

**Cambridge Microprocessor  
Systems Limited  
Micro-Module  
Modula-2**

Publication Date : April 1996

© Cambridge Microprocessor Systems Ltd 1996

The information contained in this document has been thoroughly checked and is believed to be factually correct. However Cambridge Microprocessor Systems Ltd. assumes no responsibility for any mistakes, omissions, or inaccuracies.

Cambridge Microprocessor Systems do not accept any liability out of the use of this or any other Cambridge Microprocessor Systems product.

Although Cambridge Microprocessor Systems Ltd. is committed to supplying its customers a standard, well tested product, we reserve the right to make modifications to the product described in this manual.

This page intentionally left blank

---

## Table Of Contents

### Getting Started

1.1	Introduction . . . . .	1-1
1.2	Installation . . . . .	1-1
1.3	Getting Started . . . . .	1-2
1.4	'Hello World' . . . . .	1-4
1.5	Multi-tasking . . . . .	1-6
1.6	PCs Directory Paths . . . . .	1-7
1.7	The 'Pop Up' Terminal . . . . .	1-8
1.7.1	Executing . . . . .	1-8
1.7.2	Back and forth . . . . .	1-8
1.7.3	Function Keys . . . . .	1-9
1.7.4	Terminal Screen Commands . . . . .	1-9
1.8	Using a Word processor . . . . .	1-9
1.8.1	Down Loading . . . . .	1-10
1.8.2	Module Libraries . . . . .	1-10
1.8.3	Designing a Program . . . . .	1-12
1.9	MODULEs . . . . .	1-12
1.9.1	C . . . . .	1-13
1.9.2	Save . . . . .	1-14
1.9.3	Load . . . . .	1-14
1.9.4	M2 . . . . .	1-14
1.9.5	Mdir . . . . .	1-15
1.10	Immediate mode . . . . .	1-15
1.11	Debug . . . . .	1-17
1.12	Trace . . . . .	1-17

1.13	The semi-colon . . . . .	1-18
1.14	Libraries . . . . .	1-18
1.15	Batch Files . . . . .	1-18
1.16	Hardware Description . . . . .	1-19
1.16.1	Digital I/O . . . . .	1-19
1.16.2	Serial Port . . . . .	1-20
1.16.3	The Terminal . . . . .	1-21
1.16.4	The Reset Link . . . . .	1-22
1.16.5	Expansion Connector . . . . .	1-22
1.17	Terminal Escape Sequences . . . . .	1-22
1.17.1	Colour . . . . .	1-23
1.17.2	Background Colour . . . . .	1-23
1.17.3	Clear Screen . . . . .	1-23
1.17.4	Cursor Move . . . . .	1-23
1.17.5	Define a Window . . . . .	1-24
1.17.6	Enter a Window . . . . .	1-24
1.17.7	Hide a Window . . . . .	1-24
1.17.8	Display this Window on Top . . . . .	1-24
1.17.9	Insert a Line . . . . .	1-24
1.17.10	Delete a Line . . . . .	1-24
1.17.11	Clear to End of Line . . . . .	1-25
1.17.12	Cursor on/off . . . . .	1-25

**The Modula-2 Programming Language**

2.1	Introduction . . . . .	2-1
2.2	History . . . . .	2-2
2.3	Other Languages . . . . .	2-2

---

2.3.1	Pascal . . . . .	2-2
2.3.2	'C' . . . . .	2-3
2.3.3	BASIC . . . . .	2-3
2.4	About this Section . . . . .	2-4
2.5	Versions . . . . .	2-4
2.6	A Simple Program . . . . .	2-5

### **Syntax and Conventions**

3.1	Identifiers . . . . .	3-1
3.2	Case Sensitivity . . . . .	3-1
3.2.1	Variables and Constants . . . . .	3-1
3.2.2	Procedure names . . . . .	3-1
3.2.3	Types . . . . .	3-2
3.2.4	Reserved words . . . . .	3-2
3.3	Standard Modula-2 reserved Words . . . . .	3-2
3.4	Separators . . . . .	3-3
3.5	The Semi-colon . . . . .	3-3
3.6	Comments . . . . .	3-4

