level I/O to the online peripheral devices. The UNITNUMBER parameter specifies the device unit number. ARRAY can be any variable; it is used as the starting word address of the memory buffer. LENGTH indicates the number of bytes to transfer. The optional parameter BLOCKNUMBER is required only when accessing disk units; it indicates the starting block number of the data transfer.

The optional parameter FLAGS controls the mode of the I/O operation; though of type integer, it is treated as a bit array. If bit 2 of FLAGS is set, the I/O system does not expand blank compression characters (DLEs). If bit 3 of FLAGS is set, the I/O system does not append line feed characters to carriage returns. The default value of FLAGS is 0.

If bit 1 of FLAGS is set, I/O is performed in **physical sector mode.** BLOCKNUMBER is interpreted as a physical disk sector number. LENGTH must be set to zero — in physical sector mode, each I/O operation automatically reads only one sector. Track numbers are 0-based, sector numbers 1-based. The mapping between physical sector numbers and a disk's track and sector numbers is as follows:

```
PhysSect# = (Track# *SectorsPerTrack) + Sector# - 1;
```

NOTE- UNITREAD and UNITWRITE do not perform any error checking, so IORESULT should be checked after every operation.

PROCEDURE UNITSTATUS (UNITNUMBER, ARRAY, CONTROL);

UNITSTATUS returns status information on the specified unit.

The UNITNUMBER parameter specifies the device unit number. ARRAY is used to return status information. ARRAY can be any variable, but it should be an array or record of at least 30 words to contain the status data and any additional system-dependent information. CONTROL is an integer parameter which specifies whether input or output information is desired. If CONTROL = 1, UNITSTATUS returns input information; if CONTROL = 0, output information.

On serial units, UNITSTATUS sets the first word in ARRAY to the number of characters queued on the unit. UNITSTATUS returns 0 if there are no characters queued or if the unit's state cannot be determined.

On block-structured units, UNITSTATUS sets the first four words in ARRAY as follows:

- word 1: # of queued characters
- word 2: # of bytes per sector
- word 3: # sectors per track
- word 4: # of tracks on disk

8.3.2 String Manipulation

Strings can be manipulated either with the string instrinsics or by accessing them as character arrays. The zero'th character in a string variable is used as the length byte (and is inaccessible when range checking is on except via Length). The remaining characters in the array comprise the string. When accessing a string as a character array, be sure to stay within the (dynamic) bounds of the string and do not set the length byte inappropriately.

FUNCTION CONCAT (STRING,STRING,...) : STRING

CONCAT returns a string that is the concatenation of its string parameters. Note that any number of string parameters can be passed (separated by commas).

FUNCTION COPY (STRING, INDEX, SIZE) : STRING

COPY returns a string containing SIZE characters copied from STRING, starting at position INDEX in STRING.

FUNCTION LENGTH (STRING) : INTEGER

LENGTH returns the number of characters in the parameter STRING.

FUNCTION POS (STRING1, STRING2) : INTEGER;

POS returns the starting position of the first occurrence of the parameter STRING1 in the parameter STRING2. If the string is not found, POS returns zero.

PROCEDURE DELETE (STRVAR, INDEX, SIZE);

DELETE deletes SIZE characters from the string variable STRVAR, starting at the position INDEX.

PROCEDURE INSERT (STRING, STRVAR, INDEX)

INSERT inserts the parameter STRING into the variable STRVAR, starting at the position INDEX in STRVAR.

8.3.3 Byte Array Manipulation

The byte array intrinsics are used for manipulating large amounts of byte-oriented data in memory. They must be used with care, as no type or range checking is performed. The SIZEOF intrinsic is often used with these intrinsics to accurately specify the number of bytes in an array parameter.

FUNCTION SCAN (LENGTH, PARTIAL EXPRESSION, ARRAY) : INTEGER;

SCAN starts at the byte address ARRAY and scans (up or down) through memory until either it has checked LENGTH bytes or it finds a character satisifying the expression PARTIAL EXPRESSION. SCAN returns the number of characters checked before the expression was satisfied.

The ARRAY parameter should be a packed character array; it can be subscripted to denote the starting address. If the LENGTH parameter is negative, SCAN scans backwards and returns a negative result. If PARTIAL EXPRESSION is satisfied by the character residing at the starting address, SCAN returns 0. If PARTIAL EXPRESSION is not satisfied, SCAN returns LENGTH.

The PARTIAL EXPRESSION is a distinctly nonstandard construct with the following form:

> <> or = followed by a character expression (e.g. "<> ch")

PROCEDURE FILLCHAR (DESTINATION, LENGTH, CHARACTER);

FILLCHAR fills memory with the character passed in the CHARACTER parameter, starting at the byte address DESTINATION and filling for LENGTH bytes. Negative LENGTH values are treated as zero.

PROCEDURE MOVELEFT (SOURCE, DESTINATION, LENGTH);
PROCEDURE MOVERIGHT (SOURCE, DESTINATION, LENGTH);

MOVELEFT and MOVERIGHT move LENGTH bytes from the byte address SOURCE to the byte address DESTINATION. MOVELEFT increments the source and destination addresses after moving each byte; that is, it moves data starting from the left end of the buffer. MOVERIGHT decrements the source and destination addresses after moving each byte; that is, it moves data starting from the right end of the buffer. Negative LENGTH values are treated as zero.

8.3.4 Miscellaneous

PROCEDURE GOTOXY (COLUMN, ROW);

GOTOXY moves the cursor to the specified integer coordinates. The upper left corner is (0,0); the lower left corner (0,<screenheight>-1).

PROCEDURE HALT;

HALT terminates the current program.

PROCEDURE EXIT (PROCEDURE);

EXIT terminates the current procedure. See 8.4.6 for details.

PROCEDURE MARK (VAR MARKPTR: ^INTEGER);
PROCEDURE RELEASE (VAR RELPTR: ^INTEGER);

MARK and RELEASE are used to manage the heap space. MARK stores the current heap top in MARKPTR. RELEASE cuts the heap top back to the address stored in RELPTR.

FUNCTION PWROFTEN (EXPONENT: INTEGER) : REAL;

PWROFTEN returns the ten raised to the EXPONENTth power. With 32-bit reals, EXPONENT must be an integer between 0 and 37. With 64-bit reals, it must be between 0 and 307.

FUNCTION SIZEOF (VARIABLE OR TYPE): INTEGER;

SIZEOF returns the number of bytes allocated for the specified variable (or a variable of the specified type). SIZEOF is particularly useful with the byte array intrinsics. Note that SIZEOF is evaluated at compile time.

PROCEDURE TIME (VAR HITIME, LOTIME: INTEGER);

TIME returns the current system time as defined by a 32-bit clock value which is incremented at 1/60th second intervals. The values returned in HITIME and LOTIME should be treated as unsigned values. TIME returns (0,0) on systems without clocks.

8.4 Differences From Standard Pascal

This section describes differences between VS Pascal and standard Pascal. (The standard referred to here is the PASCAL USER MANUAL AND REPORT (2nd edition) by Jensen and Wirth.) Differences fall into two categories:

- Extensions to standard Pascal
- Deviations from standard Pascal

8.4.1 Case Statements

In VS Pascal, if no label matches the value of the case selector, the next statement executed is the statement following the case statement. In standard Pascal, this case is considered an error.

```
PROGRAM fallthru;
VAR ch: CHAR;
BEGIN
  ch := 'a';
  CASE ch OF
   'b': WriteLn('hi there');
   'c': WriteLn('the character is a c');
  END;
  WRITELN('that''s all, folks');
END.
```

NOTE- VS Pascal accepts the OTHERWISE clause in CASE statements as a way to catch unspecified values.

