Implementation Guide

TABLE OF CONTENTS

IMPLEMENTATION GUIDE

Catchwords of this part are:

- Program Description and Operation
- How to Compile and Link
- System Restrictions
- Standard Library Description
- Utility Library Description

The **Introduction to Modula-2** explained the Modula-2 language by comparing it to its predecessor, Pascal. Here, you will learn to operate the programs that pertain to the Modula-2 System for Z80 CP/M. Also covered in this section are the descriptions of the library modules that are delivered with this implementation.

There are several Modula-2 conventions that are at best strange for CP/M users. For instance, a ReadString operation terminates not only upon receipt of a CR character denoting a line's end, but also upon receipt of a space. So, please read the specifications of each and every library module carefully before using it.

Chapter 1. Program Operation and Usage

In this section, the programs of the Modula-2 System and their particular usage will be explained.

Section 1. File Types used by the System

The files used by the System can be categorized after their type. This is done in the following table.

1. Table Of File Types

The Modula-2 System uses the following file types:

Type	Usage
MOD	Files of type **MOD** are used to store the source text of Program Modules or IMPLEMENTATION MODULEs. They are compiler input files.
DEF	DEF-typed files serve as DEFINITION MODULE source holders. Like .MOD files, they are compiler input files.
MSY	From each successful compilation of a DEFINITION MODULE, a so called **symbol file** results. These are of type **MSY.** These files are generated by the compiler and are also compiler input files.

Type	Usage
MRL	Upon successful compilation of a MOD file, the compiler generates an **object file** or **relocatible file.** These files are of type **MRL.** They are also linker input files. Why MRL, not REL: Although there exist already a whole bunch of more or less compatible derivations from the standard Microsoft REL-Format, we decided to create a new, not directly compatible variant. This was necessary to have longer names than 8 characters, among other reasons. For a more detailed discussion, please have a look at the appendix **The MRL Format.**
MAC	If you selected the assembler output commandline switch of the compiler, a Microsoft M80 compatible **MAC** file is created instead of the usual MRL file. NOTE - You cannot simply create all MAC files, assemble them and link them using L80, because the Modula-2 linker does some work that cannot be done by L80. Please read the assembly language interface section in the **Advanced Programming Guide** for more information concerning this topic.
MAP	Upon request, the linker creates a file of the addresses of all symbols that are known to him. "Known" are only symbols that are imported by some module. These files are of type **MAP**; they are not compatible with SYM files.
LST	The compiler lister generates a file of type **LST** upon unsuccessful compilations (i.e. if the compiler found errors in your source program). The compiler's error messages are interspersed with your source code.
REF	The reference lister generates files of type **REF.**
COM	The linker generates standard executable command files of type **COM.**

Type	Usage
R2M	Files of this type are supposed to contain name translation tables for assembly language modules. For further information about assembly language integration, consult the **Advanced Programming Guide.**
$$$	This file type is used by the compiler for its intermediate files. Should there rest any on your disk after an interrupted compilation, you can delete all of them without danger.

2. Modula-2 Program Files

Program files (they are also referenced to as **compiler input files**) can be created and updated with any ASCII text editor (WordStar, Magic Wand, Mince, PolyVue,...). The compiler isn't sensible regarding the "unused" hi bit of characters; you can even edit your programs in WordStar's famous "document mode" without you having to use PIP to get a clean compilation.

The compiler doesn't accept **control characters** in string constants. Control characters are ASCII NUL (0C, 0H) .. ASCII US (37C, 1FH) and ASCII DEL (177C, 7FH)) .

3. Symbol Files

From a DEFINITION MODULE compilation, a so called **symbol file** results. It contains a compressed representation of the information contained in the definition module. It also contains extracts of all modules imported by that definition module.

Section 2. Program Description

All programs of the Modula-2 System for Z80 CP/M will subsequently be described. All options and switches of each program are presented. Therefore, you can use this chapter as a reference manual for program operation. First, let us introduce the general calling conventions for all programs of the System.

1. General Calling Conventions

All programs of the System use the same command line interpreter and therefore, all of them are similar to use, i.e. to invoke. How you can do this is explained in this section.

There are two forms of calls:

```
(1)  A>ProgramName CommandLine <CR>

(2)  A>ProgramName <CR>
      *CommandLine <CR>
```

'ProgramName' denotes the actual name of the program you want to invoke. It is given later in the detailed program descriptions. The form of the 'CommandLine' is for both call forms as follows:

```
CommandLine    = ModuleFileName {"/" Switch} .
ModuleFileName = ModuleName ["." ModuleType] . [1]
ModuleName     = character {character | digit} .
ModuleType     = "DEF" | "MOD" .
Switch         = character [":" (HexNumber | DecNumber | String)] .
String         = character {character} .
DecNumber      = digit {digit} .
HexNumber      = HexDigit [HexDigit [HexDigit [HexDigit]]] .
HexDigit       = digit | "A" | .. | "F" .
digit          = "0" | .. | "9" .
```

[1] The ModuleType addition is not allowed in the linker and the converter.

```
character        = "A" | .. | "Z" | "a" | .. | "z" .
```

When you call a program using form (1), it signs on this way:

```
MODULA-2 <Program Function> for Z80 CP/M   Version dd-mm-yyyy
```

The program immediately starts his activity.

If you call one of the programs by using the second invocation variant, it signs on as follows after you entered the first line:

```
MODULA-2 <Program Function> for Z80 CP/M   Version dd-mm-yyyy
*
```

The star prompt is displayed and the cursor is placed right behind it. Now, you can enter the command line, which is terminated by another hit of the RETURN, ENTER, or NEW LINE key on your terminal.

NOTE - None of the programs is case sensitive in its command line.

NOTE - The command line interpreter is tolerant against spaces that are entered before the program name, i.e. typing

```
A> mc myprog/v <CR>
```

actually compiles what you intended to compile, 'myprog' in verbose mode, and not 'mc'.

You may make a mistake when entering a command line. Errors of this kind are flagged by the following messages:

```
---- Filename Expected
---- Illegal Command Line
---- Illegal Switch /x        ('x' is what you entered)
---- Illegal Use of x-Switch    (        "        )
```

In most cases, they lead to another star-prompt. Just start over and enter the whole line again. If you have specified an unknown file name, all the programs abort with the following message

```
---- FILENAME.TYP Not Found
```

Another error message can occur when your disk is nearly full on program start and during the execution, the disk really fills up. Then, either

```
---- DISK FULL (or Directory Full)
```

or

```
---- Cannot Write to FILENAME.TYP
```

resp.

```
---- Cannot Create FILENAME.TYP
```

appear as error messages before the programs abort.

After this general remarks about program invocation and error messages, you will find detailed descriptions of each program including an 'action summary' describing what's going on during you're waiting for the respective program to finish, together with a detailed description of the switches of each one, as well as some remarks about program input-, output- and intermediate files.

2. MC the Modula-2 Compiler

a) General Description

The compiler's duty is to analyze your program source text, to check it for errors, and finally to create a machine language program that is equivalent to your source program. This whole process is called a compilation of your program.

The Modula-2 Compiler for Z80 CP/M accomplishes this task in four steps or four **passes.**

Pass 1 checks the program syntax. It detects all syntax errors except for misspelled identifiers (remember, Modula-2 distinguishes upper- and lowercase in identifiers). If it finds errors, then, depending on the mode you selected for the compilation, it either writes an 'E' for the first of every ten errors, or it outputs for each error a message of the form

```
     Error eee on line llll, pos. ppp
```

'eee' is a 1 to 3 digit error number. 'llll' indicates the line on which the error occurred, first line of your program being line one, 'ppp' the position, the first position on each line being position one. For each error, a beep sounds. If you selected the verbose output and a serious error is detected, then the compiler stops after Pass 1; in non verbose mode, the lister is called. You can look up the error messages at the very end of manual, just before the index.

Depending on the seriousness of your error(s), Pass 1 either calls the lister or it 'remedies' your error(s) so that Pass 2 can work properly on your program. Among others, missing semicolons and Pascal-style BEGINs are 'corrected'. The correct(ed) version goes to

Pass 2, which checks all declarations for correctness and suitability. This is the place where some of the misspelled identifiers are likely to be detected. The error handling is the same as in Pass 1, but Pass 2 doesn't make any corrections and assumptions whatsoever on its own. If no error is detected and depending on what you compile, it calls either the SymPass (for definition modules) or

Pass 3 which does all the type checking and will flag all of the unknown identifiers resulting from misspellings. The error handling is the same as in the previous passes.

Additionally, Pass 3 tests if any errors occured in Pass 1 (the corrected ones). For safety's sake, it aborts compilation if Passes 2 and 3 found no error, while Pass 1 had corrected some minor ones. If no errors occur,

Pass 4 gets in command of the system. Its task is to generate code. Normally, no error occurs in it; should there be an error, it will be a fatal one which ends the compilation with a message looking like:

```
Expected Token: mm Received Token: nn ****
**** Compiler Error 5xx on Line llll
```

Such errors always flag an internal compiler error. If you get one, please send us a preferably small listing that provokes it. We'll try to fix it as soon as possible.

If you request a listing by a command line option, Pass 4 passes control to the lister after a successful compilation. The output of Pass 4 is called an **object file** or a **relocatible file**, if the assembly language output has been selected, it is called an **assembler file.**

SymPass is called by Pass 2, if you compile a definition module. It generates a **symbol file.**

The last part of the compiler is the **Lister.** Its task is to merge the compiler's error messages with your source program into a listing file that has numbered lines and error messages in it. The error messages look as follows:

```
*****                    ^nnn
```

The 'star bar' is between line numbers; 'nnn' represents the error number that applies to a symbol beginning just above the up arrow ('^'). Should there be any errors that apply to the beginning or end of the file, they are indicated at the beginning or end of the file as follows:

```
/          nnn,nnn,...
```

As in the above form, 'nnn' are (different) error numbers. The **error message explanations** are located at the very end of the manual, just before the index.

b) Technical Description

ProgramName	MC
Input	Files of type **DEF** or **MOD.** Pass 2 needs files of type **MSY** of all imported modules. Default file type is MOD, i.e. if you simply enter a file name without type, the type is assumed to be MOD.
Intermediate	Several files of type **$$$** are used. They are all deleted after a successful compilation. After an interrupted one, you can delete all of these files without danger.
Output	Depending on the type of the compiled module and on the commandline switches, either a **MSY,** a **MRL** or a **MAC** file will be created. If compilation errors occur and you didn't select verbose mode (see later), a **LST** file is generated by the lister.
Switches	/A, /L, /V, /X.

The **A** switch selects assembly language instead of the default MRL format file to be output. The output file bears the **MAC** type in this case. Please read the **Advanced Programming Guide** if you want to use this output not just to see how well the compiler optimizes. (Default: MRL output)

The **L** switch is used to force the compiler to generate a listing also on clean compilations. Please consider the use of MX the reference lister (see later). (Default: No listing)

To make the compiler verbose, set the **V** switch. Several informations about usage of tables internal to the compiler is written to the screen in this case. (Default: non verbose)

The **X** switch directs the compiler to delete the file '$$$.SUB' and thereby interrupts the execution of a submit file if an error occurs. (Default: submit continues)

NOTE - The file name of a given module has to coincide with the first eight characters of the module name (less if the module's name is shorter). To 'coincide' means that all characters are uppercased. No lower case characters are allowed in file names to fulfill the CP/M conventions.

NOTE - All compiler switches may also be embedded in the source code in the usual form (i.e. (*$A+ *), (*$L- *)). This comment may stay anywhere in the source code. If you specify one switch multiple times, only the last setting will be acted upon.

c) Examples

NOTE - In accordance to the typographic conventions, all user input is underlined in the examples.

- The definition module InOut shall be compiled.

```
A>MC INOUT.DEF <CR>
MODULA-2 Compiler for Z80 CP/M    Version 17-03-1985
Compilation of INOUT.DEF
P1.
P2.
SymPass.
A>
```

resulting from this compilation is the file 'INOUT.MSY'.

- Next, the implementation module InOut is to be compiled; the generation of a listing is forced.

```
A>MC INOUT/L <CR>
MODULA-2 Compiler for Z80 CP/M    Version 17-03-1985
Compilation of INOUT.MOD
P1..............
Imported Modules:
    INOUT   .MSY - InOut
    TERMINAL.MSY - Terminal
    SEQIO   .MSY - SeqIO
    CONVERSI.MSY - Conversions
```

```
          STRINGS .MSY - Strings
          MOVES   .MSY - Moves
     P2............
     P3............
     P4............
     Lister
          No Error detected
     A>
```

This compilation generated 'INOUT.MRL' and 'INOUT.LST'.