### **Modules**

4.1	Programs . . . . .	4-1
4.2	Libraries . . . . .	4-2
4.2.1	Qualified Identifiers . . . . .	4-5
4.2.2	Unqualified Identifiers . . . . .	4-5
4.3	Local . . . . .	4-6
4.3.1	Export . . . . .	4-6
4.4	Version Control . . . . .	4-6
4.5	Battery backed RAM . . . . .	4-7

**Errors**

5.1 Error Types . . . . . 5-1

**Modula-2**

6.1 Data Types . . . . . 6-1

6.1.1 Simple Types . . . . . 6-1

6.1.2 Structured Types . . . . . 6-13

6.2 Type Compatibility . . . . . 6-14

6.2.1 Expression . . . . . 6-15

6.2.2 Assignment . . . . . 6-15

6.2.3 Parameter . . . . . 6-16

6.3 Type Conversion . . . . . 6-17

6.4 Declarations . . . . . 6-18

6.5 Control Structures: Selection . . . . . 6-19

6.5.1 IF THEN ELSIF ELSE END . . . . . 6-19

6.5.2 CASE OF | END . . . . . 6-20

6.6 Control Structures: Iteration . . . . . 6-21

6.6.1 FOR TO BY DO END . . . . . 6-21

6.6.2 REPEAT UNTIL . . . . . 6-22

6.6.3 WHILE DO END . . . . . 6-22

6.6.4 LOOP EXIT END . . . . . 6-23

6.7 PROCEDURES . . . . . 6-23

6.7.1 Visibility . . . . . 6-24

6.7.2 Local and Global Variables . . . . . 6-24

6.7.3 Parameters . . . . . 6-25

6.7.4 Function PROCEDURES . . . . . 6-26

6.7.5 Recursion . . . . . 6-27

---

6.7.6	BYTE and WORD parameters . . . . .	6-28
-------	------------------------------------	------

**Language Summary**

7.1	Key Words . . . . .	7-1
7.1.1	ABS . . . . .	7-1
7.1.2	ADR . . . . .	7-2
7.1.3	ADDRESS . . . . .	7-2
7.1.4	AND or & . . . . .	7-3
7.1.5	ARRAY . . . . .	7-3
7.1.6	BEGIN . . . . .	7-4
7.1.7	BITSET . . . . .	7-4
7.1.8	BOOLEAN . . . . .	7-5
7.1.9	BY . . . . .	7-5
7.1.10	BYTE . . . . .	7-6
7.1.11	CAP . . . . .	7-6
7.1.12	CARDINAL . . . . .	7-7
7.1.13	CASE . . . . .	7-7
7.1.14	CHAR . . . . .	7-8
7.1.15	CHR . . . . .	7-8
7.1.16	CONST . . . . .	7-9
7.1.17	DEC . . . . .	7-9
7.1.18	DEFINITION . . . . .	7-10
7.1.19	DIV . . . . .	7-10
7.1.20	DO . . . . .	7-11
7.1.21	ELSE . . . . .	7-11
7.1.22	ELSIF . . . . .	7-12
7.1.23	END . . . . .	7-12

7.1.24	EXCL	7-13
7.1.25	EXIT	7-13
7.1.26	FALSE	7-14
7.1.27	FLOAT	7-14
7.1.28	FOR	7-14
7.1.29	FROM	7-15
7.1.30	HALT	7-15
7.1.31	HIGH	7-16
7.1.32	IF	7-16
7.1.33	IMPLEMENTATION	7-17
7.1.34	IMPORT	7-17
7.1.35	IN	7-18
7.1.36	INC	7-19
7.1.37	INCL	7-19
7.1.38	INTEGER	7-19
7.1.39	LOOP	7-20
7.1.40	MAX	7-20
7.1.41	MIN	7-20
7.1.42	MOD	7-21
7.1.43	MODULE	7-22
7.1.44	NIL	7-22
7.1.45	NOT or ~	7-23
7.1.46	ODD	7-23
7.1.47	OF	7-24
7.1.48	OR	7-24
7.1.49	ORD	7-24