8.4.2 Comments

VS Pascal considers the comments delimiters '(*' and '{' to be unique, thus permitting one level of nested comments:

```
{ I := 1; (* nested comment *) }
```

Standard Pascal considers the two different forms of comment delimiters to be equivalent, and thus does not permit nested comments.

8.4.3 Dynamic Memory Allocation

VS Pascal does not implement the standard procedure DISPOSE; instead, it provides the MARK and RELEASE intrinsics.

Dynamic storage is allocated in a stack-like structure called the 'heap'. NEW allocates variables on the top of the heap. To recover the storage occupied by a dynamic variable, it is necessary to call MARK before allocating it; MARK saves the current address of the top of the heap. Subsequent calls to NEW allocate dynamic variables 'above' the heap mark. RELEASE sets the top-of-heap pointer back to the heap mark established by MARK, releasing all subsequently allocated variables.

NOTE- Careless use of MARK and RELEASE can lead to 'dangling pointers' which no longer point to dynamic variables.

```
PROGRAM heapchop;
VAR heap: ^INTEGER;
   i,j,k: ^ARRAY[1..10] OF CHARACTER;
BEGIN
  MARK(heap);            (* save current heap position *)
  NEW(i);
  NEW(j);
  NEW(k);
  RELEASE(heap);         (* cut heap back to old position *)
END.
```

8.4.4 EOF and EOLN

When the console unit is used as an input text file, the end-of-file condition is set by typing the <eof> key (ctrl-C on most systems).

EOF(F) returns TRUE if F is closed. On text files, EOLN(F) is always TRUE when EOF(F) is true.

EOLN(F) is defined only for files of type TEXT or whose window variable is of type CHAR. EOLN becomes True after the end-of-line character <cr> is read.

8.4.5 Files

The predeclared files INPUT and OUTPUT are **interactive files,** which are declared with the predefined file type INTERACTIVE. Interactive files are similar to files of type TEXT, but the procedures EOF(F), EOLN(F) and RESET(F) are defined differently when reading from the console. See 8.4.10 for more information on interactive files.

The predeclared file KEYBOARD is an interactive file which can be used to read characters directly from the keyboard (characters are not echoed to the console screen as they are read.)

The VS Pascal intrinsics BLOCKREAD and BLOCKWRITE operate on **block files.** Block files are declared with type 'FILE'. (Note that the usual 'OF <type>' sequence is missing.) See 8.3.1 for more information on block files.

The VS Pascal intrinsic SEEK is used to randomly access record-oriented (non-text) files. See 8.3.1 for more information on SEEK.

VS Pascal does not permit READ or WRITE to access files of type other than TEXT or FILE OF CHAR. Only GET and PUT can be used to access record-oriented files.

8.4.6 GOTO and EXIT Statements

VS Pascal does not allow out-of-block GOTO statements; a GOTO can jump only to labels declared in the same procedure as the GOTO.

As an alternative to the out-of-block GOTO, VS Pascal provides an EXIT statement. EXIT accepts a procedure name as its parameter; what it does is transfer program control to the end of the named procedure. If EXIT is called in a recursive procedure, it exits the most recent procedure invocation. If exit names a procedure which has not been called, execution error 3 ('exit to uncalled procedure') occurs. EXIT(PROGRAM) terminates the current program.

```
PROGRAM ExitDemo;

  PROCEDURE GlobalProc;

    PROCEDURE Terminate;
    BEGIN
      EXIT(GlobalProc);  (* exits out end of GlobalProc *)
    END;

  BEGIN
    Terminate;
    WriteLn('This never prints');
  END;

BEGIN
  GlobalProc;
END.
```

8.4.7 Packed Variables

The VS Pascal compiler attempts to compress the machine representations of records and arrays when their type definitions are prefixed with the reserved word **PACKED.** Packing significantly reduces the amount of memory needed to store certain data types, but at the expense of slightly increased execution time and code size required for packed field access. Packing is syntactically allowed for all structured types, but affects only records and arrays.

Examples of packed variable declaration:

```
TYPE  manybits =  PACKED ARRAY [0..31] OF BOOLEAN;
      smallrec =  PACKED RECORD
                    a,b: CHAR;
                    i: INTEGER;
                  END;
```

Machine representations of the basic data types are as follows:

type	unpacked	packed
BOOLEAN	1 word	1 bit
CHAR	1 word	8 bits
INTEGER	1 word	1 word
REAL	2 words	2 words
SET OF x..y : y<16	1 word	(y+1) bits
subrange x..y : x>=0	1 word	(log2(y+1)) bits

Subrange types with negative lower bounds are not packable. Array and record subtypes are word aligned and thus unpackable. The compiler is limited to packing fields into single words; fields cannot be packed across word boundaries. Thus, records are packed only if they contain consecutively declared fields that can be packed into a single word, and arrays are packed only if their element types can be stored in 8 bits or less. Unpackable fields are referenced as unpacked data. (In records, this includes fields which cannot be packed because of adjacently declared unpackable fields.)

NOTE- Packed fields cannot be passed as VAR parameters. VS Pascal does not support the standard procedures PACK and UNPACK.

8.4.8 Procedure Parameters

VS Pascal does not support procedure (or function) parameters.

8.4.9 Program Headings

VS Pascal ignores parameters passed along with the program heading. External files are accessed by calling the RESET and REWRITE intrinsics. The standard files INPUT, OUTPUT, and KEYBOARD are predeclared and automatically opened in every program.

8.4.10 READ and READLN

Standard Pascal defines the statement READ(f,ch) as:

```
ch := f^;
GET(f);
```

Because this definition is unsuitable for interactive programming, VS Pascal defines interactive file type. Given an interactive file i, the statement READ(i,ch) is defined as:

```
GET(i);
ch := i^;
```

Note that the definition of READ for interactive files affects some other I/O intrinsics. When the interactive file i is first opened, the window variable i^ is not loaded with an initial value. EOLN and EOF work differently when reading from interactive files; they return true only AFTER the appropriate line or file terminator is read. On interactive input files, the end-of-line character <return> is returned as a blank character.

8.4.11 RESET and REWRITE

Resetting an interactive file i does not automatically initialize the window variable i^.

VS Pascal provides a second form of RESET for gaining access to external files. See 8.3.1 for details.

The VS Pascal intrinsic REWRITE is used to create external files. See 8.3.1 for details.

8.4.12 Segment Procedures

A VS Pascal procedure can be specified as overlayable by preceding its declaration with the reserved word SEGMENT. Segment procedures are used to divide large Pascal programs into separate disk-resident sections so that only part of the program is memory-resident at any one time.

A program can contain up to nine segment procedures. Segment procedures must be declared as the first procedures in a program. When a segment procedure is called, its code segment is read into memory from the disk. When a segment procedure returns, its code segment is released from memory.

WARNING- When a program calls a segment procedure, the disk volume containing the program code file must be online and in the same drive as when the program was executed. Otherwise, a segment procedure call crashes the program when the system tries to read the code segment from the missing disk.

```
PROGRAM SegDemo;

  SEGMENT PROCEDURE Initialize;
  BEGIN
    ...
  END;

BEGIN
  Initialize;     (* code read in for call *)
  MainProg;          (* code released after call returns *)
END.
```

8.4.13 Code Procedures

A code procedure is a procedure declaration whose body consists of a sequence of constants denoting P-code instructions and operands. This code sequence is substituted inline for each code procedure call.

Code procedures are used to perform low-level operations and to access routines defined in the Modula operating system. As in regular procedure calls, code procedure parameters are pushed onto the evaluation stack (in the order they appear) before the (inline) procedure code is executed.

WARNING- Code procedures must be used with utmost care, as any programming errors may cause the system to crash in mysterious ways. Be prepared!