- Working in verbose mode, the compiler shall compile the program module MAIN.

```
     A>MC MAIN/V <CR>
     MODULA-2 Compiler for Z80 CP/M   Version 17-03-1985
     Compilation of MAIN.MOD
     P1......
     307 bytes and 27 names in name table
     Imported Modules:
        INOUT   .MSY - InOut
     P2......
     485 bytes in name table
     P3......
     15% of expression evaluation buffer used
     P4......
     Code Size: 1307 bytes
     Data Size:   54 bytes
     A>
```

The resulting file is 'MAIN.MRL'.

3. ML the Modula-2 Linker

a) General Description

The linker's duty is to join different relocatible files to form an executable program. The Modula-2 Linker is implemented as a two pass linker (no, we do not have an acute multipassitis!).

Pass 1 searches all parts of the program (including all library modules that are used by it) and collects the information that

Pass 2 needs to generate an executable program.

This split into two passes was done for the following reasons:

- You can link programs as big as 64 kBytes.

- The symbol table space got bigger.

- The linker automatically finds all modules necessary to construct the program in its first pass. This makes it amazingly simple to operate.

- All modules that are used by your program-to-be-linked have to be initialized before the program body (the main program) executes. If one of your modules relies on a second one, it is intuitively clear that the second one has to be initialized before the first one. This is also assured by the linker.

The runtime library **MODLIB.MRL** is necessary to get the linker doing its task. The compiled code heavily relies on little routines that are collected in MODLIB.

Normally, the linker works in-memory, i.e. it links the program and then flushes its buffer to disk creating your executable program file this way.

Now, if your program gets really big, the linker starts to flush its buffer as soon as no more space for a module is available. This continues until either your program is linked completely, or 64 kBytes are reached. In the second case, we recommend for instance a VAX computer[1].

Another possibility to reach the linker's limits are to create single modules of more than 25000 bytes of code. This would equal to about 4000 source program lines, according to our experiences. In this case, the linker outputs a very concise error message:

```
---- Module Too Big
```

and aborts its operation. Don't you think you can split up such an enormous module?

b) Technical Description

ProgramName	ML
Input	Files of type **MRL.**
Output	A **COM** file, upon request (see /M) also a **MAP** file.
Switches	/C:xxxx, /D:xxxx, /H:xxxx, /J:xxxx, /L, /M, /N, /O:FileName, /S:FileName, /T:xxxx, /V, /X.

The **C** switch serves to set the code start address and is used in ROM applications only. (Default: /C:100)

The **D** switch is the data counterpart of /C. As the above one, it is used only in ROM applications. (Default: First data byte after last code byte)

[1] If you want your product to be mentioned here, please contact us. Make sure that it has more memory capacity than a Z80 and that a Modula-2 compiler (preferably made by ourselves) is available for it. Thank You.

The **H** switch allows to set the heap start to a specific address. The heap is the area used to dynamically allocate memory by means of the NEW and DISPOSE procedures. You need to set it in either ROM applications or in program systems with shared data. (See **Advanced Programming Guide** for more information regarding this topic. (Default: First heap byte after last data byte)

The **J** switch serves to reserve space in front of the actual program start. A jump around an area of xxxxH-3 bytes is placed in the first three code locations. You can insert a copyright message, for instance. (Default: no jump generated)

The **L** switch directs the linker to show the order in which the module initializations (the bodies of implementation modules) are executed, along with a list of the requests of each module. For each module, the linker shows if it has an initialization or not by a (+) or (-) following the module's name plus a list of the file names of all modules requested by the current one. (Default: no list of initalizations)

The **M** switch directs the linker to create a MAP file containing all visible symbols of a program. MAP files aren't SYM file compatible, i.e. you cannot read them into SID or ZSID. This is because the Modula-2 System allows for much longer names than the normal REL format. (Default: no symbol map)

Normally, every program initializes its heap on startup. To be able to retain the heap across several chained program parts (as for instance in the compiler), you can direct the linker to omit the code that does this initialization. This is done by specifying the **N** switch. Programs linked with the N switch are still stored as COM files, but when started from CP/M, they print the error message

---- **Modula-2 Runtime Error: Cannot Execute Chain File**

Refer to the **Advanced Programming Guide** for more information about this topic. (Default: Heap gets initialized on program startup)

The **O** switch allows to specify an arbitrary output file name instead of the main program file name. The COM as well as the MAP file will bear the specified name. (Default: output name is main program name)

The **S** switch allows the user to specify a shared data module. A shared data module **may not contain any code.** The end address of the shared data module is defined to be the address of the first heap byte - 1. So, it is located just below the heap. This implies that /S demands that /H is used, too. The **Advanced Programming Guide** contains more information about sharing data between several programs. (Default: no shared data module)

The **T** switch is used to set the Top Of Stack. This switch is used for ROM applications only. (Default: Stack Top is indicated by address found at locations 6,7 at program start)

The **V** switch sets the linker to the verbose mode. You will see the size of code and data of each of the nonMODLIB modules in your program, as well as a statistic about code-, data-, heap- and stack size, several start and end addresses, etc. In the normal mode, the linker just outputs a dot per module in both passes. (Default: non verbose)

The **X** switch directs the linker to delete the '$$$.SUB' file leading to abortion of a submit operation if errors occur during the linkage process. (Default: submit continues)

NOTE - The linker is able to detect **circular references.** These are created by so called **cross-imports.** These can be explained as follows: You program two library modules, say X and Y. X.MOD uses variables or procedures declared in Y.DEF. Y.MOD, on the other hand, uses variables or procedures declared in X.DEF. The whole thing is often referred to as chaotic programming style -- don't worry, this can happen to everybody. Such a (non-fatal) mishap is announced by a message of the form

---- Circular reference detected -- FILNAME1 FILNAME2

The order of the two initializations will be arbitrary. You can see how it is actually done by using the /L switch. Please note that there may be circular references among more than two modules. This is just a warning and therefore, it is non-fatal.

c) Anatomy of a linked Modula-2 Program

A linked program loaded into the memory looks as follows:

```
|                    |
|    high memory     |
+--------------------+<-- T switch value
|  stack grows down  |
|                    |
|   heap grows up    |
+--------------------+<-- H switch value
|  shared data (/S)  |
+--------------------+
|                    |
|    global data     |
|                    |
+--------------------+<-- D switch value
|                    |
|    program code    |
|                    |
+--------------------+
|initialisation calls|
+--------------------+<-- J switch value
|   reserved table   |
+--------------------+<-- C switch value
|    low  memory     |
|                    |
```

The default values of the switches are appropriate for CP/M .COM files.

d) Examples

Some examples should show you that this sounds all a lot more complicated than it actually is.

- The program CopyFiles is to be linked. Let's name it simply 'COPY'.

```
A>ML COPYFILE/O:COPY <CR>
MODULA-2 Linker for Z80 CP/M        Version 17-03-1985
P1..........
P2..........
A>
```

- Next, we will link the program ERATOS. The linker works in the verbose mode; it also generates a MAP file.

```
A>ML ERATOS/V/M <CR>
MODULA-2 Linker for Z80 CP/M                      Version 17-03-1985
P1
ERATOS                      Code:  265          Data:        8201
TERM1                       Code:  417          Data:          12
Initial Code     Start:  100H      End:  10DH   Length:        14
Code             Start:  10EH      End:  532H   Length:      1061
Data             Start:  533H      End: 2650H   Length:      8477
Heap             Start: 2651H
Stack and Heap   Space: 48037 bytes (on this System)

P2................

A>
```

This results in 'ERATOS.COM' and 'ERATOS.MAP'.

4. MP the Patcher

a) General Description

The patcher serves to install the search paths in the compiler and the linker. All other programs do not require installation. Normally, installation is performed once only. The system as distributed is installed as used for "normal" two drive systems, i.e. search paths "@AB" for both data and programs in compiler and linker.

WARNING - do NOT install on your distribution disks! Use your master diskette for this purpose.

b) Technical Description

ProgramName	MP
Input	User entered search path definition and MC.COM as well as ML.COM.
Output	Installed versions of MC.COM and ML.COM.
Switches	None.

c) Examples

Refer to the **Startup Guide** for sample installation dialogues.

5. MX the Reference Lister

a) General Description

This tool serves to create line numbered listings with or without an added reference table that lists for every identifier the numbers of each line on which it appears.

Every 16 lines, MX writes a dot to the console.

There are several formatting options provided, including page length, line length, direct output to the printer, and suppression of the table generation. The reference lister -as the compiler lister- automatically expands hard tabs.

The reference lister sets a header at the top of each page. For each program listing page, this header takes the form

```
Reference-Lister for MODULA-2      Version 19-03-1985      Page P-x
Listing for : FILENAME.TYP
```

On all reference table pages, the header looks like

```
Reference-Lister for MODULA-2      Version 19-03-1985      Page X-x
Listing for : FILENAME.TYP
```

Where 'x' denotes the actual page number in both headers.

Every 16 program lines, a dot is written to the console to show the program's progress.

b) Technical Description

ProgramName	MX
Input	Files of type DEF or MOD. Default file type is MOD, i.e. by specifying no file type, you will get a reference listing of the MOD file of the specified name. This is equal to the compiler's practice.
Output	Files of type REF, or direct printer output (see /P).
Switches	/I:ddd, /L:ddd, /P, /W

The **I** switch serves to define how many spaces the text shall be indented from left margin in characters. (Default: 15 characters indent)

The **L** switch allows to set the page length, i.e. the number of program lines that will be printed per page. Please note that the header is 3 lines high and cannot be changed. (Default: 60 program lines per page)

Setting the **P** switch routes the output directly to the printer instead of creating an output file. (Default: File output)

The **W** switch controls the generation of a reference table. Setting /W suppresses this table. (Default: table added to the listing)

c) Examples

- A listing of the definition module InOut is to be generated. No indent shall be made.

```
A>MX INOUT.DEF/I:0 <CR>
MODULA-2 Reference Lister for Z80 CP/M     Version 19-03-1985
Writing Reference Listing of INOUT.DEF
 .........
A>
```

- The implementation module InOut shall be listed without a reference table, output shall be routed to the printer directly.

```
A>MX INOUT/P/W <CR>
Modula-2 Reference Lister for Z80 CP/M     Version 19-03-1985
Printing Listing of INOUT.MOD
 ........................
A>
```

6. MR the REL-MRL Converter

a) General Description

This program is used only if you want to integrate assembler modules into your program. If you don't intend to use assembly language modules, you can skip this section.

To be able to integrate assembler modules into a Modula-2 program, you have to read the assembly language interfacing section in the **Advanced Programming Guide.**

The duty of MR is to convert machine code that is stored as a subset of the Microsoft REL format to the Modula-2 MRL format.

To accomplish this task, you have to provide

- The REL file of the module
- Eventually a name translation table stored in a R2M typed file.

The format of the name translation table is described in the above- cited assembly language interfacing section of the **Advanced Programming Guide.**

b) Technical Description

ProgramName	MR
Input	Files of type REL, eventually a translate table in a file of type R2M.
Output	A MRL typed file.
Switches	/C, /C:FileName, /O:FileName, /V

The **C** switch serves to specify a translation table file. Its name defaults to the name of the input file. If you want to use this R2M file, use /C only. By specifying a file name, the file FILENAME.R2M is used as a translation table. This is useful to build a 'library' of names to be translated and their translations. (Default: no translation table)

The **O** switch allows you to specify the output file name. This file name also serves as the default module base name, if no translation is done. An example: An assembled module containing the entry points GET and PUT is is translated by

```
A>MR IODRV/O:PORTIO <CR>
```

This results in an MRL module containg the entry points PORTIO.GET and PORTIO.PUT.

The **V** switch serves to switch MR into the verbose mode. Has only an effect if a translation table is used. In that case, all the translations are listed on the screen.

c) Examples

- Let us convert MOVES.REL to MOVES.MRL using MOVES.R2M as translation table.

```
A>MR MOVES/C/V <CR>
Modula-2 REL to MRL converter for Z80 CP/M    Version 27-12-1984
MOVES           --> Moves
MVL             --> Moves.MoveLeft
MVR             --> Moves.MoveRight
FILL            --> Moves.Fill
A>
```

Chapter 2. System Restrictions

There are several restrictions you have to watch for. These are:

- string constants may not exceed 128 characters. They have to be on a single source line.

- structures may have a maximum size of 32767 bytes. This applies to records and arrays.