---

7.1.50	POINTER	7-25
7.1.51	PROCEDURE	7-25
7.1.52	REAL	7-27
7.1.53	REPEAT	7-27
7.1.54	RETURN	7-28
7.1.55	SIZE	7-29
7.1.56	THEN	7-29
7.1.57	TO	7-30
7.1.58	TRUE	7-30
7.1.59	TRUNC	7-30
7.1.60	TYPE	7-31
7.1.61	UNTIL	7-31
7.1.62	VAR	7-32
7.1.63	WHILE	7-32
7.1.64	WORD	7-33
7.2	Operators	7-34
7.2.1	Precedence	7-34
7.2.2	Unary	7-34
7.2.3	Binary	7-35
7.2.4	Set	7-36
7.3	Punctuation	7-38
7.4	CMS Extensions	7-40
7.4.1	Indirection	7-40
7.4.2	Bit Shift	7-40
7.5	System Key Words	7-41
7.5.1	BatRam	7-41

7.5.2	BatRamTop	7-41
7.5.3	BufSize	7-41
7.5.4	C	7-41
7.5.5	Continue	7-42
7.5.6	Debug	7-42
7.5.7	Eor	7-43
7.5.8	ErrMax	7-43
7.5.9	Escape	7-43
7.5.10	Even	7-44
7.5.11	Heap	7-44
7.5.12	HeapTop	7-44
7.5.13	Help	7-44
7.5.14	List	7-44
7.5.15	Load	7-45
7.5.16	Lock	7-46
7.5.17	Mdir	7-46
7.5.18	M2	7-47
7.5.19	New	7-47
7.5.20	Program	7-47
7.5.21	Quit	7-47
7.5.22	Rnd	7-48
7.5.23	Save	7-48
7.5.24	Sgn	7-49
7.5.25	Stack	7-49
7.5.26	StackTop	7-49
7.5.27	StackSize	7-50

---

7.5.28	StartProcess . . . . .	7-50
7.5.29	Symbol . . . . .	7-50
7.5.30	Trace . . . . .	7-51
7.5.31	TurnKey . . . . .	7-51
7.5.32	UnLock . . . . .	7-52

**Co-routines**

8.1	PROCEDURES . . . . .	8-1
8.2	Built in functions . . . . .	8-1
8.3	MODULEs . . . . .	8-2
8.4	Machine Code . . . . .	8-2

**Assembly Code**

9.1	Introduction . . . . .	9-1
9.2	File Search Order . . . . .	9-2
9.3	Implementation . . . . .	9-2
9.3.1	Definition File . . . . .	9-2
9.3.2	Assembly File . . . . .	9-3
9.3.3	Assembly Only Programs . . . . .	9-4
9.4	Parameter passing . . . . .	9-4
9.4.1	Heap Pointer . . . . .	9-5
9.4.2	Parameter Pointer . . . . .	9-6
9.4.3	Type storage requirements . . . . .	9-8
9.5	Debugging . . . . .	9-11