Example of code procedure declaration:

```
PROCEDURE UpperByte(VAR i: INTEGER): CHAR;
(* word address pushed as parameter *)
CODE
  1;              (* load constant byte offset *)
  190             (* load byte as function result *)
END;
```

8.4.14 Sets

Sets of subrange types are restricted to positive values. Sets can be declared to contain up to 4080 elements. Set comparison and other operations are allowed only between sets that are either of the same base type or subranges of the same underlying type.

8.4.15 Strings

VS Pascal provides **strings** for manipulating variable-length character strings. String variables are declared with type STRING. String variables are implemented as packed character arrays with a length byte stored in the first character index (s[0]). Note that the length of a string can be obtained by calling the LENGTH intrinsic.

Every string variable has a maximum length. A string cannot grow longer than its string variable without causing execution error 13 ('string too long'). A string variable's maximum length is specified in its type declaration; the desired maximum length (enclosed in brackets) follows the type identifier STRING. For instance, STRING[20] declares a string type with a string length of 20 characters. The maximum string length is 255 characters. The default string length is 80 characters. String assignment is performed by the assignment statement, a READ statement, or one of the string intrinsics.

```
s := 'string assignment';
READLN(s);
s2 := CONCAT(s, '.TEXT');
```

Individual characters within a string variable are accessible by treating the string as a character array with an index range of 1 to the current string length. Indexing a string outside of this range causes execution error 1 ('invalid index').

All comparison operators accept strings. String comparison is lexicographical; that is, comparison is done left-to-right, and if strings are equal up to the length of the shorter string, the shorter string is less.

When reading into a string variable, all characters up to the end-of-line character are assigned to the string. Note that READLN(s1, s2) is equivalent to:

```
READ(s1);
READLN(s2);
```

See 8.3.2 for details on the VS Pascal string intrinsics.

8.4.16 WRITE and WRITELN

The standard procedures WRITE and WRITELN do not accept Boolean arguments.

When a string variable is written without a field width specification, the number of characters written equals the current length of the string. If the specified field width exceeds the string length, the appropriate number of leading blank characters are written. If the string length exceeds the field width specification, excess characters are truncated from the right-hand side of the string.

8.4.17 Array Comparison

The comparison operators = and <> accept arrays as arguments.

WARNING- Array comparison can be error prone with packed data structures (which may contain uninitialized data areas between packed fields).

8.4.18 Implementation Limits

- The maximum number of bytes of object code in a procedure is 1200.
- The maximum number of words in a data segment is 16383.
- The maximum number of elements in a set is 4080.
- The maximum number of segment procedures in a program is 9.
- The maximum number of procedures in a segment is 127.
- The maximum level of lexical nesting is 14.
- The Modula operating system does not detect integer overflow or the use of uninitialized variables. NIL pointer checking is available on some processors.

9 Yet Another Line Oriented Editor

YALOE (acronymous with 'yet another line-oriented editor') is a line-oriented text editor. It is designed to run on teletypewriters or — more commonly — terminals on unconfigured systems.

Section 9.1 explains how to start YALOE.

Section 9.2 explains how to enter commands and text.

Sections 9.3 thru 9.7 describe the various YALOE commands. Section 9.8 contains a command summary.

NOTE- Most people use YALOE just long enough to create the proper Gotoxy for their system, then drop it like a hot iron in favor of ASE.

9.1 Entering YALOE

YALOE is invoked by X(ecuting YALOE.CODE.

YALOE keeps the file being edited in its **text buffer.** Note that files must fit in the text buffer in order to be successfully edited.

If a work file exists, it is automatically read into the text buffer, and the following message appears:

Workfile GUMBY read in

If there is no workfile, this message appears instead:

No workfile read in

In this case, use the R(ead command to read a file into the text buffer.

9.2 Entering Commands and Text

YALOE runs in one of two modes: **command mode** or **text mode.** It starts off in command mode.

YALOE displays an asterisk ('*') when it is ready for a command. Commands are entered by typing command characters; they appear on the screen as they are typed.

In command mode, YALOE interprets all input as edit commands (spaces, returns, and tabs are ignored; commands can be in upper or lower case). You can enter commands one at a time, or you can type in a whole string of commands to be executed in sequence; in either case, they are not executed until you type <esc><esc>. Commands with text string parameters are separated by the <esc> that terminates the text string. When YALOE finishes executing the commands, it redisplays the command prompt ('*').

YALOE goes into text mode when you type a command that accepts a text string parameter. In text mode, all characters (including carriage returns) are treated as text until you terminate the text string by typing <esc>; YALOE then goes back to command mode.

NOTE- If an error occurs while YALOE is executing a comand string, the remaining commands in the command string are ignored.

NOTE- <esc> echoes a dollar sign ('$') when typed. The <esc> terminates the text string and returns control to Command mode. The examples in this chapter display <esc> as '$'.

Command Arguments

Some YALOE commands accept **command arguments.** Command arguments are characters which precede the command character. Command arguments specify repeat factors and/or the cursor direction.

The following definitions are used in the command descriptions:

n Denotes an integer. In YALOE commands that accept this argument, the default value is 1. If only a minus sign is present, the default value is -1. Negative argument values specify backwards cursor movement.

/ Denotes the integer value 32700. '-/' denotes -32700. '/' is used to specify a large repeat factor.

m Denotes an integer between 0 and 9.

O Denotes the start of the current line.

= Denotes the integer value '-n', where n is the length of the last text string parameter. '=' works only with the J(ump, D(elete, and C(hange commands.

9.3 Special Commands

YALOE defines certains keys as special commands.

<esc>

A single <esc> terminates a text string. A double <esc> executes the command string. <esc> echoes as '$'.

RUBOUT

RUBOUT (linedel) deletes the current line.

CTRL H

CTRL H (chardel) deletes a character from the current command string. Deletions are done right to left up to the beginning of the command string. CTRL H may be used in both command and text modes.

CTRL X

CTRL X causes the editor to ignore the entire command string currently being entered. YALOE responds by redisplaying the command prompt ('*'). Note that CTRL X takes out even multi-line command strings.

9.4 Input & Output Commands

The following commands control input and output: L(ist, V(erify, R(ead, W(rite, E(rase, and Q(uit.

L(ist

Syntax:

nL

Display the specified number of text lines.

Examples of L(ist:

*5L$$	Lists the five lines following the cursor.
*OL$$	Lists from the start of the current line up to the cursor.

V(erify

Syntax:

V

Redisplay the current text line.

R(ead

Syntax:

```
R<file title>$
```

Read the specified file into the text buffer starting at the current cursor position. <file title> is a text string containing a valid file title. If YALOE cannot find the file, it appends a '.TEXT' suffix and tries again.

WARNING- If the file read in does not fit in the text buffer, the buffer contents become undefined; i.e. the current edit session is lost.

W(rite

Syntax:

```
W<file title>$
```

Write the text buffer contents to the specified disk file. <file title> is a text string containing a valid file title; the file suffix '.TEXT' is automatically appended if it is not specified.

If the disk volume does not have enough space to contain the new file, the following error appears:

```
OUTPUT ERROR. HELP!
```

In this case, W(rite the file out to another disk volume.

Q(uit

The Q(uit command has the following forms:

QU	Quit and write to the work file.
QE	Quit and exit YALOE; do not save the text.
Q	Issue a prompt requesting one of the following options: U, E, or R. R returns to the edit session.

E(rase

Syntax:

E

Erase the screen.

9.5 Cursor Moving Commands

The following commands move the cursor: J(ump, A(dvance, B(eginning, G(et, and F(ind.

Cursor direction is specified by the command argument. For instance, the command '10J' moves the cursor forward 10 characters, while '-10J' moves the cursor backwards the 10 characters.

NOTE- Carriage returns and tabs are treated as single characters.

J(ump

Syntax:

nJ

Move the cursor the specified number of characters.

A(dvance

Syntax:

nA

Move the cursor the specified number of lines. The cursor is left at the beginning of the line. 'OA' moves the cursor to the start of the current line.