- Standard procedure and function procedure names may not be redefined nor assigned to procedure variables.

- Type transfers by means of Type Identifiers are restricted to either both argument and result having array or record structure (structured) or being scalars (unstructured), even if the sizes are the same.

- CASE labels may be in the INTEGER range only (-32768 .. 32767). The compiler allows for 256 labels (where each element of a range counts as one label) per CASE statement.

- FOR statements may have upper bound values up to MAX(INTEGER) - StepWidth, or MAX(CARDINAL) - StepWidth, for FOR loops over the INTEGER or CARDINAL range, respectively. An attempt to use higher values generates an eternal loop because of the way bound conditions are checked in the generated code. The same phenomenon allows downcounting but to MIN(INTEGER) - StepWidth resp. StepWidth for INTEGER and CARDINAL FOR loops. The lower bound rule applies to enumerations, too.

- No runtime error checking for INTEGER/CARDINAL overflow and underflow, use of uninitialized variables or NIL pointer references is done. The only error that gets detected is a stack - heap collision.

- The maximum length of names which can be handled by the linker are 24 characters. All of these identifiers have the form "ModuleName.ObjectName". The compiler issues a warning, if two identifiers are differing only after the 14th character.

- There may be 16 LOOPs nested into each other, maximum.

- The compiler allows for 16 nesting levels, at most. This means that you cannot nest more than 16 modules, procedures and/or WITH statements. LOOP statements don't affect this nesting.

- No variable expressions may be passed to an ARRAY OF WORD. Constants and constant expressions as well as variables may be passed freely, however.

- The index of a ARRAY OF WORD is equal to the size (-1) of the object passed to it in **bytes.** It is suggested to access ARRAY OF WORD parameters bytewyse by using a POINTER TO CHAR. All this results from TSIZE(WORD) = 2.

Chapter 3. The Standard Library

Section 1. Introduction

Modula-2's great advantage is that you can buid toolboxes for specific purposes using library modules. So, an important part of each implementation of Modula-2 is a library that provides some basic tools as well as some more advanced ones. To have some portability ensured, a standard library has been defined by Prof. Wirth. Meanwhile, MODUS, the Modula-2 User's Society, works on an extended standard library. Some of these proposed modules are present in the utility library of this implementation (i.e. Strings, MathLib, Conversions and ConvertReal). As soon as the standard is fixed, we will adopt it. Up to that time, the Volition Systems approach will be adopted as far as possible.

The standard library provides several modules originally postulated by Prof. Wirth in **Programming in Modula-2.** These modules provide simple, easy to use, and portable I/O. Because of that portability, they are not best adapted to the environment which they run on, except perhaps for the Lilith computer. The higher the level, the more compromises were made regarding space requirements.

The whole standard library is constructed in levels or **layers.**

The top layer consists of InOut and RealInOut. It is designed to be used for dialog programming, i.e. for interaction between operator and computer.

The second layer is formed by Texts and RealTexts. This layer is in charge of the text functions, i.e. providing procedures to read from and write to files and devices in terms of text units like characters and lines. Also, it provides means to detect end-of-line and end-of-text marks.

End-of-text (EOT) may or may not be identical to the end-of-file, since the CP/M operating system accesses files in terms of 128-byte sectors, and you normally don't care about them. These sectors are "translated" to single characters, lines, numbers, etc. by the lowest standard library layer which consists of the modules SeqIO (sequential file access) and the utility modules Conversions and ConvertReal which provide conversions between ASCII text and the standard data types CARDINAL, INTEGER, and REAL. The CARDINAL conversion may be to any base from 2 to 63.

Generally speaking, these modules are useful to make programs that have to be portable among several implementations of Modula-2. If you want small programs, you have to resort to other I/O modules, eventually written by yourself. You can use all the utility library modules to accomplish this task.

There is one exception to what has been said above: The module **SYSTEM.** Although it is a standard module that is present in every Modula-2 implementation, its task is to allow machine access, forcing machine dependency. **SYSTEM is built into the compiler of each Modula-2 implementation.** It can be thought of as a part of the standard objects.

NOTE - **To keep the amount of paper 'eaten' away small, all of the listings of definition modules are presented here with most of the comments removed. Please list the sources on your program disks and keep these listings at hand when using the modules.**

Section 2. Terminal

1. General Description

The module Terminal is one of the standard modules postulated by Prof. Wirth in **Programming in Modula-2.** It contains routines to read and write characters and character strings from and to console.

2. Character Input

BusyRead returns either ASCII.nul (0C) if no character was typed, or the character that was typed. No echo occurs.

Read waits until a character is entered at the console. The character is then echoed to the console and returned.

3. String Input

The string input procedures ReadString and ReadLn offer the following common CP/M edit facilities:

- ASCII.bs (Backspace) or ASCII.del causes the deletion of the last character in the string, if there are any characters in it.
- Ctrl-X or ASCII.can clears the whole string entered so far.
- Ctrl-C or ASCII.etx causes program abortion and return to CP/M (HALT) if entered as the first character of the input. Otherwise, it has no effect.

ReadString terminates its operation as soon as a space or any character with a lower ordinal number in the ASCII alphabet is entered, except for the edit function characters. This is not quite comfortable on CP/M. Therefore, ReadLn offers a similar function, but only ASCII.cr or ASCII.lf are recognized as terminators. So, use ReadLn to get a 'normal' input line containing spaces etc. with a single call of an input procedure. The ReadLn procedure is compatible to the Volition Systems implementation of Terminal.

The character that caused termination of a read string operation is assigned to the variable termCH.

In general, it useful to read file names and single (text)word items with ReadString, and items that may contain blanks by ReadLn.

NOTE - ReadString as well as ReadLn ignore leading blanks.

4. The Interface

```
DEFINITION MODULE Terminal;


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


  VAR
    termCH: CHAR;


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

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

END Terminal.
```

Section 3. SeqIO (Files)

1. General Description

This module is not truly portable among several implementations; it has to make some compromises regarding machine dependency.

It is the standard file system provided with this implementation. It concentrates on buffered sequential file I/O and character oriented device IO. Random file I/O is provided by the module Files.

SeqIO offers several unique features that make it easy to use, yet it is full featured and very powerful.

2. File Names

The file names accepted by SeqIO procedures can have the following form:

```
FileName   = [Drive ":"] Name ["." Extension] | Device ":".
Drive      = '@' | .. | 'P' .
Name       = FnChar [FnChar [FnChar [FnChar [FnChar [FnChar
             [FnChar [FnChar]]]]]]] .
Extension  = FnChar [FnChar [FnChar]] .
Device     = "CON" | "KBD" | "TRM" | "LST" | "RDR" | "PUN".
FnChar     = 'A' | .. | 'Z' | 'a' | .. | 'z' | '0' | .. | '9' | '$' | '?' | '*'.
```

In words: A file name consists of an optional **drive specifier** denoting a CP/M drive (the character may be lower case) separated by a colon from the name part. The **name** part (which has to be specified) consists of 1 to 8 characters or digits. The optional **extension** (often referred to as the **file type**) is separated by a dot and consists of 1 to 3 characters or digits. Alternatively, you can also specify a **device name** which consists of three letters and a semicolon.

In name, extension, or device name, no interpunctuation symbols are accepted.

3. Error Handling

Errors may be processed in two ways when using SeqIO:

- The traditional way, testing for errors after each read/write operation, using the FileStatus procedure.
- The easier way using special error handler procedures, that can be assigned to a file after it has been created or opened.

A file's handler procedure is automatically invoked after a read/write error has been detected. A handler procedure has to comply to a procedure type with the following declaration:

```
TYPE FileHandler = PROCEDURE(VAR FILE);
```

To allow for general handlers, there is a utility procedure that retrieves a file's name. So, you can write literal error messages including the file's name and take any necessary actions (like closing the file).

WARNING - Handler procedures should limit their actions regarding SeqIO to closing the erroneous file. Otherwise, it would be possible that the handler gets re-invoked because of a second error.

To attach a handler to a file, use the SetFileHandler procedure after the file has been successfully opened or created.

4. Reading From Files

A typical command sequence to read from a file would be:

```
status := Open(file, name)
IF status = FileOK THEN
  SetFileHandler(file, MyHandler);
  WHILE NOT EOF(file) DO
    Read(file, char);
  END;
ELSE
  WriteErrorMessage(file, message).
END;
```

5. Writing to Files

To write to a file, you could use the following code:

```
status := Create(file, name)
IF status = FileOK THEN
  SetFileHandler(file, MyHandler);
  REPEAT
    Write(file, char);
  UNTIL dataExhausted OR FileStatus(f) # FileOK;
  status := Close(file);
  IF status = FileOK THEN
    WriteErrorMessage(status);
  END;
ELSE
  WriteErrorMessage(status).
END;
```

NOTE - There is no implicit file closing on program end in Modula-2. Files are unknown to the compiler, so it cannot generate such code. Therefore, a call to close is **mandatory** for output files. For input files, close is necessary to reclaim the space used by the file and its buffer.

6. Deleting Files

The Delete procedure which serves this purpose allows for ambigous file name specifications by including question marks and/or stars in the file name. Naturally, devices cannot be deleted... The procedure is straightforward:

```
status := Delete(name);
IF status # FileOK THEN
  WriteErrorMessage(status);
END;
```

7. Renaming Files

The file denoted by oldName is renamed to newName. No ambiguous file names can be used for the purpose. An example:

```
status := Rename(oldName, newName);
IF status # FileOK THEN
  WriteErrorMessage(status);
END;
```

8. Changing the File Buffer Size

SeqIO allows to set the size of a file's internal buffer. This buffer is assigned to each disk file. Device files don't need a buffer since devices are character oriented. The default buffer size is 1024 bytes. This equals 8 CP/M logical disk sectors. The buffer has to be one sector in size, minimum. Its maximum size is limited to 32k bytes or 256 sectors. If you attempt to set a larger size, or a zero buffer, the SetBufferSize procedure just leaves everything as it was. An example:

```
SetBufferSize(24); (* make 3k buffers *)
```

This statement directs SeqIO to assign 3k buffers to all files susequently opened or created. Already opened or created files aren't altered.

The bottleneck in disk I/O is almost always the transfer of data from and to the disk. An example: During the development of the Modula-2 Compiler, the buffer size of the files were changed from 512 bytes (4 sectors) to what was available between data end and shared data start. This resulted in buffers of about 1.5 kBytes. This increased buffer sizes made the compiler faster by a factor of almost 2 without any other changes!

There is some optimum measure for every case, though. Increasing the compiler's buffers further resulted in minor speedups. There are even some applications where it is desirable to use only a single sector buffer.

9. An Application of SeqIO

```
MODULE CopyFiles;

  FROM SeqIO IMPORT
     FILE, FileState, FileStatus,
     EOF,
     SetBufferSize,
     Open, Create, Read, Write, Close;

  FROM Terminal IMPORT WriteString, ReadLn, WriteLn;

  VAR
    f, g: FILE;
    ch  : CHAR;
    fName, gName: ARRAY [0..13] OF CHAR (* 'D:FILENAME.TYP' *)

  PROCEDURE CloseErrorMessage(name: ARRAY OF CHAR);
  BEGIN
    WriteString('---- Error In Closing File "');
    WriteString(gName);
    WriteString('"');
    WriteLn;
  END CloseErrorMessage;

BEGIN
  WriteString('CopyFiles'); WriteLn;
  LOOP
    WriteLn;
    WriteString('Copy From File (^C to abort): ');
    ReadLn(fName);
    IF Open(f, fName) = FileOK THEN
```

```
      REPEAT
        WriteLn;
        WriteString('Copy To   File (^C to abort): ');
        ReadLn(gName);
      UNTIL (fName # gName) & (Create(g, gName) = FileOK);
      WHILE NOT EOF(f) DO
        Read(f, ch);
        Write(g, ch);
      END;
      IF Close(g) # FileOK THEN
        CloseErrorMessage(gName);
      END;
    ELSE
      WriteString('---- Cannot Find "');
      WriteString(fName);
      WriteString('"');
      WriteLn;
    END;
    IF Close(f) # FileOK THEN
      CloseErrorMessage(fName);
    END;
  END; (* LOOP *)
END CopyFiles.
```

10. The Interface

```
DEFINITION MODULE SeqIO;

  EXPORT QUALIFIED
        FILE, FileState, FileHandler,
        FileStatus, SetFileHandler, DummyHandler, GetFileName,
        Open, Create, SetBufferSize,
        Close, Delete, Rename,
        Read, EOF, Write;

  TYPE
    FILE;

    FileState = (FileOK, DirOpsOK,
                 NameError, DeviceError, EndError, UseError);

    FileHandler = PROCEDURE(VAR FILE);


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

  PROCEDURE Delete(name: ARRAY OF CHAR; VAR reply: ErrorType): FileState;
  PROCEDURE Rename(old, new: ARRAY OF CHAR): FileStatel
  PROCEDURE Close(VAR f: FILE);

  PROCEDURE Write(VAR f: FILE; info: CHAR);
  PROCEDURE Read(VAR f: FILE; VAR info: CHAR);
  PROCEDURE EOF(f: FILE): BOOLEAN;

  PROCEDURE FileStatus(f: FILE): FileState;

  PROCEDURE DummyHandler(VAR f: FILE);
  PROCEDURE SetFilehandler(f: FILE; handler: FileHandler);
  PROCEDURE GetFileName(f: FILE; VAR name: ARRAY OF CHAR);

END SeqIO.
```

Section 4. Texts

1. General Description

Texts provides **text stream** operations. A TEXT variable gets connected (associated) to a previously opened or created FILE variable. Contrary to SeqIO's operations, Texts interprets the characters it reads to be able to detect the end of an input line (EOL) or the end of the text in the file, which is indicated either by a ^Z (ASCII.sub) or by the physical end of file as indicated by SeqIO.EOF. So, a TEXT variable is accessed either by character or by line. However, besides tab expansion and interpretation of the usual CP/M control characters in interactive input from CON:, no characters are altered or swallowed by Texts procedures.