**The Debugger**

10.1	Introduction . . . . .	10-1
10.2	Using the Debugger . . . . .	10-1
10.3	Abbreviating Commands . . . . .	10-2

10.4	Data Sizes . . . . .	10-3
10.5	Volatile Registers . . . . .	10-3
10.6	Odd And Even Addresses . . . . .	10-3
10.7	Options . . . . .	10-4
10.8	Debugger Command Summary . . . . .	10-5
10.8.1	BREAK . . . . .	10-5
10.8.2	BYTE . . . . .	10-6
10.8.3	CHANGE . . . . .	10-7
10.8.4	COMPARE . . . . .	10-8
10.8.5	DISASSEMBLE . . . . .	10-8
10.8.6	DUMP . . . . .	10-9
10.8.7	FILL . . . . .	10-10
10.8.8	FIND . . . . .	10-10
10.8.9	GOTO . . . . .	10-11
10.8.10	HELP . . . . .	10-11
10.8.11	KILL . . . . .	10-12
10.8.12	LONG . . . . .	10-12
10.8.13	MOVE . . . . .	10-13
10.8.14	NOREAD . . . . .	10-13
10.8.15	ODD . . . . .	10-14
10.8.16	READ . . . . .	10-14
10.8.17	REGISTER . . . . .	10-15
10.8.18	TEST . . . . .	10-16
10.8.19	TRACE . . . . .	10-17
10.8.20	WORD . . . . .	10-18
10.9	Option Summary . . . . .	10-18

**Appendix**

11.1 ASCII Character Codes . . . . . 11-1  
11.2 References . . . . . 11-2

This page is intentionally left blank

# 1 Getting Started

## 1.1 Introduction

This section of the manual describes the setting up and use of the Micro-Module. Emphasis is on getting going, allowing the system to be used as quickly and as simply as possible. Teaching is by example and one should follow through this manual and its programs hand in hand. There is an extensive printout of several larger example programs which are presented to give an idea of the use of programming in Modula-2.

A more detailed look at Modula-2, descriptions of the key words and I/O libraries can be found in later sections of this manual. The manual is not a language tutorial and the teachings of how to program in Modula-2 should be obtained from another text book. It is in fact easy to program most common requirements. Finally there is an operating system section and a hardware section which is written to help those requirements where machine code is required.

Throughout this manual and its associated manuals the following should be observed:

- i. Modula is used to represent CMS Modula-2
- ii. Carriage returns are assumed as line endings
- iii. *This type style is used for keyboard input*
- iv. Modula is case sensitive
- v. Pressing the Esc key will terminate programs.

## 1.2 Installation

To install the software from the 3.5 inch floppy disc provided, your computer should be an IBM PC compatible with MSDOS 2.2 or above with a hard drive on 'C' and a 3.5 inch floppy drive. If you can not use the disks provided please contact your distributor who will be happy to provide disks in the required P.C. format. If your system is different to this you can still use the software directly from the installation disc. Do not install the disc on a system without a hard disc.

Two things need to be done before starting. Firstly to install the PC software. The discs provided are a 3.5 inch 1.44M Double Sided High Density in standard PC format.

Insert this into the drive and type as follows:

*a: or b:*

as required

*\install*

The disc will make a new directory named 'C:\UMODM2'. Into this will be copied several demonstration files including the ones listed in this manual. These are pure text files containing no text formatting instructions.

### **1.3 Getting Started**

Before we go any further lets make the Micro-Module

do something. First enter the UMODM2 directory

*c:*

*cd\UMODM2*

Inside this directory is the terminal program 'TARGET' and several Modula example programs. Access the directory to see the files.

*dir /w*

Now invoke the terminal:

*TARGET*

A help screen will appear showing that the terminal has been loaded. This shows the current setting of the soft keys and other set up information.

More details can be found in the section on the terminal. The terminal is a terminate and stay resident program (TSR) and will remain in the PCs memory. Hit any key to return to DOS.

To enter the terminal use:

*Alt Z*

Alt F10 will exit the terminal program leaving it resident in memory for future entry.

CTRL F10 will terminate the terminal program and remove it from the PCs memory.

The terminal is now ready for use. Connect the serial lead provided to the serial port on the Micro-Module and the PCs COM1 serial port. Connect the mains adapter to the screw terminal on the Micro-Module checking that the **RED wire goes to +5V** and the **BLACK wire goes to 0V**. The green LED will come on when the power is applied to the Micro-Module. If all is well the start up message will appear on the PCs screen after a delay of about 15 seconds followed by a prompt. This delay occurs while the operating system checks all programs in RAM and EPROM to make sure that they are valid programs have have not been corrupted.

M >

If nothing appears on the screen first check that the serial cable is secure and that the green LED comes on with the power. The reset link can be used as an equivalent to power on. The serial port is 9600 Baud, 8 data bits, 1 start bit, 1 stop bit and no parity. The terminal program sets itself up with the same specification.