B(eginning

Syntax:

```
B
```

Move the cursor to the front of the text buffer.

G(et & F(ind

Syntax:

```
nG<target string>$
nF<target string>$
```

G(et and F(ind are synonymous. Starting at the current cursor position, the text buffer is searched for the n'th occurrence of the target string. (The argument sign determines the search direction.) If found, the cursor is left at the end of the text string. If not found, an error message appears and the cursor is left at the end of the buffer.

Examples of Cursor Moving Commands

The cursor position is indicated by boldface.

Here is the original text:

```
      The time has come
      the walrus said
      to talk of many things
```

*7J$$ moves the cursor forward 7 characters:

```
      The time has come
      the walrus said
      to talk of many things
```

*-A$$ moves the cursor up a line:

```
      The time has come
      the walrus said
      to talk of many things
```

```
*BGsaid$=J$$ moves the cursor to the front of the text
             buffer and search for the string 'said'.
             When the string is found, move the cursor
             to the front of the string:

    The time has come
    the walrus said
    to talk of many things
```

9.6 Text Changing Commands

The following commands change text: I(nsert, D(elete, K(ill, C(hange, and X(change.

I(nsert

Syntax:

I<text string>$

Insert the text string into the text starting at the current cursor position. The cursor is left after the last inserted character.

NOTE- The message 'Please finish' may appear while you are inserting a large text string. If this happens, type <esc><esc> to finish the current I(nsert command, then type 'I' and continue.

D(elete

Syntax:

nD

Delete the specified number of characters from the text buffer, starting at the current cursor position. The cursor is left at the character following the deleted text.

K(ill

Syntax:

```
nK
```

Delete the specified number of lines from the text buffer, starting at the current cursor position. The cursor is left at the front of the line following the deleted text.

C(hange

Syntax:

```
nC<text string>$
```

Replace the specified number of characters with the text string, starting at the current cursor position. The cursor is left after the changed text.

X(change

Syntax:

```
nX<text string>$
```

Replace the specified number of lines with the text string, starting at the current cursor position. The cursor is left after the changed text.

Examples of Text Changing Commands

*\/K$$	deletes all lines following the cursor.
*-4D$$	deletes the four characters preceding the cursor.
*B$Gpeace$=D$$	finds the first occurrence of 'peace' and deletes it.
*BGP$=CV$$	replaces the first occurrence of 'P' with 'V'.
*OCxyz$$	replaces the characters from the front of the line to the cursor with 'xyz'.
*-5XPOW$$	replaces all characters from the front of the fifth line back to the cursor with the string 'POW'.

9.7 Miscellaneous Commands

Miscellaneous commands include: S(ave, U(nsave, M(acro, N (macro execution), and '?'.

S(ave

Syntax:

```
nS
```

Copy the specified number of lines into the **save buffer,** starting at the current cursor position. YALOE prints a warning message and stops the S(ave command if the specified text is larger than the save buffer.

U(nsave

Syntax:

```
U
```

Copy the save buffer contents into the text buffer, starting at the current cursor position. YALOE prints a warning message and stops the U(nsave command if there is not enough room in the buffer for the inserted text.

M(acro

A **macro** is a single command that executes a user-defined command string. Macros are created with the M(acro command. A macro can invoke other macros (including itself recursively).

Syntax:

```
mM%<command string>%
```

The command argument m (an integer between 0 and 9) identifies the macro definition. The default macro number is 1. The command string delimiter ('%' above) is defined to be the character following the 'M'. It can be any character that does not appear in the macro command string itself. The second occurrence of the delimiter character terminates the macro definition.

Any character may appear within a macro definition, including a single <esc>.

YALOE prints the following message if an error occurs during macro definition:

```
Error in macro definition.
```

Example of macro definition:

```
*4M%FLOOP$=CEND LOOP$V$%$$
```

This defines macro number 4. When the macro is executed (by typing 'N4'), YALOE searches for the string 'LOOP', changes it to 'END LOOP', and displays the altered line.

NOTE- Up to 10 macros (0 through 9) can exist at any one time.

N (execute macro)

Syntax:

```
nNm$
```

Execute the specified macro. 'm' identifies the macro to execute (0 to 9). The default macro number is 1. Because m actually represents a text string of commands, the N command must be terminated by <esc> (echoed as $).

If you try to execute an undefined macro, this message appears:

```
Unhappy macnum.
```

If an error occurs during macro execution, this message appears:

```
Error in macro.
```

? (help)

Syntax:

```
?
```

Display all YALOE commands, current size of the text and save buffers, currently defined macro numbers, and memory left in the text buffer.

9.8 Command Summary

n = integer argument   m = macro number

?	display command list and file information.
nA	advance the cursor to the beginning of the n'th line from the current position.
B	jump to beginning of file.
nC	change by deleting n characters and inserting the following text. Terminate text with <esc>.
nD	delete n characters.
E	erase the screen.
nF	find the n'th occurrence from the current cursor.
nG	position of the following string. Terminate target string with <esc>.
I	insert the following text. Terminate text with <esc>.
nJ	jump cursor n characters.
nK	delete n lines of text from the current cursor position.
nL	list n lines of text.
mM	define macro number m.
nNm	perform macro m, n times.
Q	quit this session, followed by: U:(pdate Update SYSTEM.WRK.TEXT; E:(scape Escape from session; R:(eturn Return to editor
R	read file into buffer starting at cursor; form is: R<file name><esc>.
nS	put the next n lines of text from the cursor position into the save buffer.
U	insert (Unsave) the contents of the save buffer into the text at the cursor; does not destroy the save buffer.
V	verify: display the current line.
W	write file (from start of buffer); form is: W<file name><esc>.
nX	delete n lines of text, and insert the following text; terminate with <esc>.

10 Utility Programs

This chapter describes the utility programs provided with the Modula operating system. Utility programs assist the following tasks:

- Disk management
- File management
- Program management
- Communication
- System configuration

Section 10.1 describes the disk management utilities.

Section 10.2 describes the file management utilities.

Section 10.3 describes the program management utilities.

Section 10.4 describes the communication utilities.

Section 10.5 describes the system configuration utilities.

10.1 Disk Management

This section describes the disk management utilities:

- Bootstrap copier (10.1.1)
- Disk copier (10.1.2)
- Duplicate directory copier (10.1.3)
- Disk directory flipper (10.1.4)

10.1.1 Bootstrap Copier

The Booter utility (BOOTER.CODE on the disk) copies the system bootstrap code from one system boot disk to another. Bootstrap code is assumed to reside on logical disk blocks 0 and 1.

After you X(ecute BOOTER, the following prompt appears:

```
Copy Boot From #4: to #5:  ?
<cr> to Copy, <esc> <cr> Exits
```

The disk already containing the bootstrap code must be placed in disk unit 4. The disk that needs the bootstrap is placed in disk unit 5. To exit Booter, type <esc> followed by <return>. To proceed with bootstrap copying, type <return>.

10.1.2 Disk Copier

The Backup utility (BACKUP.CODE on the disk) copies the contents of one disk (called the **master disk**) onto a second disk (called the **backup disk**). For disk copying, Backup has the following advantages over the filer's T(ransfer command:

- It performs double read and read-after-write checking to ensure that the backup disk is an exact copy of the master disk.
- It copies across any bootstrap code stored in blocks 0 and 1.

After you X(ecute BACKUP, the following prompt appears:

Master in #4: Backup in #5: ?

Type 'y' to specify disk unit 4 as the master disk. If you want to transfer from unit 5 to unit 4, type 'n' and the prompt reappears as:

Master in #5: Backup in #4: ?

Once you have chosen which way to transfer, Backup verifies your choice by printing the volume name of the master disk:

Master on #5: Volume MYDISK:

If the backup disk contains an existing volume, Backup reminds you that it will be destroyed:

Destroy #4: Volume OLDDISK: ?