The implementation of EOL and EOT is a lazy I/O interface, i.e. EOL and EOT become TRUE after the first "invalid" character has been read. This is necessary because of the interactive console I/O, where one can't foresee the end. This is in contrast to SeqIO's normal interface on files. SeqIO doesn't provide an end of file function for device files.

Three basic TEXT streams are offered by the module Texts: input, output and console. Input and output are initially connected to the console device (i.e. CON:). They may be redirected to other files. Console, however, cannot be redirected. This TEXT is used for messages to the console by higher level modules, if input and/or output have been redirected.

NOTE - A TEXT stream allows only for operations that are possible on the associated file, i.e. it is impossible to write to an input disk file or to read from the printer!

2. Text Output

Text output can be done using Write, WriteCard, WriteInt, WriteLn and WriteString. All of these procedures don't do any interpretation. You may write control characters as well as all printable characters. Especially, an end of line consists of a **cr, lf** sequence in Texts just to allow the use of standard CP/M text files with the Modula-2 System without you having to write your own text handling.

3. Text Input

Text input is implemented through the procedures Read, ReadAgain, ReadCard, ReadInt, and ReadLn.

The Read operation is transparent, i.e. it returns all characters it reads, but it sets the flags according to the currently read characters. The following rules are applied:

- EOT (end of text) becomes TRUE if either the associated file's physical end is reached (indicated by SeqIO.EOF), or if a ^Z (ASCII.sub) character is read.
- EOL (end of line) becomes TRUE if either EOT is TRUE or either an ASCII.cr or an ASCII.lf not immediately following a cr was encountered.

ReadAgain allows to re-read the last character read.

NOTE - Calling ReadAgain multiple times without Read-ing a character in between does **NOT** produce characters before the last read one, i.e. you cannot read the second-to-last read character again by using ReadAgain!

ReadCard and ReadInt read cardinal or integer numbers from a text stream. They skip any leading blanks or control characters and stop after the first character not being a part of a number (i.e. other than '0'..'9', '-' or '+'). EOL is only TRUE after ReadCard and ReadInt if the physical end-of-line has been encountered.

The ReadLn procedure reads the rest of a line from a text stream. If reading from the CON: or KBD: devices, ReadLn expands ASCII.tab to spaces according to CP/M rules. ^X restarts the input, backspace and delete delete the last character in the entered string. ^C aborts (HALTs) the program if entered as the first character.

NOTE - Texts does NOT close any files or disconnect any TEXT variables if a ^C is entered to ReadLn.

NOTE - After ReadLn, EOL is always TRUE by definition. EOT is set according to the above-mentioned rules.

If your string is too short to hold the currently read line, EOL is set to true and ReadLn returns after having overread the rest of the line.

4. Error Handling

Error Handling under Texts may be done using either explicit tests or a handler approach similar to SeqIO's. Handlers may be assigned to connected texts only. After disconnecting a text, the handler is not retained. So, you will have to set the handler after each new connection of a given TEXT.

NOTE - Whereas a FILE can be identified by a general purpose handler, this is not possible for TEXTs. Therefore, it is recommended to use either a very general handler or a separate handler for each text used.

5. An Example Program

The module ET expands tabs of an input file you specify. Note that actually it is simply a ReadLn/WriteString/WriteLn sequence of text operations that does this expansion.

```
MODULE ET;

  FROM Texts IMPORT
         TEXT, TextState, TextHandler,
         console, input, output,
         SetTextHandler,
         Connect, Disconnect,
         ReadLn, EOT, Write, WriteString, WriteLn;

  FROM SeqIO IMPORT
         FILE, FileState, FileHandler,
         SetFileHandler,
         GetFileName,
         Open, Create, Close, Delete, Rename;

  FROM Strings IMPORT Pos, Length, Concat;
  IMPORT Strings;        (* .Delete *)
  FROM ASCII IMPORT sub;

  VAR
    f,g: FILE;
    string: ARRAY [0..100] OF CHAR;
    inName,outName: ARRAY [0..13] OF CHAR;
  PROCEDURE WriteTextMessage(state: TextState);
  BEGIN
    CASE state OF
      FormatError : WriteString(console, 'Format Error');
    | FileError   : WriteString(console, 'File Error');
    | ConnectError: WriteString(console, 'Connect Error');
    END;
  END WriteTextMessage;

  PROCEDURE OutputError(state: TextState);
  BEGIN
    WriteLn(console);
    WriteString(console, 'Output: ');
    WriteTextMessage(state);
    HALT;
  END OutputError;

  PROCEDURE InputError(state: TextState);
  BEGIN
    WriteLn(console);
    WriteString(console, 'Input: ');
    WriteTextMessage(state);
    HALT;
  END InputError;

BEGIN
  WriteString(console, 'ExpandTabs');
  WriteLn(console);
  REPEAT
    WriteLn(console); WriteString(console, 'Input File: ');
    ReadLn(console, inName);
  UNTIL Open(f, inName) = FileOK;
  GetFileName(f, outName);
  Strings.Delete(outName,Pos('.', outName), Length(outName) - Pos('.',outName));
  Concat(outName, '.$$$', outName);
  IF (Create(g, outName) = FileOK) &
    (Disconnect(output) = TextOK) & (Connect(output, g) = TextOK) &
    (Disconnect(input) = TextOK) & (Connect(input, f) = TextOK) THEN
    SetTextHandler(input, InputError); SetTextHandler(output, OutputError);
    LOOP
      ReadLn(input, string);
      IF EOT(input) THEN EXIT; END;
      WriteString(output, string); WriteLn(output);
    END;
    Write(output, sub);  (* EOT marker *)
    IF (Disconnect(output) = TextOK) & (Disconnect(input) = TextOK) &
      (Close(g) = FileOK) & (Delete(inName) = FileOK) &
      (Rename(outName, inName) = FileOK) THEN
      WriteLn(console); WriteString(console, 'File "');
      WriteString(console, inName); WriteString(console, '" converted.');
    END;
  END;
END ET.
```

6. The Interface

```
DEFINITION MODULE Texts;

  FROM SeqIO IMPORT FILE;

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

  TYPE
    TEXT;
    TextState = (TextOK, FormatError, FileError, ConnectError);
    TextHandler = PROCEDURE(TextState);

  VAR
    input, output, console: TEXT;

  PROCEDURE EOL(t: TEXT): BOOLEAN;
  PROCEDURE EOT(t: TEXT): BOOLEAN;

  PROCEDURE TextStatus(t: TEXT): TextState;
  PROCEDURE SetTextHandler(t: TEXT; handler: TextHandler);
  PROCEDURE DummyTextHandler(TextState);

  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.
```

Section 5. RealTexts

1. General Description

RealTexts provides two procedures that read and write real numbers to text streams.

2. WriteReal

The parameter n in WriteReal determines the overall length of the resulting ASCII representation of the REAL variable r on the text stream t.

The parameter digits is used to determine the number of fractional digits to be calculated. The maximum number of digits should not exceed 7, since the REAL data type does not provide for more than 7 significant digits. If digits < 0, then the exponential representation is produced (i.e. 3.141593E+00); if digits = 0, no fraction and no decimal dot is generated (i.e. 100107). If digits > 0, the fixed point representation (2.718282) with digits fractional digits is produced.

3. ReadReal

ReadReal reads the REAL variable r from the text stream t. It disregards leading blanks and control characters and stops after the first character that is not part of the REAL number, similar to the number reading procedures found in Texts.

4. The Interface

```
DEFINITION MODULE RealTexts;

  FROM Texts IMPORT TEXT;

  PROCEDURE ReadReal(VAR t: TEXT; VAR r: REAL);
  PROCEDURE WriteReal(VAR t: TEXT; r: REAL; n: CARDINAL; digits: INTEGER);
END RealTexts.
```

Section 6. InOut

1. General Description

InOut provides standard I/O with redirection from the console to files. It is the highest level module in the standard library. Its definition is contained in **N.Wirth: Programming in Modula-2** on page 103 ff.

2. I/O Redirection

The I/O redirection procedures (OpenInput, OpenOutput) always ask for a file name at the console. They issue the prompts 'in>' respectively 'out>' and then wait for you to enter a file name. If the file couldn't be found, the boolean variable Done is set to FALSE. Upon successful operation, TRUE is returned in Done.

If the file name that was entered has no dot ('.') in it, the two procedures append the **default file type** (also called extension) defext to it. The first three characters in defext are considered, at most. All file names are uppercased; there is no way to include lower case characters in file name using either InOut or the standard file system SeqIO. You can include a drive specifier in the file name, or enter one of the following CP/M device names:

CON: (console I/O; uses CP/M standard console I/O, checks for ^C)
KBD: (Keyboard input; uses CP/M direct input, checks for ^C)
TRM: (Screen output; uses CP/M direct output)
LST: (Printer output; uses CP/M list output)
RDR: (Reader input; uses CP/M reader input)
PUN: (Punch output; uses CP/M punch output)

To return to normal console I/O (CON:), use the procedures CloseInput and CloseOutput, respectively.

NOTE - A call to CloseOutput is mandatory to record an output file permanently.

If a program that uses input redirection ends without executing a CloseInput, no damage whatsoever will effect from this, although this practice isn't recommended.

NOTE - The external files connected to input and output use some heap space. Failing to close input (or output) does not reclaim this heap space until the program terminates, i.e. returns to CP/M.

All the read operations take place on the current input file, or the console if no redirection is in effect.

All write operations go to the current output file, or to the console if no redirection is in effect.

NOTE - There is no provision to redirect the input or output from within your program. Only the console operator can redirect it.

3. Input

There are procedures for character and for line input.

Read provides for single character input. Read interprets ASCII.cr as the line end character. If a line feed follows the carriage return, it gets swallowed by Read. This complements the behaviour of Write. If multiple LF's follow a CR, only the first one gets stripped.

The last character read can be re-read by calling ReadAgain.

NOTE - ReadAgain returns but the last character read. There is no way to retrieve, for instance, the second-to-last character read.

ReadString stops reading as soon as a white character is entered. A white character is anything that is below 41C and is not an editing character. This means, a space and every control character except ^X, BS, DEL or ^C stops the read operation.

The edit functions available are:

DEL: delete last character in string.
BS: same as above.
^X: clear whole string; start over.
^C: immediate program abort. The program immediately returns to the Operating System without closing any files, in particular.

---

4. Output

As stated in the standard, WriteLn has the same effect as Write(EOL). In our implementation, this means that writing an ASCII.cr character forces the writing of a ASCII.lf character. Read does complement this behaviour by swallowing a lf immediately following a cr.