M> *NoDog*

The Micro-Module has a software watchdog running which will reset the processor every 34 seconds if the watchdog is not toggled. In order to turn the watchdog off there is a routine in the EPROM called NoDog. This will prevent the Micro-Module from resetting every 34 seconds.

## 1.4 'Hello World'

Having now connected up the system and installed the software we will try to write our first program on the Micro-Module. We are going to type commands directly from the keyboard. First we need to get these commands into the Micro-Module. Function key F1 has been shipped set up to do this automatically:

F1

Try the command `WriteLn` (a library word). This will write a new line.

```
M> WriteLn
M>
```

Now lets try two library words. Note that a semi-colon separates the words.

```
M> WriteString('Hello World'); WriteLn
Hello World
```

Here are some other examples of writing:

```
M> WriteInt(2 + 3 * 4 , 0); WriteLn
14
```

Note the precedence of the operators. The second parameter is the write field width. Try it with other values.

```
M> WriteHex(-1, 0); WriteLn
FFFFFFFF
```

```
M> VAR x
M> x := 999
M> WriteInt(x+1, 0); WriteLn
1000
```

This is the immediate mode where any command typed at the keyboard is executed immediately. This is very useful in debugging, when a program is stopped, and we wish to examine or change any of its variables.

Now try the following simple program. No semi-colons are required on the line ends when typing directly from the keyboard. They are most definitely required in a text file. If an error is made it will be indicated and you can retype it. ESC will abort a program entry.

```
M> MODULE Test
  FROM Terminal IMPORT WriteString, WriteLn
  BEGIN
    WriteString('Hello World')
    WriteLn
  END
M>
```

This is now stored in memory under the name 'Test'. To execute it just type Test.

```
M> Test
Hello World
M>
```

The system has a convenience feature in that it can list a program stored in memory.

```
M> List Test
MODULE Test;
IMPORT Terminal;
END Test.
M>
```

The listing of a program only displays the IMPORT list. This is a list of other library modules that this program will use. In our Test case, the Library 'Terminal' contains the routines 'WriteString' and 'WriteLn'. Listing a library module shows all of the procedures and variables that are available for use by any program.

Try listing 'Terminal'.

'List' on its own will list out all of the module names.

```
M> List
```

Program Module(s)

Clear            Test

Library Module(s)

.....            Terminal            .....

M>

The word 'New' will clear the memory of all existing programs that are not Locked and are not in permanent EPROM storage (Lock comes later.)

M> *New*

M> *List*

Note that 'Test' has now gone.

Finally the word Help will list out all of the key words in the language. Words in Upper case are standard Modula. Words starting with a capital letter are system words that are provided by ourselves. Double barreled words have two capitals e.g. TurnKey. This unwritten rule should be adhered to when writing your own programs to achieve a degree of portability.

The other words used in programs will be the variable names. These should use lower case throughout.

M> *Help*

## 1.5 Multi-tasking

Lets load another program and run it.

M> *C Demo*

M> *Demo*

This flashes the red LED on the Micro-Module. To stop the program press the any key. Now try:-

M> *Demo &*

M>

This time the prompt returns! The program is running as a second process. Now try:-

```
M> List
```

This is as before, while at the same time the LED continues to flash. Just for fun, try and run a few copies of Demo.

```
M> Demo &  
M> Demo &  
M>
```

To stop the programs type Esc as before and they will all stop.

## 1.6 PCs Directory Paths

Up to now we have been working from the UMODM2 directory. Later we shall be using the PC's word processor. This will be in another directory. It will be convenient to create a path to the word processor. This is done in different ways depending on the word processor in question. It is up to the user how this is arranged. If the word processor is already in the DOS command directory no action need be taken.