If you have cold feet, type 'n' to halt Backup; otherwise, type 'y' to continue. If the master disk contains a disk volume, Backup tells you how many blocks are going to be copied — note that it copies only the files on the disk volume and not any of the unused disk space past the end of the last file. (This saves a lot of time if your master disk volume only contains a few files.) If the master disk does not contain a disk volume, then Backup asks you how many blocks to transfer.

Backup then proceeds to copy the master disk; it writes dots to the screen to indicate its progress. When copying is completed, the following prompt appears:

May I rename MYDISK: to BACKUP: ?

What this prompt means is that the master and backup disks have the same volume name at this point, so Backup will change the volume name on the backup disk to the volume name BACKUP:. Type 'y' to change the name to 'BACKUP:'.

Before Backup terminates, one last prompt appears:

E(xit to Boot Diskette ?

Place the boot disk back in its drive, type 'y', and Backup is finished. However, if you want to copy another disk, type 'n' and the original Backup prompt reappears:

Master in #4: Backup in #5: ?

10.1.3 Duplicate Directory Copier

The CopyDupDir utility (COPYDUPDIR.CODE on the disk) copies the duplicate disk directory onto the main directory; it is used to attempt the recovery of disk volumes whose main directory has been ruined. (Note that CopyDupDir is of no help if the duplicate directory itself has also been obliterated.)

After you X(ecute COPYDUPDIR, the following prompt appears:

Drive number with victim disk already in it:

Put the wounded disk volume in a drive and type the disk unit number. CopyDupDir then reads the duplicate directory and displays the volume name stored in it. One last prompt appears before the duplicate directory is actually copied:

May I write over original directory {y/n} ?

Type 'y' to copy the directory. If you get cold feet, type 'n' and CopyDupDir terminates without copying.

10.1.4 Disk Directory Flipper

The FlipDir utility (FLIPDIR.CODE on the disk) byte-flips the word quantities in a disk directory so the disk volume can be read on machines with the opposite byte sex. (The 68000 and 9900 are the opposite byte sex from all other popular microprocessors.)

After you X(ecute FLIPDIR, the following prompt appears:

Unit number to flip?

Put the disk volume in a drive and type the disk unit number. FlipDir then reads the directory into memory, flips the appropriate words, and writes it back to the disk. It also indicates the number of files on the volume. When FlipDir is finished, the following prompt appears:

Directory Flipped, <Return> to Exit

10.2 File Management

This section describes the file management utilities:

- Disk file editor (10.2.1)
- File copier (10.2.2)
- Text file compare (10.2.3)
- Binary file compare (10.2.4)

10.2.1 Disk File Editor

The Patch utility (PATCH.CODE on the disk) is used to examine and alter data stored in a disk file.

Disk files consist of a series of 512-byte blocks; the first block in a file is block 0. Patch lets you read individual blocks from the file and display them in either pure hex or mixed hex and ASCII format. To alter disk file data, move the cursor to the desired location in the block display, type in the new data values, and write the block back to the file.

After you X(ecute PATCH, the following prompt appears:

F(ile, Q(uit

Type Q(uit to exit Patch. If you type F(ile, the following prompt appears:

Filename: <cr for unit i/o>

Type in the name of the file you want to edit. If you want to edit the entire disk volume, type <return>, and the following prompt appears:

Unitnum [4,5,9..12]:

Type in the unit number containing the disk. Patch treats disks like they are big files — the first block on the disk is block 0.

Once you have specified a file to edit, the original Patch promptline reappears containing a new command:

G(et, F(ile, Q(uit

Typing G(et produces the following prompt:

BLOCK:

Type in the block number of a block you wish to examine; Patch then reads the specified block into memory. The promptline reappears this time with two new commands:

G(et, P(ut, H(ex, M(ixed, F(ile, Q(uit

H(ex displays the block in hex characters. M(ixed displays the block in ASCII characters wherever possible; all non ASCII character values appear in hex.

H(ex and M(ixed produce the following prompt at the top of the block display:

Alter: Arrows; L,R,U,Z; 0..F hex chars; S(tuff, Q(uit

The cursor appears at the first byte in the block; it can be moved around either with the vector keys or the letters L,R,U, and Z. (Z means 'down' — D cannot be used because it denotes a hex digit.) Once the cursor has been positioned over the data you want to modify, type the new values in as hex digits. (This works just like the ASE's eX(change command.)

The S(tuff command is used to assign the same value to a number of adjacent bytes. After you type S(tuff, the following prompt appears:

Stuff for how many bytes:

Type in the number of bytes to be modified (starting from the current cursor position). The next prompt appears:

Fill with what hex pair:

Type in the two hex characters which make up the desired byte value. S(tuff then assigns the new value to the specified byte range.

The P(ut command is used to write the altered disk block back to disk. P(ut writes the current block back to the file block it was read from.

10.2.2 File Copier

The FileCopy utility (FCOPY.CODE on the disk) works just like Backup, but it is used to copy a single disk file from one disk volume to another.

After you X(ecute FCOPY, the following prompt appears:

Source FileName?

Put in the disk volumes you want to transfer the file between, then type in the name of the file to transfer. The next prompt is:

Dest FileName?

Type in the name of the destination file, and FileCopy starts transferring the disk file. It writes out the source and destination file names and then displays a series of dots while transferring the files. When FileCopy is finished, it displays the number of blocks transferred.

Before FileCopy terminates, one last prompt appears:

E(xit to Boot Diskette ?

Place the boot disk back in its drive, type 'y', and FileCopy is finished. However, if you want to copy another file, type 'n' and the original FileCopy prompt reappears:

Source FileName?

10.2.3 Text File Compare

The Compare utility (COMPARE.CODE on the disk) compares two text files and reports any differences.

After you X(ecute COMPARE, the following prompt appears:

compare file:

Type in the name of the first text file. (Note that the '.TEXT' suffix is automatically appended.) The next prompt appears:

with file:

After you type in the second file name, the next prompt appears:

WHERE DO YOU WANT TO OUTPUT [<esc-ret>, <filename>] printer: ?

The file name you type here is where the differences description is written to. After the file name is entered, another prompt appears:

**MATCH CRITERION = 3 LINES.
DO YOU WANT TO CHANGE IT (Y/N) N ?**

If you type 'y', Compare prompts for a new match criterion value. The match criterion is defined as the number of lines of text which must match in order to terminate a prior mismatch. Larger values tend to produce fewer — but larger — mismatches than small values; values should be chosen according to how much the two files differ. The default value of 3 yields good results when comparing Pascal program text.

The last prompt is:

IGNORE INDENTATION (Y/N) Y ?

If you type 'y', lines containing the same text but with differing indentation are considered identical.

Once past all these prompts, Compare proceeds to compare the two text files. It displays a dot on the screen for every line compared. (Every tenth dot is displayed as a '+' to improve the display format.)

When Compare finishes comparing the files, it writes out a report describing differences between the two files. If there are no differences, it writes 'no differences'; otherwise, it lists each mismatch including the line numbers in both files, and (if the difference is restricted to one line) both text lines with pointers to the beginning of the difference.

When Compare finishes writing the difference report, it displays the following prompt:

R(epeat or E(nd:

To terminate Compare, type 'e'. However, if you want to compare another pair of files, type 'r' and the original prompt reappears:

compare file:

NOTE- Compare stores its report data on the heap, so files that generate very large difference reports may cause the system to stack overflow.

10.2.4 Binary File Compare

The CompCode utility (COMPCODE.CODE on the disk) compares two disk files and reports any differences. Unlike Compare, CompCode performs a binary comparison of all data in the two files.

After you X(ecute COMPCODE, the following prompt appears:

Name of file1 ?

Type in the name of the first file. (Note that you must type in the file suffix.) The next prompt appears:

Name of file2 ?