5. An Example Program

The following small program types a file to the console, much like CP/M's TYPE command, but paged, i.e. it waits after every screen it wrote for you hitting any key to continue with the next screen.

```
MODULE TypeAFile;

  FROM InOut IMPORT
    EOL,
    Done,
    OpenInput, CloseInput, Read, Write, WriteLn, WriteString;

  IMPORT Texts; (* the console text *)


  CONST
    linesPerScreen = 23;       (* leave one for prompt *)
  VAR
    ch: CHAR;
    lines: CARDINAL;


BEGIN
  WriteString('Paged File Typing'); WriteLn;

  LOOP
    WriteString('Enter Name of File to Type: ');
    OpenInput('');               (* prompts on next line *)
    IF NOT Done THEN EXIT; END;
    lines := 0;
    LOOP
      Read(ch);
      IF NOT Done THEN EXIT; END;
      Write(ch);
      IF ch = EOL THEN
        INC(lines);
        IF lines = linesPerScreen THEN
          WriteString('-- More --');
          Text.Read(Text.console, ch);     (* get input from console *)
          lines := 0;
          WriteLn;
        END;
      END;
    END; (* LOOP *)
    CloseInput;
    WriteLn;
    WriteString('--- EOF ---');
    WriteLn;
  END;
END TypeAFile.
```

6. The Interface

```
DEFINITION MODULE InOut;

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

  CONST
    EOL = 15C;

  VAR
    Done: BOOLEAN;

  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.
```

Section 7. RealInOut

1. General Description

The purpose of this module is to allow REAL number input and output. It is one of the standard modules defined in **Programming in Modula-2.** The WriteRealOct procedure, which is hardly used by application programs, has been removed from this module. More versatile REAL output formatting can be found in the **ConvertReal** and **RealTexts** modules. RealInOut internally relies upon these modules.

2. REAL Input

REAL numbers are accepted nearly free-form. The EBNF notation of the accepted input looks like:

```
ReadableReal   = Sign Mantissa [('E'|'e') Sign Number].
Sign           = ['+'|'-'].
Mantissa       = Number ['.' Number] | '.' Number.
Number         = Digit {Digit}.
Digit          = '0'|..|'9'.
```

Examples of correct REAL numbers:

1.0   0   0.0000000007   1E38   1.978e-9   +15.793e-5   20000.9

The following conditions have to be fulfilled by your input:

- The maximum REAL value is about 1.7e38 ($< 2^{127}$), the smallest non-zero, positive REAL is about 3e-39 (2^{-128}).

- The chosen format allows for about 7 digits of precision because it uses 24 binary mantissa digits internally. Numbers with more significant digits will be truncated; the digits that aren't representable are discarded (if they are behind the '.') or used to get the exponent.

- The maximum exponent of a number is 38. The minimum value that leads to non-zero numbers is -39.

---

3. REAL Output

The output formatting is rather limited, because it uses always the scientific format. The scientific format contains up to 7 mantissa digits. The minimum length of the output is 6 characters ('0.E+00'). Setting the format-parameter n to more than 12 results in leading blanks in the output.

4. The Interface

```
DEFINITION MODULE RealInOut;

  EXPORT QUALIFIED ReadReal, WriteReal;

  PROCEDURE ReadReal(VAR r: REAL);
  PROCEDURE WriteReal(r: REAL; width: CARDINAL);

END RealInOut.
```

Section 8. MathLib

1. General Description

MathLib was originally postulated by Prof. Wirth in **Programming in Modula-2** (page 85) as **MathLib0.** In an attempt to standardize a larger library, MODUS (Modula-2 Users Society) has created a draft standard that adds the **power** function to the original MathLib0 and names it simply MathLib.

Please keep in mind that REALs have a very finite precision. The calculations allow for a theoretic maximum precision of about 7.2 decimal digits (24 binary mantissa bits --> log10(2^{24}) = 7.2...). For numbers near the maximum representable REAL number, the difference between two adjacent representable numbers is $2^{(127-24)} = 2^{103} = 10^{31}$. So, there is no use to print results of any calculations with more than 7 significant digits -- you just generate a "better" precision than the computer does.

The algorithms used were chosen because their **average** performance is sufficient and their execution times compare favorable to those of most competitors. None of the calculations are iterative. If series are evaluated, only a fixed number of components gets calculated. These components are spelled out instead of iterated.

If you are mathematically inclined or in need of better precision, D.E. Knuth offers a wealth of numerical algorithms in volume 2 of **The Art of Computer Programming, Seminumerical Algorithms.**

2. Error Handling

Errors during the calculation of functions are fatal. The program is halted with an appropriate message to the terminal using the Terminal module. An exception hereto are sin and cos, which may cause an overflow if too big arguments are passed to them.

3. Trigonometric Function Procedures

MathLib contains a set of trigonometric function procedures, namely the sine (sin), the cosine (cos) and the arctangent (arctan). Out of this set, every other trigonometric function can be calculated. In fact, even the cosine isn't necessary to do so.

WARNING - Do not use arguments greater than 32767.0 for sine and cosine. Due to the algorithm used for these procedures, an overflow could occur otherwise.

4. Exponential and Power Function Procedures

The set of exponential and power functions includes the natural logarithm (ln) to the base e = 2.7182818. Note that the precision of the calculations is limited as is the range. The largest argument of exp leading to a successful calculation is about 88 (< ln(2^127)). The natural logarithm works on positive numbers greater than 0 only.

The power function allows to raise a REAL number x to the REAL power y.

The square root sqrt works on positive arguments only.

WARNING - exp arguments may be up to 88, power allows for x = ln(MAX(REAL)) / ln(y) , maximum, ln arguments must be greater than 0 and sqrt doesn't accept negative argument values.

5. Conversion Function Procedures

The entier and real functions provide conversions between INTEGERs and REALs in both ways.

WARNING - entier and real work on the INTEGER range only. entier returns the maximum or minimum INTEGER values for arguments that are out of the integer bounds.

6. The Interface

```
DEFINITION MODULE MathLib;

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

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

END MathLib.
```

Section 9. SYSTEM

1. General Description

This module is present in every Modula-2 implementation. It covers everything necessary to go down to the machine level. Its use most certainly makes a module nonportable.

NOTE - Although SYSTEM.DEF is listed here, it is in fact built into the compiler. This is inevitable because it contains some generic compile time functions(i.e. TSIZE, SIZE). Therefore, you cannot modify SYSTEM or create your own implementation of it.

2. Normal Contents

The ADDRESS type is officially declared as a pointer to a word. It is assignment compatible with all pointer types.

The type WORD is a 16 bit value. Its special property is that it can serve as a general purpose 16 bit parameter. This means that a WORD parameter can be substituted by INTEGERs, CARDINALs, enumerations of more than 256 elements and subranges thereof, ADDRESSes, and pointers.

The function ADR returns the adress of its parameter. The parameter may be any variable, but no procedure. Types and constants do not have addresses; they are so called **compile time objects.**

The SIZE and TSIZE functions return the size of their parameter. In SIZE, any variable may be passed as a parameter. In TSIZE, the parameter is a type, optionally followed by tagfields (given your parameter is a variant record). In SIZE, no tagfields may be specified.

3. Special Additions: Storage Management

The procedures ALLOCATE and DEALLOCATE are normally located in a module called Storage. This would allow for writing a proprietary heap management. This possibility does not exist in our implementation, at the moment. Currently, a first fit algorithm allocating at least 4 bytes per item, for a bigger item just its size, is used.

ALLOCATE returns NIL if there wasn't enough room to allocate a given item in the heap.

The Storage module will be supplied in a later version.

NOTE - for ROM-based applications, it is necessary to know that the heap has to **reside below the stack** in the memory. This is necessary because a stack-heap collision test is made before each procedure entry and also before each load of a longer object (record, array or string constant) onto the stack. The runtime library source is available for customers needing access to it.

4. The Interface

```
DEFINITION MODULE SYSTEM;      (*--- COMPILER BUILT IN ---*)

  EXPORT QUALIFIED
    ADDRESS, WORD,
    ADR, SIZE, TSIZE,
    ALLOCATE, DEALLOCATE;

  TYPE
    WORD;
    ADDRESS = POINTER TO WORD;

  PROCEDURE ADR(AnyVar): ADDRESS;
  PROCEDURE SIZE(AnyVar): CARDINAL;
  PROCEDURE TSIZE(AnyType, tag1, ....): CARDINAL;

  PROCEDURE ALLOCATE(VAR pointer: ADDRESS; size: CARDINAL);
  PROCEDURE DEALLOCATE(VAR pointer: ADDRESS; size: CARDINAL);

END SYSTEM.
```

Chapter 1. The Utility Library

Section 1. Introduction

The utility library's purpose is to make life a little bit easier for the programmer by providing an extended environment.

This includes modules that provide fast memory transfers and initialization, operating system access, string handling primitives, number/string conversions, ASCII control character constant definitions, as well as others.

These modules provide you with a base to make your own modules that fit specific needs of a program, as for instance fully checked I/O procedures, etc.

The examples in this chapter most often are a little bit narrow-chested and do not cover any and all possible uses of the modules. Feel free to find your own applications!

Section 2. ASCII

1. General Description

ASCII provides all ASCII control characters by their official ASCII shorthands.

The related module **Controls** provides the same constants as control characters as entered at the keyboard (CtrlA,...).

2. The Interface

```
 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.
```

Section 3. Chaining

1. General Description

The module Chaining provides a procedure Chain that can handle all the special formats generated by the linker's diverse switches (/J,/N). Its usage is described in detail in the **Advanced Programming Guide.**

The file names accepted by Chain are normal CP/M file names consisting of an optional drive specifier, the file name, and the optional type.

Examples of file names accepted by Chain:

```
"B:CUSTOMER.COM"   "A"   "OTHELLO.COM"
```

NOTE - If the specified file can't be found, Chain returns to the calling program without taking any actions. This is convenient to use with code like:

```
LOOP
  Chain(nextProgram);
  WriteLn;
  WriteString('Please Load Program Diskette Into Drive A:');
  WriteString('And Hit Any Key To Continue: ');
  ResetDiskSystem;
END;
```

2. The Interface

```
DEFINITION MODULE Chaining;

  EXPORT QUALIFIED
     Chain;

  PROCEDURE Chain(fileName: ARRAY OF CHAR);

END Chaining.
```

Section 4. CmdLin

1. General Description

The CmdLin module provides a procedure that reads the CP/M command line into a string.

If the commmand line isn't empty, it is checked for correctness. If a space is entered before the program's name, CP/M puts the program name into the command line, too. The default file control blocks, however, are set up correctly. Using this fact, CmdLin finds the correct beginning of the command line. Simply, if you enter (your input underlined)

```
A> copy alpha beta <CR>
```

or

```
A>copy alpha beta <CR>,
```

CmdLin will return ' alpha beta' as command line.

If the CP/M command line (stored at 80H .. 0FFH) is empty, the ReadCommandLine procedure displays a star prompt and awaits the user's input.

ReadCommandLine can be called multiple times. It returns the CP/M command line but on the first call, if the command line isn't empty at that time. Afterwards, the star prompt is displayed and user input awaited. This input is read with the aid of Terminal.ReadLn, so all the editing features possible there are also present in CmdLin, including program abortion by ^C.

The command line returned by ReadCommandLine is guaranteed to be non-empty.

2. The Interface

```
DEFINITION MODULE CmdLin;

  EXPORT QUALIFIED ReadCommandLine;

  PROCEDURE ReadCommandLine(VAR commandLine: ARRAY OF CHAR);

END CmdLin.
```

Section 5. Controls

1. General Description

This module provides most of the keyboard enterable ASCII control characters. Characters like CtrlBackslash aren't included; if they are used, they are most often entered by special keys, so you will most certainly use the ASCII names for those characters.

For the official ASCII control character names, please refer to the module **ASCII.**

2. The Interface

```
DEFINITION MODULE Controls;

  EXPORT QUALIFIED
        CtrlAt,CtrlA, CtrlB, CtrlC, CtrlD, CtrlE, CtrlF, CtrlG,
        CtrlH, CtrlI, CtrlJ, CtrlK, CtrlL, CtrlM, CtrlN, CtrlO,
        CtrlP, CtrlQ, CtrlR, CtrlS, CtrlT, CtrlU, CtrlV, CtrlW,
        CtrlX, CtrlY, CtrlZ;

  CONST
    CtrlAt= 00C; CtrlA = 01C; CtrlB = 02C; CtrlC = 03C;
    CtrlD = 04C; CtrlE = 05C; CtrlF = 06C; CtrlG = 07C;
    CtrlH = 10C; CtrlI = 11C; CtrlJ = 12C; CtrlK = 13C;
    CtrlL = 14C; CtrlM = 15C; CtrlN = 16C; CtrlO = 17C;
    CtrlP = 20C; CtrlQ = 21C; CtrlR = 22C; CtrlS = 23C;
    CtrlT = 24C; CtrlU = 25C; CtrlV = 26C; CtrlW = 27C;
    CtrlX = 30C; CtrlY = 31C; CtrlZ = 32C;

END Controls.
```

Section 6. Strings

1. General Description

This module provides the some utility procedures to handle character strings conveniently, and the STRING data type.