To add a path to a directory is a simple matter. This will allow the word processor to be invoked from any other directory, including our UMODM2 directory. In the root directory, usually C:, will be found a text file called 'autoexec.bat'. This needs to be edited to include a path to the directory containing the word processor.

```
PATH C;; .....other paths.....
```

add the new path to the list

```
PATH C;; .....other paths..... ; C:\new_path_name
```

## 1.7 The 'Pop Up' Terminal

The 'Pop Up' terminal has already been used in the previous section. It is so called because it appears to pop up out of any program when invoked by the Alt Z key. It is a terminate and stay resident type of program which stays in the PC's memory ready for use. It completely takes over the PC's serial port. Any previous settings will be lost after its use.

### 1.7.1 Executing

To run the terminal just type its name. It is supplied as standard on the installation disc as a file called 'TARGET'. When used from the execution directory, typing its name loads it into the PC's memory. This does NOT execute the terminal but it remains available for use from anywhere by pressing the Alt Z key.

Alt F10 will exit the terminal and return back to exactly the same place it was called from leaving the terminal in memory.

To safely remove the terminal from memory adopt the following routine. This assumes that we are already in the terminal. If not, miss the first step.

- (a) Alt F10 to exit the terminal
- (b) Get out of any editor or program to the DOS prompt
- (c) Alt Z to enter the terminal
- (d) Ctrl F10 terminate the terminal

### 1.7.2 Back and forth

Using Alt Z and Alt F10 moves backwards and forwards in and out of the terminal. This can be of significant time saving when editing a program. If both the word processor and the terminal can be co-resident, one can jump from one to another at the touch of a key.

### 1.7.3 Function Keys

When executing the terminal, several of the function keys have been set up. For example function key F3 will repeat the last line. Any of the 10 function keys can be made to output a string of characters. This is easily done by pressing the required function key whilst holding down the Shift key.

The screen will now prompt for the line required. The line can now be typed in or the last line edited. Control codes can be entered using the ^ character followed by a capital letter. For example F1 could contain:

```
C Name^MName^M
```

where Name is the current file name. This would Compile 'Name' and then run it.

From an editor the sequence: Alt Z, F1 will leave the editor, enter the terminal, compile the file and then run it. This assumes the file has been previously saved by the editor.

### 1.7.4 Terminal Screen Commands

The terminal program resident in the PC contains many special features. These are accessed by 'Escape sequences'. For example the cursor can be positioned anywhere on the screen, the colours can be changed, windows can be created etc. A full specification is detailed in a later section.

## 1.8 Using a Word processor

Although programs can be written from the keyboard in immediate mode any program requiring more than a few lines should be written with a word processor. Any word processor can be used that produces straight text files. Also there should be sufficient memory available to hold the word processor and the terminal at the same time.

### 1.8.1 Down Loading

Down loading a program is easily achieved from the Modula prompt. Lets try the program 'Tables'.

```
M> C Tables
M> List
M> Tables
```

The file 'Tables' is loaded from the PCs disc down the serial cable, is compiled into fast execution code and stored in the Modules memory. The instruction 'List' shows us that the module 'Tables' is now in the module directory. Note that the file name is the same as the program name. This is an unwritten rule that should be followed for tidiness. All files on the installation disc have .MOD extensions. The 'C' statement will automatically add the extension to the end of the name if no other extension exists.

The name can be a full file path if required. The complete path name for the above example would be:

```
M> C c:\UMODM2\Tables.MOD
M>
```

### 1.8.2 Module Libraries

It may be crossing the minds of ambitious programmers that when the program starts getting very large the process of down loading, at 9600 Baud, might get a little tedious. Very likely. To this effect the Micro-Module can use library modules. Library and program modules can be down loaded as normal and then Locked into RAM. They will remain there until the power is turned off or the programmer 'UnLock's them. Now large programs can be built up from lots of smaller programs.

The program 'Tables' is currently in the modules memory. This can be erased with the key word 'New'.

```
New
List Tables
    ^ Module not found
```

If we want to preserve 'Tables' in the RAM we have to Lock it into the memory.

*C Tables*  
*Lock Tables*  
*List Tables*

the program will be listed

*New*  
*List Tables*

It is still there!