After you type in the second file name, CompCode starts comparing the two files. No message is displayed if the two files match exactly. If CompCode discovers a difference, it displays the message:

Error at blk 2, offset 129

... indicating the block number and byte offset where the two files differ.

10.3 Program Management

This section describes the program management utilities:

- Librarian (10.3.1)

10.3.1 Librarian

The Librarian utility (LIBRARY.CODE on the disk) is used to manipulate code segments within a program's code file.

After you X(ecute LIBRARY, the following prompt appears:

Output Code FileName:

Type in the name of the output code. (Note that a file suffix is not automatically appended — you must type it yourself.) The input file prompt appears next:

Input Code FileName:

Type in the name of the input code file. (Note that the file suffix .CODE is automatically appended.) Librarian then lists the name, segment number, and size in bytes of each segment in the code file.

After the code segments are displayed, a promptline appears:

P(rompt, #_of_seg, N(ew {<bl> or <cr>}, Q(uit, or A(bort:

The P(rompt command steps through all segments in the input file, asking if you want to copy the segment to the output file:

Link 1 PASCALCO {y/n} ?

Type 'y' to copy the segment across; type 'n' to skip the segment.

The '#_of_seg' command is used to copy individual segments to the output file. Type in the segment number of the segment you wish to copy and the following prompt appears:

Source segnum: 8

Notice that the segment number you typed is already entered; you can change it by backspacing and typing in another segment number. After you type <return>, the next prompt is:

Target segnum:

Type in the segment slot where the segment is to reside in the output file. The segment is then copied into the output file display.

The N(ew command selects a new code file as the input file. N(ew lets you copy segments from a number of code files into one output file. The following prompt appears:

New input file:

Type in the name of the new input file. The input file display is then updated with the segments in the new file.

The Q(uit command displays this prompt below the promptline:

Notice:

Type in a copyright notice. (If you don't care about having a copyright notice, just type <return>.) Librarian stores the copyright notice in the segment dictionary (block 0 of the code file).

The A(bort command exits Librarian without saving the output file.

10.4 Communication

This section describes the communication utilities:

- Remote file transfer (10.4.1)
- Electronic mail transfer (10.4.2)

10.4.1 Remote File Transfer

The SerialTalk utility (SERIALTALK.CODE on the disk) is used to transfer files from one machine to another via a serial communication line (RS232). Both machines must have their serial ports online, set at the same baud rate, and connected to each other by the appropriate cable. Both machines must also have the SerialTalk program running.

To transmit data files, SerialTalk is executed on both machines. One machine is specified as the 'slave' machine; the other becomes the 'master'. File transfer commands are entered on the master machine, which then sends the appropriate commands to the slave machine.

After you X(ecute SERIALTALK, the following prompt appears:

1) Let both machines reach this prompt
2) Press S(lave on one machine
3) Press M(aster on the other

M(aster S(lave Q(uit

The S(lave command does not display a prompt; it merely places the machine in the hands of the master machine. Any further information displayed on the screen is a result of commands from the master.

The M(aster command displays the following prompt:

S(end R(eceive C(onfigure F(ile Q(uit

The S(end and R(eceive commands prompt for the name of a file to send or receive. Once a file name has been typed in, it is displayed on both the master and slave machines. While transmitting file data, SerialTalk displays a dot for each block transferred. If a parity error is detected, a question mark is displayed and the erroneous block is retransmitted.

The C(onfigure command changes the modes used by SerialTalk to transmit data. C(onfigure displays the following prompt:

P(acket size D(ata size Q(uit

The P(acket size command controls the number of bytes sent between the machines at one time. P(acket size displays the following prompt:

A)512 B)256 C)128 D)64 Q(uit

The default packet size is 512 bytes. Smaller packet sizes are used when one of the machines isn't fast enough to keep up with the specified baud rate: this in turn is often caused by using a packet size larger than the machines' serial port data buffers.

The D(ata size command controls the mode in which individual characters are transmitted. D(ata size displays the following prompt:

P(arity R(aw Q(uit

The default mode is P(arity. In Parity mode, SerialTalk translates all data into ASCII alphabetic character codes before transmitting it. This prevents control character values from being interpreted as commands by the master or slave machines' serial port I/O drivers. (This problem occurs most often when sending data between foreign operating systems.) R(aw mode transmits all data without converting it to alphabetic character codes. It can be used only when the serial port drivers on both machines transmit all 8 bits in a data byte and do not perform any special handling of control characters. Note that R(aw mode transmits about three times faster than P(arity mode.

The F(ile command prompts for the name of a command file. Command files are used to automate the transmission of a number of files. A command file should contain exactly the characters that would have been typed in to transmit the files manually. (Note in particular that they are 'pure' text files and not the command files described in chapter 6.)

The Q(uit command terminates SerialTalk.

10.4.2 Electronic Mail Transfer

The TeleTalk utility (TELETALK.CODE on the disk) is used for sending and recording text files during electronic mail sessions. Sending text files lets you compose your messages with the editor beforehand, then send them at high speed to reduce your connect time. Recording mail sessions lets you save your incoming messages in text files so you can read them afterwards, again saving you connect time.

To start an electronic mail session, X(ecute TELETALK. The following prompt appears:

BaudRate 1(200, 3(00 ?

Select the appropriate baud rate for your modem by typing '1' or '3'. TeleTalk then displays the follwing message:

<Ctrl-A> for option menu

This indicates that typing ctrl-A at any time during an electronic mail session halts the session and displays the TeleTalk prompt.

TeleTalk is then ready for you to sign on to the network. All subsequently typed characters are written to the network connection along with being echoed to the screen. Characters received from the network are displayed on the screen.

NOTE- The message "LOST CARRIER" appears if you haven't dialed into the network yet. If this happens, dial into the network and type <space> or <return> a few times to establish the connection.

Whenever you reach a point where you want to send or record a text file, type ctrl-A. The current session is suspended and the following prompt appears:

FileOptions: S(end, R(ecord, G(o, E(xit -

The G(o command resumes the session. E(xit terminates the TeleTalk program and returns you to the system prompt. Note that you can terminate TeleTalk, run other programs in the system (such as the filer), then return to TeleTalk without interrupting a mail session: reexecuting TeleTalk pops you back into the current session.

To send a text file, type S(end. The following prompt appears:

Send what textfile?

Type in the name of the text file you want to send (don't add ".TEXT"). Type G(o and TeleTalk returns to the session, transmitting the file while echoing it to the screen. The following message appears on the screen when TeleTalk is finished sending the file:

MYFILE.TEXT Finished

If you want to stop sending a file before it reaches the end, type ctrl-A to get the prompt and type S(end again. The following prompt appears:

Currently Sending MYFILE.TEXT C(lose it?

C(lose stops sending the file. Typing anything else causes TeleTalk to resume sending the file.

To record a session in a text file, type R(ecord. The following prompt appears:

Record as what textfile?

Type in the name of the text file you want to record (don't add ".TEXT"). Type G(o and TeleTalk returns to the session, with all characters subsequently written to the screen being recorded in the text file.

To finish recording a file, type ctrl-A to get the prompt and type R(ecord again. This time the following prompt appears:

Currently Recording MYFILE.TEXT C(lose P(urge

C(lose saves the recorded file on disk. P(urge removes the file.

WARNING- Be sure there's enough space on your disk before you start recording. If TeleTalk runs out of disk space while recording, it discards all subsequent text without issuing a warning.

10.5 System Configuration

This section describes the utility programs Setup and Binder. These utilities are used to configure the system software to operate with different terminals.

The Setup utility (10.5.1) is used to create and modify the system information file SYSTEM.MISCINFO.

The MISCINFO file is always stored on the boot disk. It contains three types of system-dependent information:

- Miscellaneous system data
- Terminal-dependent control characters
- Key definitions for ASE edit commands

When the system bootstraps, it reads the contents of MISCINFO into an operating system data structure known as SYSCOM. The system programs access SYSCOM to obtain system-dependent information.