String variables are stored in character arrays. You can use any CARDINAL based index you like, but:

NOTE - Strings procedures assume that an array's low index is 0 and all indices are calculated for that low index. If you use another low index, do not forget to account for the actual low index in all indices passed to or received from a Strings procedure.

If a string variable's contents is shorter than the variable itself, a 0C (ASCII.nul) character determines the end of the valid contents by convention. If length of variable and contents are the same, this trailing nul character is omitted.

NOTE - If you assign the contents characterwise, do not forget to put the trailing nul or the results of Strings procedures will be rather surprising!

String constants may be up to 128 characters long in this implementation. In contrast to Pascal, string constants of any length up to the size of an array may be assigned to it. If they are shorter, at least one nul is added, if they are same size, no nul will be present. If a string constant is longer than the array it should be assigned to, the compiler issues an error.

NOTE - Open Array Parameters may be accessed element-by-element only. You may not assign a string constant to an Open Array Parameter. This restriction is necessary because it cannot be asserted that enough space will be available until the call of the respective procedure is executed.

An example of the usage of the Strings module:

```
PROCEDURE FileType(fileName: ARRAY OF CHAR; VAR fileType: ARRAY OF CHAR);
VAR
  i,j: INTEGER;
BEGIN
  i := Pos('.', fileName);
  IF i > HIGH(fileName) THEN
    fileType := '   ';
  ELSE
    Delete(fileName, 0, i + 1);
    fileType := fileName;      (* delete also '.' *)
    i := 0;
    LOOP                          (* get legal characters *)
      ch := fileType[i];
      IF (i = 3) OR (ch < ' ') OR (ch = '.') OR (ch = ':') THEN
        EXIT;
      END;
      INC(i);
    END; (* LOOP *)
    WHILE i < 3 DO                (* pad with blanks *)
      fileType[i] := ' ';
      INC(i);
    END; (* WHILE *)
    fileType[3] := 0C;          (* end of string *)
  END;
END FileType;
```

This procedure parses a CP/M file type which consists of 3 characters in its final form. Here, lacking characters are supplied, additional ones clipped, etc.

2. The Interface

```
DEFINITION MODULE Strings;


  EXPORT QUALIFIED
        STRING,
        Assign, CompareStr, Concat, Copy,
        Delete, Insert, Length, Pos;

  CONST
    StringLength = 80;

  TYPE
    STRING = ARRAY [0..StringLength - 1] OF CHAR;


  PROCEDURE Assign(VAR source, dest: ARRAY OF CHAR);
  PROCEDURE CompareStr(string1, string2: ARRAY OF CHAR): INTEGER;
  PROCEDURE Concat(string1, string2: ARRAY OF CHAR;
                   VAR result: ARRAY OF CHAR);
  PROCEDURE Copy(string: ARRAY OF CHAR; inx, len: CARDINAL;
                   VAR result: ARRAY OF CHAR);
  PROCEDURE Delete(VAR string: ARRAY OF CHAR; inx, len: CARDINAL);
  PROCEDURE Insert(subString: ARRAY OF CHAR; VAR string: ARRAY OF CHAR;
                   inx: CARDINAL);
  PROCEDURE Length(VAR string: ARRAY OF CHAR): CARDINAL;
  PROCEDURE Pos(subString, string: ARRAY OF CHAR): CARDINAL;

END Strings.
```

Section 7. LongSets

1. General Description

LongSets is provided to use it to build Pascal-like character sets. It is intended as a simple example for how to build a utility module. Although you cannot define the resulting sets directly in the program, you can use a LONGSET as a replacement for such a declaration. A very simple, but effectful enhancement of this module would be the introduction of a

```
PROCEDURE InclChars(VAR set: LONGSET; chars: ARRAY OF CHAR);
```

Try it!

An example:

```
MODULE LongSetExample;

TYPE
  CharSet = LONGSET;

VAR
  answer: CharSet;
...

BEGIN
  Init(answer);
  InclRange(answer, ORD('a'), ORD('g'));
  REPEAT
    Read(ch);
  UNTIL Included(answer, ORD(ch));
END LongSetExample;
```

This module does Pascal-style input checking.

2. The Interface

```
DEFINITION MODULE LongSets;


  EXPORT QUALIFIED
        LONGSET,
        Init, Unite, Diff, InterSect, Xor,
        Include, InclRange, Exclude, Included;


  CONST
    LongBits = 256;                (* # of bits in long sets *)
    BitsPerSet = 16;               (* bits in a BITSET *)

  TYPE
    LONGSET = ARRAY [0 .. (LongBits-1) DIV BitsPerSet] OF BITSET;


  PROCEDURE Init      (VAR set: LONGSET);
  PROCEDURE Unite     (s1, s2: LONGSET; VAR union: LONGSET);
  PROCEDURE Diff      (s1, s2: LONGSET; VAR difference: LONGSET);
  PROCEDURE InterSect (s1, s2: LONGSET; VAR intersect: LONGSET);
  PROCEDURE Xor       (s1, s2: LONGSET; VAR symSetDiff: LONGSET);
  PROCEDURE Include   (VAR set: LONGSET; bit: CARDINAL);
  PROCEDURE InclRange (VAR set: LONGSET; start, end: CARDINAL);
  PROCEDURE Exclude   (VAR set: LONGSET; bit: CARDINAL);
  PROCEDURE ExclRange(VAR set: LONGSET; start, end: CARDINAL);
  PROCEDURE Included  (set: LONGSET; bit: CARDINAL): BOOLEAN;

END LongSets.
```

Section 8. Conversions

1. General Description

Conversions provides several standard conversions between INTEGERs, CARDINALs and character strings. Additionally, two general conversion routines are provided that allow for arbitrary conversion bases between 2 and 63 using characters (upper- and lower case) as digits, if necessary.

All of the number writing procedures contained in InOut are based upon these routines.

All of these routines are implemented as BOOLEAN function procedures. In conversions to string representation, only an impossible base can cause a FALSE return value. The opposite direction conversions return FALSE if wrong digits were used, or if an overflow occured.

An example:

> I suppose that you are as curious to see some numbers represented using base 51 as I am. So, let's have a look:

```
MODULE Base51;

  FROM Terminal IMPORT ReadString, WriteString, WriteLn, Write;
  FROM Conversions IMPORT NumToStr, StrToCard;

  CONST Base = 51;

  VAR
    asciiRepresentation: ARRAY [0..7] OF CHAR;
    dualRepresentation: CARDINAL;

BEGIN
  WriteString('Conversion of decimal numbers to base 51'); WriteLn;
  WriteLn;
  WriteString('Enter your number: ');
  ReadString(asciiRepresentation);
  IF StrToCard(asciiRepresentation, dualRepresentation) &
    NumToStr(dualRepresentation,Base,asciiRepresentation) THEN
    WriteString(' = ');
    WriteString(asciiRepresentation);
    WriteLn;
  END;
END Base51.
```

Perhaps there are even more useful applications of the general conversion routines. They were introduced mainly because they shorten the code of Conversions quite a bit.

2. The Interface

```
DEFINITION MODULE Conversions;


  FROM SYSTEM IMPORT
        WORD;

  EXPORT QUALIFIED
        ConvertToNumber, ConvertToString,
        IntToStr, StrToInt, CardToStr, StrToCard, HexToStr, StrToHex;


(* NOTE -- string arguments cannot have any leading or trailing blanks;      *)
(*          result is TRUE upon successful conversion.                       *)


  PROCEDURE NumToStr((* convert *)     num   : CARDINAL;
                     (* using   *)     base  : CARDINAL;
                     (* into    *) VAR str   : ARRAY OF CHAR): BOOLEAN;

  PROCEDURE StrToNum((* convert *)     str   : ARRAY OF CHAR;
                     (* using   *)     base  : CARDINAL;
                     (* into    *) VAR num   : CARDINAL): BOOLEAN;

  PROCEDURE IntToStr(int: INTEGER; VAR str: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE StrToInt(str: ARRAY OF CHAR; VAR int: INTEGER): BOOLEAN;

  PROCEDURE CardToStr(card: CARDINAL; VAR str: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE StrToCard(str: ARRAY OF CHAR; VAR card: CARDINAL): BOOLEAN;

  PROCEDURE HexToStr(hex: WORD; VAR str: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE StrToHex(str: ARRAY OF CHAR; VAR hex: WORD): BOOLEAN;

END Conversions.
```

Section 9. ConvertReal

1. General Description

ConvertReal includes procedures that do conversions between strings and REAL data. This module is part of a standard library proposed by MODUS.

A real number consists of two major parts: The **mantissa** and the **exponent.** Both can have a sign. The mantissa may be any floating point number, i.e. 1.8, -1567.04, 0.000002, etc. The exponent may be an integer number in the range -39 up to and including 38. Note that numbers that are too small to be different from zero, are automatically set to zero. No error is indicated in this case. On the other hand, if the upper limit of $2^{127} = 1.7..* 10^{38}$ is reached or surpassed in a calculation, a fatal runtime error occurs.

2. StrToReal

The input procedure StrToReal converts an ASCII string into the internal REAL format. A legal REAL number has to comply to the syntax:

```
AcceptedReal   = Sign Mantissa [('E' | 'e') Sign Number].
Sign           = ['+' | '-'].
Mantissa       = Number ['.' Number] | '.' Number.
Number         = Digit {Digit} .
Digit          = '0' | .. | '9'.
```

The maximum range of the REAL number format is +/- (5e-39..1.70141 18e38). The exponent -the number after 'E' or 'e'- is to the decimal base. Thus, the REAL number is composed as mantissa * 10^{exponent}.

NOTE - Differing from the compiler's real number syntax which allows it to determine the type of a constant, the dot may be omitted in input to this procedure.

Errors during a conversion are indicated by setting the BOOLEAN variable parameter success to FALSE. Errors may occur because of numbers too big to be represented, or if illegal characters are in the string holding the number.

3. RealToStr

This procedure converts a REAL real to the string str producing width digits or blanks. The decPlaces parameter serves different purposes: If decPlaces is greater than zero, the REAL is converted to a fixed-point representation having decPlaces decimal places (for example '1.50' for decPlaces = 2). If decPlaces is zero, an integer representation of the number without a '.' (100000) is produced. If it is negative, the scientific notation (1.8E+10, etc.) is created. In any case, the string contains leading blanks if the number is not exactly width characters long. If the scientific format has been chosen, at least 1 and at most 8 significant digits are output for the mantissa. The value of width determines the actual number of digits according to the formula

digits = width - ExponentChars (4) - Dot (1) - sign (0 or 1).

success is set to FALSE if the conversion couldn't be accomplished due to too small a field width. If width is greater than the size of the string, an error gets flagged, too.

The minimum field widths are:

Integer:	1	
Decimal:	2 + decPlaces	("0." + decPlaces)
Scientific:	6	("0.E+00")

For each representation, negative numbers require one more character.

NOTE - Due to the REAL number format, only about 7 digits are significant in a number.

4. The Interface

```
DEFINITION MODULE ConvertReal;

  EXPORT QUALIFIED
    RealToStr, StrToReal;


  PROCEDURE RealToStr((* converts *)     r          : REAL;
                      (* using    *)     width      : CARDINAL;
                      (* and      *)     decPlaces  : INTEGER;
                      (* into     *) VAR s          : ARRAY OF CHAR;
                      (* if       *) VAR success    : BOOLEAN);

  PROCEDURE StrToReal((* converts *)     s          : ARRAY OF CHAR;
                      (* into     *) VAR r          : REAL;
                      (* if       *) VAR success    : BOOLEAN);

END ConvertReal.
```

Section 10. FileNames

1. General Description

This module provides conversion of file name strings into the CP/M File Control Block (FCB) format and vice versa. Wildcards ('?' and '*') are possible in both conversion directions. A separate procedure that returns error message strings is also supplied.

These procedures are used internally by SeqIO and Files to parse file names. FileSys uses a primitive AssignName procedure that accepts file names in the form output by StrToFCB.

2. StrToFCB

StrToFCB scans the file name given in nameStr and returns a FCBFileName, which has the form required by the CP/M operating system. A default file name is passed to StrToFCB through the output parameter FCBName. The default fields are substituted for any missing fields in the scanned string. A typical calling sequence of StrToFCB looks like:

```
WriteString("Please Enter the File's Name: ");
ReadLn(s);
name.disk := myDisk; name.name := myName; name.type := myType;
nameState := StrToFCB(s, name);
IF nameState = NameOK THEN
  ...
ELSE
  NameErrorMessage(nameState);
END;
```

If you don't want to specify a default, use the following statement to specify the default:

```
name.text := '';
```

The CP/M default drive is set if neither the string nor the default value set any disk. A blank file type is returned if neither the string nor the default do specify a file type. An error message is issued if neither the string nor the default do specify a file name.