Reset the Micro-Module. It will still be there. Finally any program can be made to run after reset by issuing a 'TurnKey' instruction.

*TurnKey Tables*

Note:

The line after 'TurnKey' will be executed after a reset. Any text can be turnkeyed. If the text is a module name it must have previously been locked. The TurnKey system module produced in the RAM can be Saved and added to the final target EPROMs if required.

Now reset the Micro-Module using the reset link.

The program runs from power on!

*TurnKey*

on its own will cancel the effect. So what about a program that continues to run? Here the 'Esc' key can be used to escape from that program. Pressing it once will terminate the program after the current statement has completed. If a program has disabled the Escape effect, the only way of removing the TurnKey line is to turn the power off.

### 1.8.3 Designing a Program

The above Module concept is in fact a very good way to design programs. The small sections can be tested individually with known input/output data before being committed to their main task in the large program. One big-un is being made from lots of good little-uns.

This concept extends later into the EPROM, where the libraries can be included as user routines in a larger pair of EPROMs.

## 1.9 MODULES

Every program or library whether it be machine code or Modula-2 is saved as a module. This has a header which contains details on the module which includes its length, starting offset and a check sum. On power on or reset the system searches the EPROM and RAM for valid modules. These are entered into the module directory. The word 'List' can be used to list the module directory and will show a list of modules that are provided as standard in the EPROM. Listing a module is possible and this will show the contents of that module. Only library modules will contain information. Enter the terminal as before by typing *TARGET*.

A help screen will now be shown press any key to exit. We are now back in DOS but with the target TSR installed. To enter it type Alt Z and to exit Alt F10.

From the Micro-Module:

M> *List Terminal*

Running a module is achieved by typing its name. There is a program module in the EPROM which can be used to clear the display at any time for example after running a video program.

M> *Clear*

Running a library will do nothing but initialise that library. 'List' shows the programs and libraries separately. Also Libraries can be identified by the word DEFINITION at the start when listed individually. Programs start with MODULE. It will be noticed that the program 'Clear' when listed contains an

IMPORT statement. The program imports 'Video'. This means that it can have access to any of the procedures and variables that appear when 'Video' is listed. When the program 'Clear' runs first it will run 'Video' and 'Video' may run other programs that it imports and so on. This allows library modules the chance to initialise themselves. The order is as described, the bottom of the chain first and the actual program module last. If a module is imported twice it is only run the first time it is encountered.

The software supplied contains the source of all of our libraries for your inspection. It is usually easier to see how a program is written by looking at examples. It is not necessary to do this as only the definition and binary file of a module need to be supplied. The definition being a description of the library and the binary the actual code itself. This can give a strong degree of software protection as the users and ourselves can decide upon exactly how much information the definition file shall hold, giving only the pertinent information required.

During development of programs the following commands will be used:

### 1.9.1 C

Compile a program or library. Programs are stored in the current directory. Libraries are stored in the LIB sub-directory.

M> *C Tables*

This will compile the program Tables. To run it type its name.

M> *Tables*

List will now show Tables as the last entry. During development a program will be compiled several times. The system automatically deletes the last copy and compacts the memory where necessary.

### **1.9.2 Save**

This program can now be saved to disc if required.

M> *Save Tables*

This module will be placed in the REL sub-directory. It is stored in its compiled form.

### **1.9.3 Load**

Compiled programs can be Loaded into memory before running. This is a lot faster than recompiling each time. 'C' compiles a source program from disc to memory. 'Load' loads a previously compiled and saved program from disc to memory. The module in memory will be the same. 'Load' can take multiple names.

### **1.9.4 M2**

The 'M2' command performs a compile followed by a 'Save'. It can have multiple names. This is very useful when recompiling a batch of files after an error has been corrected in a low down library that is used by many higher up modules. This is a point to remember.

If a program is edited and recompiled, any programs that call this module will need to be recompiled as well.