The Binder utility (10.5.2) binds a new Gotoxy procedure into the operating system code file. The operating system procedure Gotoxy is used to move the cursor to arbitrary positions on the screen. Because most terminals use different character sequences for cursor positioning, a different Gotoxy procedure is usually needed for each terminal.

10.5.1 Terminal Setup

The utility program Setup (SETUP.CODE on the utilities disk) is used to create a new MISCINFO file or to modify an existing one.

After you X(ecute SETUP, the MISCINFO fields are displayed on the screen with their current values. (If Setup cannot find the file SYSTEM.MISCINFO, it displays the values already loaded into memory in SYSCOM.)

Each MISCINFO field is displayed in the following format:

```
Crt home              0 ["H"   72.]
```

The field name is followed by a menu number; this is used in the S(ingle command to select individual fields for modification. The menu number in the example above is 0.

The values enclosed in the brackets indicate the current field value. Fields contain either character values or Boolean values. Boolean values are displayed either as 'True' or 'False'. Character values are displayed in both symbolic and numeric form. If a field value denotes a printable character, the character is displayed; otherwise, the ASCII control character symbol is displayed (e.g. ACK, LF).

The numeric value may be displayed in one of three radices: decimal, hex, or octal. Decimal numbers end with a period ('72.'). Hex numbers end with 'h' ('BEh'). Octal numbers end with 'o' ('26o').

Some MISCINFO fields may be designated as 'prefixed'. This means that the field value represents the second character of a two-character sequence. The first character is known as the **prefix character.** The prefix character is defined by two MISCINFO fields: 'Crt prefix character' and 'Keyboard prefix char'. Fields designated as prefixed are displayed with the letter 'p' on the right-hand side of the field value:

```
Crt right                6 ["J"  4Ah] p
```

The following promptline appears below the field display:

```
P(rompt, S(ingle, N(ew, R(adix, M(em, D(isk, <Escape> ?
```

The S(ingle command produces the following prompt:

```
Current default Radix: Decimal
Single change: menu #, <return> to accept, <esc> to Escape?
```

If you type one of the menu numbers, Setup displays the corresponding field and waits for you to type in the new value. If the field is Boolean, type 't' for True or 'f' for False. If the field expects a character value, you can either type the character directly or you can specify the numeric value of the key by typing '#' followed by the numeric value. (Note that numerical values assume the current default radix.) Numeric values are terminated by typing <return>.

WARNING- Setup interprets entered menu numbers in terms of the current radix.

For every character field, Setup also prompts for whether the field is prefixed. Type 'y' to mark a field as prefixed; type 'n' to mark it as unprefixed. Setup defines one input ('key') prefix character and one output ('char') prefix character; fields marked as prefixed denote 2-character sequences beginning with the appropriate prefix character. Prefix characters are defined as separate fields in Setup.

The 'Single change' prompt reappears after you have set each field. To change another field, type its menu number. Note that after setting a series of fields in S(ingle, the new field values are not actually established. To install the new field values in the display, type <return>; this terminates the S(ingle command and redisplays the fields with the new values. To escape from S(ingle, type <esc>; this terminates S(ingle and redisplays the fields without updating them.

The P(rompt command steps through every field asking for a new field value. If you do not want to change a field, type <return> and Setup will skip to the next field.

The N(ew command works the same way as P(rompt, but specifies 'empty' field values as the default. N(ew is used for constructing new MISCINFO files from scratch.

The R(adix command changes the default radix. The following prompt appears:

Current default Radix: Decimal
Default Radix: O(ctal, D(ecimal, H(ex, <return>

Type the appropriate letter to set the new default radix. Type <return> to preserve the current default radix. The default radix affects both the field display and how numeric responses are interpreted.

The D(isk command writes the current field values to the file NEW.MISCINFO. This file must be changed to SYSTEM.MISCINFO in order to be used by the system.

The M(em command writes the current field values to the SYSCOM data structure in memory. This lets you test the new field values immediately after leaving Setup; however, they will be lost if the system is rebooted (or I(nitialized) before you can go back into Setup and invoke the D(isk command.

The <Escape> command is the only way to terminate Setup. It produces the following prompt:

Are you sure you want to Exit?

Type 'y' to exit Setup; type 'n' to return to Setup. Note that you must invoke either the D(isk or M(em command to preserve the results of a Setup session; <Escape> does not perform any automatic saving of the current field settings.

MISCINFO Fields

This section describes the MISCINFO fields in detail. The fields contain three types of system information: **keys, screen info,** and **parameters.**

Key fields define how character sequences received from the terminal are interpreted as system commands. Key fields in Setup have the word 'Key' in their field names.

Screen info fields define how the system screen-control commands are mapped into character sequences written to the terminal. Screen info fields in Setup have the word 'Crt' in their field names.

Parameter fields contain various integer, character, and Boolean values which control system operation.

Key accept

Used as the editor <etx> command. Standard value: ASCII ETX

Key escape

Used as the <esc> command in the editor and other programs. Suggested value: ASCII ESC

Crt clear lin

When written to the console, this character erases everything on the line that the cursor is on, leaving the cursor at the line start.

Crt clear

When written to the console, this character erases the entire screen, leaving the cursor at the top left of the screen.

Crt erase eol

When written to the console, this character erases all characters from the current cursor position to the end of the line, leaving the cursor at its current position.

Crt erase eos

When written to the console, this character erases all characters from the current cursor position to the end of the screen, leaving the cursor at its current position.

System is Terak 8510a

Set to FALSE (unless you are using a Terak).

System has clock

Set to TRUE if the VS Pascal intrinsic TIME is implemented.

Crt has u/l case

Set to TRUE if the terminal supports both upper and lower case characters.

Crt has x,y control

Set to TRUE.

Crt is slow

Set to FALSE if terminal runs faster than 600 baud.

Key flush

When typed, this key cancels all console output. See 2.3 for details. Suggested value: ctrl-F

Key stop crt

When typed, this key suspends console output. See 2.3 for details. Suggested value: ctrl-S

Key break

When typed, this key causes execution error 8 ("user break").

Key del char

When typed, this key deletes the character under the cursor and moves the cursor one space to the left. Suggested value: ASCII BS

Key del line

When typed, this key deletes the line under the cursor. Suggested value: ASCII DEL

Key end file

When typed, this key sets EOF to true while reading from a console input file. Suggested value: ASCII ETX

Key up
Key down
Key right
Key left

Used for cursor movement. These should be mapped to the terminal's arrow keys.

Keyboard mask

Used to mask off high order bits of characters received from the keyboard. Standard value: 127

Editor bad ch

The editor displays this character whenever a non-printing character is written to the console. Standard value: ASCII '?'

Keyboard prefix char

Prefix character for all prefixed key fields.

Crt prefix char

Prefix character for all prefixed screen info fields.

Crt home

When written to the console, this character moves the cursor to the upper left hand corner of the screen.

Crt right

When written to the console, this character moves the cursor one space to the right without erasing any characters.

Crt backspace

When written to the console, this character moves the cursor one space to the left. Suggested value: ASCII BS

Crt rev lf

When written to the console, this character moves the cursor vertically up one line without erasing any characters. Not used by the system.

Crt height (rows)

The number of text lines displayed on the console. Standard value: 24

Crt width (columns)

The number of characters per line displayed on the console. Standard value: 80

Nulls for move delay

Used to implement vertical move delays on slower terminals. The system will write the specified number of bogus nulls after each cursor move. Values greater than 11 are ignored. Standard value: 0

Byte sex is 9900/68000

Set to TRUE if running on 9900 or 68000 processor.

Word-addressed (FLASH)

Set to TRUE if running on word-addressed machine.

Student user

Set to FALSE.