NOTE - The file name has to be specified in any case, be it by default or by the filename string.

3. Accepted Names

The StrToFCB procedure allows for unambigous file names (i.e. "LETTER.TXT") as well as for ambigous ones like "M*.MOD" or "A?ORT.S*". It does not allow for unambigous characters after a star. In the returned file name, stars are expanded to the appropriate number of question marks.

NOTE - The file systems do not allow for ambigous file specifications for open, create and rename operations. Delete, however, allows this kind of wildcard names, because CP/M's delete operation supports them directly.

4. Return Values

StrToFCB returns a value of type NameState. If the result is NameOK, then a correct, unambigous file name has been scanned. StrToFCB returns WildOK, if any wildcard characters ('?') are contained in the completed file name. Such file names are useful for CP/M directory searches or delete operations.

DeviceOK is returned if one of the following CP/M device names has been scanned: CON:, KBD:, TRM:, LST:, RDR:, or PUN:.

A DriveError is returned if either an impossible drive character has been specified (legal are '@' up to and including 'P', lowercase is converted to uppercase).

NoNameDefault gets issued if in the final, assembled file name, the name field is filled completely with spaces.

The IllegalChars result is returned if characters not contained in 'A'..'Z', '0'..'9', ':', '.', '?' and '*' are found in the scanned file name.

5. FCBToStr

This procedure returns a string containing the file name found in the given File Control Block. The boolean parameter formatted decides whether or not the output string gets formatted. This means, that the file name is fixed to the format "D:FileName.Typ". If the file name and the type aren't full length, they are padded with blanks. The drive is omitted if the file resides on the default drive. If formatted is set to FALSE, the padding isn't done.

NOTE - FCBToStr re-folds question mark sequences to stars, if applicable, i.e. if they are at the end of a field.

6. The Interface

```
DEFINITION MODULE FileNames;

  FROM OpSys IMPORT FCBFileName;

  EXPORT QUALIFIED NameState, StrToFCB, FCBToStr;

  TYPE
    NameState = ( NameOK, WildOK, DeviceOK,
                  DriveError, NoNameDefault, IllegalChars);

  PROCEDURE StrToFCB(     nameStr : ARRAY OF CHAR;
                      VAR FCBFile : FCBFileName): NameState;

  PROCEDURE FCBToStr(     FCBFile   : FCBFileName;
                      VAR nameStr   : ARRAY OF CHAR;
                          formatted : BOOLEAN);

END FileNames.
```

Section 11. FileSys

1. General Description

The module FileSys provides non-standard, small and simple sequential buffered file I/O handling.

The buffer size (i.e. the amount of information juggled around when reading from or writing to disk files), is fixed and represented by the constant bufferSize. To alter this size, just change that constant and recompile the implementation module.

There are procedures to execute most sequential file related CP/M commands. Errors are checked whenever a CP/M operation occurs. The result of each CP/M operation is stored in result. With its aid, you can check whether an operation was successful or not.

Additionally, there are checks implemented by the procedure TestAbort; these tests lead to error messages of the form

---- cannot <action> file "D:FILENAME.TYP"

and immediate program abortion. <action> can be either 'open', 'write to', 'rename', or 'close'. The drive is only shown if it isn't the default drive.

2. File Names

The standard FCBFileName type is used to represent file names. To replace a file name parameter by a string constant giving the file name, you have to obey these rules:

- The string constant may contain uppercase letters only.
- The format is fixed: one char for the drive, eight characters for the name and finally three characters for the type.
- No separators between drive and name or name and type are allowed.

So, the string constant has to consist of 12 characters. Replace the file name parameter by

```
FCBFileName('DFILENAMETYP')
```

This is one of the uses of the 'wild' type conversions allowed by Modula-2.

To convert an arbitrary string to the required format, use the procedures offered by the FileNames module.

3. Search Paths

There is a simple scheme to allow search paths during file opening operations. The search path can be set by calling SetSearchPath with a path consisting of one to six letters out of '@' .. 'P'. In FileSys' initialization, it is set to the default drive (indicated by '@').

This search path is used in OpenFile and UsesFile operations, if the file name's drive set to the default drive. Otherwise, the indicated drive is used.

In CreateFile and MakeFile, the first drive in the path indicates the location where the file is being created, again, if no explicit drive has been specified.

4. File Variables

Files are allocated on the heap. The heap space of an unused file may be reclaimed by means of the DisposeFile procedure. Only DeleteFile automatically reclaims the space used by a file, because this is the only action that makes the file descriptor non-meaningful by killing its associated file. In all other actions, the file could be used for further purposes. Therefore, if you don't need a file any longer, you have to get rid of its descriptor by calling DisposeFile.

5. Reading From Files

A typical sequence of commands to read from a file is:

```
SetSearchPath(path);
UsesFile(file, fileName);

WHILE NOT EOF(file) DO
  ReadByte(file, char);
END;

DisposeFile(file);
```

The UsesFile procedure is equivalent to an AssignName, OpenFile sequence, except for the fact that UsesFile aborts (HALTs) the program if the file cannot be found. By using the AssignName, OpenFile sequence, you can check the operation's success by looking at the result value.

The path indicates up to six drives on which the file to open is searched for.

6. Writing to Files

A typical sequence of commands to write to a file is:

```
SetSearchPath(path);
MakeFile(file, fileName);

REPEAT
  WriteByte(file, char);
UNTIL dataExhausted;

CloseFile(file);
DisposeFile(file);
```

Here, the first drive in path indicates where the new file is created. If a file with the same name exists on the indicated drive, it is deleted first.

CloseFile records the file permanently. DisposeFile returns the space used by the file's descriptor to the storage management.

7. Renaming Files

To rename a file you work(ed) with, use the following command:

```
RenameFile(file, newName);
```

If you want to rename a file without working with it, use the Rename procedure:

```
Rename(oldName, newName);
```

Rename uses a local file for its operation. This file is deleted upon completion of the action. It is a good idea to assure a file's existence before renaming it because these procedures abort your program if the file cannot be renamed (i.e. not found).

8. Deleting Files

Like renaming operations, deletion can be done by name or by using a file variable.

The DeleteFile as well as the Delete procedure dispose automatically of the file descriptor in any case. No test on success of the operation is done. You can check this by testing result.

Call the procedures as follows:

```
DeleteFile(file);
```

or

```
Delete(name);
```

9. A Sample File Copy Program

A simple file copy program might look like:

```
MODULE CopyFiles;

  FROM FileSys IMPORT
    File, PathType,
    result,
    SetSearchPath, MakeFile, UsesFile, ReadByte, WriteByte,
    CloseFile, DisposeFile;

  FROM FileNames  IMPORT NameState, StrToFCB;
  FROM OpSys      IMPORT FCBFileName;
  FROM Terminal   IMPORT ReadString, Write, WriteLn, WriteString;

  VAR
    input, output: File;
    ch: CHAR;
    name: ARRAY [0..13] OF CHAR; (* "A:FILENAME.TYP" *)
    fname: FCBFileName;


  PROCEDURE AskFor(prompt: ARRAY OF CHAR);
  VAR
    i: INTEGER;
  BEGIN
    LOOP
      WriteLn;
      WriteString(prompt); ReadString(name);
      fname := ''; (* no default *)
      state := StrToFCB(name, fname);
      IF state = NameOK THEN
        EXIT;
      ELSE
        WriteLn; WriteString('---- ');
        CASE state OF
          WildOK: WriteString('No Wildcards (?,*) Allowed');
        | DeviceOK: WriteString('No Devices Possible');
        | DriveError: WriteString('Drive has to be in @, A..P');
        | NoNameDefault: WriteString('Please Specify a File Name');
        | IllegalChars: WritString('Illegal Chars in Name');
        END;
      END;
    END;
  END AskFor;


BEGIN
  WriteString('CopyFiles -- simple file copy program'); WriteLn;
  WriteLn;
  WriteString('Enter file names as [drive:] filename.typ'); WriteLn;
  AskFor('source file name:      ');
  UsesFile(input, fname);
  AskFor('destination file name: ');
  MakeFile(output, fname);

  LOOP                 (* ok, copy. *)
    ReadByte(input, ch); IF result # 0 THEN EXIT; END;
    WriteByte(output, ch); IF result # 0 THEN EXIT; END;
  END; (* LOOP *)

  CloseFile(output);
  DisposeFile(input);
  WriteString('done');
END Copy.
```

File name expansion and tests for the existence of the input file can be built into the AskFor procedure by using the procedures of the FileNames module and OpenFile.

10. The Interface

```
DEFINITION MODULE FileSys;

  FROM OpSys IMPORT FCBFileName;

  EXPORT QUALIFIED
        File, PathType,
        result,
        MakeFile, UsesFile, SetSearchPath, AssignName,
        CreateFile, OpenFile,
        ReadByte, EOF, WriteByte,
        CloseFile, DisposeFile, Drive,
        Delete, DeleteFile, Rename, RenameFile;

  CONST
    searchPathLength = 6;             (* max. 6 drives to search *)

  TYPE
    File;
    PathType = ARRAY [0..searchPathLength-1] OF CHAR;

  VAR
    result: CARDINAL;   (* CP/M return value. *)

  PROCEDURE SetSearchPath(newSearchPath: PathType);
  PROCEDURE AssignName(VAR f: File; fname: FCBFileName);
  PROCEDURE CreateFile(f: File);
  PROCEDURE OpenFile(f: File);
  PROCEDURE CloseFile(f: File);
  PROCEDURE Delete(delfil: FCBFileName);
  PROCEDURE DeleteFile(VAR f: File);
  PROCEDURE Rename(old, new: FCBFileName);
  PROCEDURE RenameFile(oldFile: File; newName: FCBFileName);
  PROCEDURE ReadByte(f: File; VAR ch: CHAR);
  PROCEDURE EOF(f: File): BOOLEAN;
  PROCEDURE WriteByte(f: File; b: CHAR);
  PROCEDURE MakeFile(VAR f: File; n: FCBFileName);
  PROCEDURE UsesFile(VAR f: File; n: FCBFileName);
  PROCEDURE DisposeFile(VAR f: File);

END FileSys.
```

Section 12. Files

1. General Description

This is the third File System included with this Modula-2 System. It is capable of file positioning at the byte and record-level and therefore provides some previously unavailable features.

This file system uses internal buffers of 1kByte in the current implementation. As with most other modules, you can change this size to customize the system to your needs.

Its advantage over most other random I/O implementations lies in the fact that it is able to position a file at the byte level, ignoring CP/M's 128 byte sectors. The price for this flexibility is an increased complexity and increased size, but it is very simple to operate by the user.

It is also remarkable to say that you can read and write to a file without closing and re-opening it. In fact, you can sequentially read a character, write the next, read the third, etc.

NOTE - Upon each read action, a file's internal buffer gets flushed to disk, if it has been written to since the last read operation.

2. Error Handling

Like in SeqIO, there are two ways of error checking possible: Explicit error checking by looking at FileStatus, and implicit checking by installing a FileHandler procedure.

WARNING - As in SeqIO, handlers shouldn't try to operate on the erroneous file except for closing it or writing a message. Otherwise, subsequent errors may re-invoke the handler.

3. File Names

File names are compatible with the FileNames Module, i.e. they are internally checked by the StrToFCB procedure.

4. File Variables

The FILE type is hidden from the user. To make a file acessible, you have either to open or to create it using the Open and Create procedures. To end a file's processing, you have to Close it. If you want that the file also gets deleted from the disk, use the Release procedure instead. This is especially useful for locally used files.

A normal CP/M 2.2 file may contain up to 65536 records (2^{16}). This results in 8 MBytes maximum file size. Files is laid out to allow for this maximum number of records. For large CP/M 3.0 files (up to 32 MBytes, and 2^{18} records (262'144)), Files does not work correctly. The record calculation procedure used by it, however, calculates 24 bit quantities.

5. File Position Variables

FilePos variables contain all the information necessary to position a file to a given point. A position can either be calculated using CalcPos or it can be retrieved by calling GetPos or GetEOF.

The only way to position a file is given by the SetPos procedure.