10.5.2 GOTOXY Procedure Binding

The operating system is shipped with a terminal-independent version of the cursor-moving procedure Gotoxy. While this version is portable, it is also rather slow and especially irritating to watch as the cursor jumps all over the screen. To improve system performance and reduce eye strain, you should create a Gotoxy procedure for your terminal and bind it into the operating system code file with the Binder utility.

First, a Pascal program containing the Gotoxy procedure must be written and compiled to a code file. A number of sample Gotoxy programs are provided with the system; if none of them is suitable for your terminal, modify whichever one is closest enough so it generates the character sequence needed by your terminal.

The Binder utility (BINDER.CODE on the disk) is used to bind a compiled Gotoxy procedure into the operating system code file.

After you X(ecute BINDER, the following prompt appears:

CodeFile name of new GoToXY:

Type in the code file name of the Gotoxy program. The operating system code file must be on the prefixed volume. Binder reads the operating system code into memory, binds in the new procedure, and writes the modified system code back to the disk with the name SYSTEM.PASCAL. Rebooting the system loads the new operating system.

NOTE - Binder removes the old operating system code file, so be sure you have a backup copy of it laying around somewhere in case something goes wrong.

Appendix 1 Installation Guide

This section provides an overview of the basic steps necessary to install the Modula operating system on your computer. Details and additional information are provided in a separate document.

The Modula operating system is shipped on one or more floppy disks. One of the disks is usually labelled 'SYSTEM' or 'BOOT' — this is the boot disk. To start the system, place the boot disk in your computer's system disk drive (your computer's owners manual should indicate which drive this is). Next, press the 'reset' button (or whatever it's called on your system).

After a few disk accesses, a welcome message and the system promptline should appear on the screen. (If not, reread the instructions provided with the system and try again; if it still doesn't boot, ask your software dealer for help.)

Your Modula system has booted for the first time. Congratulations! If your computer system includes a standard monitor (like the IBM PC or Osborne), your system disks have already been configured to work on your computer, so you can start using the system immediately. However, if the system you purchased is designed to work with an arbitrary terminal (like the Sage), you will have to run some configuration programs before you can make full use of the system. The rest of this section explains how to configure your system.

The ASE text editor makes extensive use of your terminal's screen control capabilities to do things like moving the cursor around the screen and erasing the screen after some commands. On an unconfigured system, the editor does not work very well (if at all) because these screen control functions have not been defined. So the first thing to do when you first boot the system is to run the utility program named Setup. To run Setup, you have to know the special characters used by your terminal to control the screen — this information can be found in your terminal's owner manual. Section 10.5.1 of this manual explains how to use Setup.

Once you've successfully run Setup, you can make full use of the system. However, you might notice a lot of transient cursor jumping-arounding while you're in the editor. This can be eliminated by running the utility program named Binder. Binder adds a new cursor-moving procedure to the operating system, eliminating the screen flicker when the cursor is moved (and also noticeably improving the screen response). Section 10.5.2 of this manual explains how to use Binder.

Once you've successfully run Setup and Binder, your system is fully configured and ready for full-time use.

Appendix 2 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 on disk or directory is full
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 3 Execution Errors

0	System 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	Halt, breakpoint
15	Bad block

Appendix 4 Compiler Error Messages

1:	Error in simple type
2:	Identifier expected
3:	'PROGRAM' expected
4:	')' expected
5:	':' expected
6:	Illegal symbol (maybe missing ';' on the line above)
7:	Error in parameter list
8:	'OF' expected
9:	'(' expected
10:	Error in type
11:	'[' expected
12:	']' expected
13:	'END' expected
14:	';' expected
15:	Integer expected
16:	'=' expected
17:	'BEGIN' expected
18:	Error in declaration part
19:	Error in <field-list>
20:	'.' expected
21:	'*' expected

50:	Error in constant
51:	':=' expected
52:	'THEN' expected
53:	'UNTIL' expected
54:	'DO' expected
55:	'TO' or 'DOWNTO' expected in for statement
56:	'IF' expected
57:	'FILE' expected
58:	Error in <factor> (bad expression)
59:	Error in variable

101:	Identifier declared twice
102:	Low bound exceeds high bound
103:	Identifier is not of the appropriate class
104:	Undeclared identifier
105:	Sign not allowed
106:	Number expected
107:	Incompatible subrange types
108:	File not allowed here
109:	Type must not be real
110:	<tagfield> type must be scalar or subrange
111:	Incompatible with <tagfield> part
112:	Index type must not be real
113:	Index type must be a scalar or a subrange

114:	Base type must not be real
115:	Base type must be a scalar or a subrange
116:	Error in type of standard procedure parameter
117:	Unsatisfied forward reference
118:	Forward reference type identifier in variable declaration
119:	Re-specified params not OK for a forward declared procedure
120:	Function result type must be scalar, subrange or pointer
121:	File value parameter not allowed
122:	Forward declared function result type can't be re-specified
123:	Missing result type in function declaration
124:	F-format for reals only
125:	Error in type of standard function parameter
126:	Number of parameters does not agree with declaration
127:	Illegal parameter substitution
128:	Result type does not agree with declaration
129:	Type conflict of operands
130:	Expression is not of set type
131:	Tests on equality allowed only
132:	Strict inclusion not allowed
133:	File comparison not allowed
134:	Illegal type of operand(s)
135:	Type of operand must be boolean
136:	Set element type must be scalar or subrange
137:	Set element types must be compatible
138:	Type of variable is not array
139:	Index type is not compatible with the declaration
140:	Type of variable is not record
141:	Type of variable must be file or pointer
142:	Illegal parameter substitution
143:	Illegal type of loop control variable
144:	Illegal type of expression
145:	Type conflict
146:	Assignment of files not allowed
147:	Label type incompatible with selecting expression
148:	Subrange bounds must be scalar
149:	Index type must be integer
150:	Assignment to standard function is not allowed
151:	Assignment to formal function is not allowed
152:	No such field in this record
153:	Type error in read
154:	Actual parameter must be a variable
155:	Control variable cannot be formal or non-local
156:	Multidefined case label
157:	Too many cases in case statement
158:	No such variant in this record
159:	Real or string tagfields not allowed
160:	Previous declaration was not forward
161:	Again forward declared
162:	Parameter size must be constant
163:	Missing variant in declaration
164:	Substitution of standard proc/func not allowed

165:	Multidefined label
166:	Multideclared label
167:	Undeclared label
168:	Undefined label
169:	Error in base set
170:	Value parameter expected
171:	Standard file was re-declared
172:	Undeclared external file
174:	Pascal function or procedure expected
193:	Not enough room for this operation
194:	Comment must appear at top of program
201:	Error in real number - digit expected
202:	String constant must not exceed source line
203:	Integer constant exceeds range
204:	8 or 9 in octal number
250:	Too many scopes of nested identifiers
251:	Too many nested procedures or functions
252:	Too many forward references of procedure entries
253:	Procedure too long
254:	Too many long constants in this procedure
256:	Too many external references
257:	Too many externals
258:	Too many local files
259:	Expression too complicated
300:	Division by zero
301:	No case provided for this value
302:	Index expression out of bounds
303:	Value to be assigned is out of bounds
304:	Element expression out of range
398:	Implementation restriction
399:	Implementation restriction
400:	Illegal character in text
401:	Unexpected end of input
402:	Error in writing code file, not enough room
403:	Error in reading include file
404:	Error in writing list file, not enough room
405:	Call not allowed in separate procedure
406:	Include file not legal

Appendix 5 ASCII Character Set

0	000	00	nul	32	040	20		64	100	40	@	96	140	60	`
1	001	01	soh	33	041	21	!	65	101	41	A	97	141	61	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	$	68	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	065	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	037	1F	us	63	077	3F	?	95	137	5F	_	127	177	7F	del

Index