NOTE - Users of Volition System's implementation may notice the absence of the SetEOF procedure: Only CP/M 3.x does allow for file truncation. Since CP/M Plus isn't very widespread yet, such a procedure has not been included in the normal form of this module.

6. Reading From Files

Although Files supports random access, a file is usually accessed sequentially. So, a read session may look as follows:

```
VAR
  f         : FILE;
  fileName  : ARRAY [0..13] OF CHAR;
  msg       : ARRAY [0..79] OF CHAR;
  inputState: FileState;
  ch        : CHAR;
BEGIN
  inputState := Open(f, fileName);

  IF inputState = FileOK THEN

    WHILE NOT EOF(f) DO
      Read(f, ch);
    END;

    inputState := Close(f);
    IF inputState # FileOK THEN
      StatusMsg(inputState);
      HALT;
    END;

  ELSE
    StatusMsg(inputState);
  END;
```

NOTE - Even when a file is only read from, a call to Close is mandatory if you want to reclaim the heap space used by the file descriptor.

Multibyte-records or arrays may be read using the ReadRec or ReadBytes procedures. You pass the variable itself or its address and its size, respectively. ReadRec returns an EndError if there wasn't enough data between the start position and the end of file. ReadBytes returns the number of bytes that where actually read from the file.

7. Writing to Files

A typical write command sequence skeleton for sequential access is:

```
outputState := Create(f, fileName);

IF outputState = FileOK THEN

  REPEAT
    Write(f, ch);
  UNTIL allWritten;

  outputState := Close(f);

  IF outputState # FileOK THEN
    StatusMsg(outputState);
  END;

ELSE
  StatusMsg(outputState);
END;
```

Multibyte-records or arrays may be written using the WriteRec or WriteBytes procedure. You pass the address of your structure, and its size. WriteRec returns a StatusError if it couldn't write the full record to disk, WriteBytes returns the number of bytes that where actually written to the file.

8. Positioning Files

Files introduces variables of type FilePos. Their structure is unimportant to the user.

NOTE - FilePos typed variables have to be initialized before they are used by invoking InitPos, and if they are declared local to a procedure, you have to dispose of them by FiniPos to reclaim the dynamic variable storage (heap) they use. This is in contrast to other nearly equivalent implementations (Volition Systems).

It is assumed that a file contains a given number of **fixed size records.** To access a given record, you can calculate its position by using CalcPos. CalcPos uses the record number and the record size which you provide, and calculates thereof the CP/M logical record number and the offset of the start of your logical record, in this record. Having calculated this position, you set the file read start to it by using SetPos. So, the scheme looks like:

```
CalcPos(recordNumber, SIZE(record), pos);
SetPos(f, pos);
```

An often used file organization is to have a file header and behind it records of some given structure. To support this scheme, the AddPos procedure has been implemented. So, you can add the offset of the header to each calculated position. The modified scheme of address calculation is:

```
CalcPos(1, headerSize, firstRec);
  ...
CalcPos(recordNumber, SIZE(record), pos);
AddPos(pos, firstRec, pos);
SetPos(f, pos);
```

If you want to append data to the end of a file, use the GetEOF procedure to find the end position of the file.

NOTE - GetEOF cannot account for your logical records. If your file doesn't end at a CP/M sector end, you have to know where the exact end of file is, either by a mark or by recalculating a record position.

NOTE - Setting the position of a file causes no immediate disk access. This access is delayed until the next read or write operation occurs. This means that positioning a file multiple times between read or write operations doesn't make your computer "fiddle around" on the disks.

9. Renaming Files

The renaming operation works straight forward. You give the old and the new file name as strings, and the rename operation is carried out. Success of operation can be determined by watching Rename's return value.

```
IF Rename('XX.OLD', 'YY.NEW') # FileOK THEN
  WriteString('Error in Renaming XX.OLD');
END;
```

10. Deleting Files

You can delete files by using unambigous or ambigous file names. As renaming, deletion is done by name, directly.

```
IF Delete('*.BAS') = FileOK THEN
  WriteString('BASIC files emptied');
END;
```

11. The Interface

```
DEFINITION MODULE Files;

  FROM SYSTEM IMPORT ADDRESS;

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


  TYPE
    FILE;

    FilePos;

    FileState = (   FileOK, UseError, StatusError,
                    DeviceError, EndError);
    FileHandler = PROCEDURE(VAR FILE);

  PROCEDURE EOF(f: FILE): BOOLEAN;
  PROCEDURE FileStatus(f: FILE): FileState;
  PROCEDURE DummyHandler(VAR f: FILE);
  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(currentName, newName: ARRAY OF CHAR): FileState;

  PROCEDURE InitPos(VAR pos: FilePos);
  PROCEDURE FiniPos(VAR pos: FilePos);

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

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

  PROCEDURE CalcPos(recNum, recSize: CARDINAL; VAR pos: FilePos);
  PROCEDURE AddPos(position, offset: FilePos; VAR pos: FilePos);

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

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

END Files.
```

Section 13. Moves

1. General Description

Moves contains procedures that do fast memory transfers or initializations. It is useful for long initializations, or for fast transfers of large areas.

To use these procedures, you have to import ADR and eventually also SIZE or TSIZE from SYSTEM.

Some examples:

```
Fill(ADR(buffer), SIZE(buffer), 0C); (* initialize buffer to 0 *)

MoveLeft(ADR(string[1]), ADR(string[0]), SIZE(string) - 1);

MoveRight(ADR(string[0]), ADR(string[1]), SIZE(string) - 1);
```

2. How to Use MoveLeft and MoveRight

WARNING - There are some conditions to watch for when moving overlapping areas, as string in the above examples. If the source starts before the destination, use MoveRight, if it starts after the destination, use Move-Left. Otherwise, you just fill your area with the contents of the memory cells that lie between source and destination start or end. In fact, this is the trick that is used to speed up Fill.

The above move examples look as follows[1]:

```
before                         action          after
-------------------------------------------------------------------
"H i   T h e r e . "           MoveLeft        "i   T h e r e . . "
 ^ ^
 | source start                source: s[1]
 destination start             dest:   s[0]
                               length: 8


"H i   T h e r e . "           MoveLeft        "H H H H H H H H H "
 ^ ^
 | destination start           source: s[0]
 source start                  dest:   s[1]
                               length: 8


"H i   T h e r e . "           MoveRight       ". . . . . . . . . "
 ^ ^
 | source start                source: s[1]
 destination start             dest:   s[0]
                               length: 8


"H i   T h e r e . "           MoveRight       "H H i   T h e r e "
 ^ ^
 | destination start           source: s[0]
 source start                  dest:   s[1]
                               length: 8
```

[1] For better legibility of the markers, the string contents are double spaced.

NOTE - This module is machine dependent; restrict its use as far as possible.

3. The Interface

```
DEFINITION MODULE Moves;

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED
        MoveLeft, MoveRight, Fill;

  PROCEDURE MoveLeft(source, dest: ADDRESS; length: CARDINAL);
  PROCEDURE MoveRight(source, dest: ADDRESS; length: CARDINAL);
  PROCEDURE Fill(start: ADDRESS; length: CARDINAL; ch: CHAR);

END Moves.
```

Section 14. OpSys

1. General Description

The OpSys module provides the interface to the operating system, i.e. CP/M. It includes procedures for standard BDOS - as well as BIOS calls. For convenience, enumeration types giving mnemonic names to the BDOS respectively BIOS function numbers are included; these can be used if you like it that way, but you are free to use any numbers as arguments in the procedures because they have WORD parameters all over.

Furthermore, the CP/M data structures FCB (File Control Block), FCBFileName (part of the FCB) and the string input buffer CPMStringBuffer as well as the BIOS-related DPH (disk parameter header), DPB (disk parameter block), DirEntry and DirBuffer are exported by this module. The last two items are used by the search first/search next functions of CP/M.

WARNING - This module provides you full control over the operating system. Though, you can cope around with your system at the lowest levels. You can use its power to your advantage, but it is possible to create a medium to king size disaster. So be careful, especially when using disk oriented functions of either the BDOS or the BIOS.

For more information about BDOS- and BIOS-calls, please read **The CP/M Interface Guide** or **The CP/M Alteration Guide** from Digital Research, respectively. If you use another operating system than CP/M 2.2, consult the equivalents to the above manuals for your system.

2. The Interface

```
DEFINITION MODULE OpSys;

  FROM SYSTEM IMPORT
    ADDRESS, WORD;

  EXPORT QUALIFIED
    FCBFileName,
    FCB, RndFCB,
    CPMStringBuffer,
    BdosFunctions,
    Bdos, BdosHL,
    DirEntry, DirBuf, DirBufPtr,
    DPB, DPBptr, DPH, DPHptr,
    BiosFunctions,
    Bios, BiosHL;

(*   WARNING - this module is a potential danger to your system if you      *)
(*         don't now exactly what you're doing. So, if you don't have any    *)
(*         experience,                                                       *)
(*                                                                           *)
(*             K E E P   Y O U R   F I N G E R S   O F F   ! ! !             *)
(*                                                                           *)
(*                                 Thank You.                                *)

  TYPE
    FCBFileName = RECORD
      CASE BOOLEAN OF
         TRUE: disk  : CHAR;
               name : ARRAY [0..7] OF CHAR;
               type  : ARRAY [0..2] OF CHAR;
       | FALSE:text  : ARRAY [0..11] OF CHAR;
      END;
    END;

    FCB = RECORD
      name: FCBFileName;
      CASE BOOLEAN OF
        TRUE:  rest  : ARRAY [0..20] OF CHAR;
       |
         FALSE: ex,                             (* extent *)
                s1,s2,                          (* system data *)
                rc   : CHAR;                    (* record count *)
                d    : ARRAY [0..15] OF CHAR;
                cr   : CHAR;                    (* current record *)
      END;
  END; (* FCB *)

  RndFCB = RECORD
    name: FCBFileName;
    CASE BOOLEAN OF
      TRUE:  rest  : ARRAY [0..23] OF CHAR;
     |
      FALSE: ex,                                (* extent *)
             s1,s2,                             (* system data *)
             rc    : CHAR;                      (* record count *)
             d     : ARRAY [0..15] OF CHAR;
             cr    : CHAR;                      (* current record *)
             rec   : CARDINAL;                  (* rnd rec # *)
             r2    : CHAR;                      (* rnd rec, hi part, ev. ovfl *)
    END;
  END; (* RndFCB *)

  CPMStringBuffer = RECORD
    maxLen : CHAR;
    curLen : CHAR;
    text   : ARRAY[0..255] OF CHAR;
           END;


TYPE
  BdosFunctions = (boot, crtIn, crtOut, rdrIn, punOut,
                   lstOut, dirIO, getIOB, setIOB, prtStr,
                   rdCBuf, crtSt, verNo, reset, selDsk,
                   openF, closeF, searchFst, searchNxt, deleteF,
                   readSeq, writeSeq, makeF, renameF, getLogin,
                   retCDsk, setDMA, getAlloc, writeProt, getRO,
                   setFA, getDPB, user, readRan, writeRan,
                   compFSize, setRec, resDrv, dummy1, dummy2,
                   writeRanZF);


PROCEDURE Bdos(func, parm: WORD; VAR return: WORD);
PROCEDURE BdosHL(func, parm: WORD; VAR return: WORD);
```

```
  TYPE
    DirEntry   = RECORD
      name    : FCBFileName;
      ex,
      s1,s2,
      rc      : CHAR;
      d       : ARRAY [0..15] OF CHAR;
    END;

    DirBufPtr = POINTER TO DirBuf;
    DirBuf = ARRAY [0..3] OF DirEntry;

    DPBptr = POINTER TO DPB;
    DPB = RECORD
      spt  : CARDINAL;
      bsh,
      blm,
      exm : CHAR;
      dsm : CARDINAL;
      drm : CARDINAL;
      al0,
      al1  : CHAR;
      cks,
      off  : CARDINAL;
    END;

    DPHptr = POINTER TO DPH;
    DPH = RECORD
      xlt        : ADDRESS;
      s0,s1,s2   : CARDINAL;
      dirbuf     : DirBufPtr;
      dpb        : DPBptr;
      csv        : ADDRESS;
      alv        : ADDRESS;
    END;

    BiosFunctions = (wBoot, conSt, conIn, conOut, list, punch,
                     reader, home, dskSel, trkSet, secSet, DmaSet,
                     read, write, listSt, tranSec);


  PROCEDURE Bios(routine, parm: WORD; VAR return: WORD);
  PROCEDURE BiosHL(routine, parmBC, parmDE: WORD; VAR return: WORD);

END OpSys.
```