### 1.9.5 Mdir

This command will display all of the system modules that are in the Micro-Module's memory. This includes both EPROM and RAM memory. Each entry is followed by an E or R ( EPROM or RAM ) and a digit which is the system module type:

0	Machine code program
1	Configuration table for a device
2	Driver
3	Special system parts
4	Modula-2 MODULEs
5	Data

### 1.10 Immediate mode

We used the immediate mode when we first printed 'Hello World' earlier. Immediate mode is used when typing statements directly on the keyboard. Only commands that are in the 'Help' table or commands that are in the current symbol table can be executed in the immediate mode. The symbol table is generated every time a module is compiled and contains the names of the procedures and variables within it. We will go through an example to explain what is happening. We will try to assign a variable and then write its value to the terminal. We will purposely generate errors to show what is happening. Start off by clearing the Micro-Module by typing 'New'.

```
M> VAR x
M> x := 123
M> WriteInt(x, 0)
    ^ Unrecognized Key word or identifier
```

'WriteInt' is not known to the system so we will have to get it into the symbol table. To do this we must compile 'Terminal' as this is the module that contains the procedure 'WriteInt'.

```
M> C Terminal
M> WriteInt(x, 0)
Attempted use of un-initialised MODULE Terminal
```

'WriteInt' cannot be run until the 'Terminal' MODULE has been initialised.

```
M> Terminal
M> WriteInt(x, 0)
      ^ No such Identifier
```

This time its the 'x' that is unrecognized. The compiling of a new module clears the symbol table.

```
M> VAR x
M> x := 123
M> WriteInt(x, 0); WriteLn
123
```

Its done it!

If you want to use the immediate mode a lot, e.g. when learning or teaching, there is a simple method to make it easy. Look at the program 'Commands' on the evaluation disc that we used earlier. The function key F1 is already programmed to compile and run 'Commands'.

```
M> F1
M> VAR x
M> x := 123
M> WriteInt(x, 0); WriteLn
123
```

Any identifiers imported in 'Commands' will be added to the symbol table and hence will be accessible from the keyboard in the immediate mode. Your favourite commands can be included in the command file and compiled and run with a function key.

### 1.11 Debug

The Debug statement will stop a program in mid-flight and provide a prompt. The same rules apply when writing from this command line. To restate: Identifiers can only be used when they are visible and when they are in the current symbol table.

When stopping in a procedure to examine its local variables with 'Debug' it is possible to use 'WriteInt' etc. if they have been imported into the program under test.

### 1.12 Trace

Trace(TRUE) will turn the trace on and Trace(FALSE) off. This can be done anywhere in the program. The Trace is automatically turned off when entering or upon leaving a PROCEDURE. If a procedure called 'Tracer' is present it will be called after each trace output. The user can write his own code here which may be to write out some variables. Try this example of a FOR loop. Use the word processor for convenience.

```
MODULE Test;
FROM Terminal IMPORT WriteInt;
VAR x : INTEGER;
    PROCEDURE Tracer;
    BEGIN WriteInt(x,4) END Tracer;
BEGIN
    Trace(TRUE);
    FOR x:=1 TO 3 DO
    END
END Test.
```

Back in the module:

```
M> C Test
M> Test
```

### **1.13 The semi-colon**

The semi-colon must be used as per the Modula-2 specification. Please refer to the Modula-2 section.

### **1.14 Libraries**

Libraries can be user written. In DOS use the LIB sub-directory. They need two files to be prepared. The IMPLEMENTATION and the DEFINITION file, as they are called, use the .MOD and .DEF extensions respectively. The definition file is usually an edited down version of the implementation file. The implementation file contains the actual program routines whereas the definition file contains definitions of those procedures and variables that are allowed to be used (imported) by other programs or libraries. The best way to see an example is to examine say Terminal.MOD and Terminal.DEF in the LIB sub- directory.

As stated before the module directory can be displayed by typing 'List' and the contents of any module by 'List Name'. The contents is actually the definition module. We will now briefly review the main library modules provided.

### **1.15 Batch Files**

A batch file can be prepared and executed using the Alt F8 key. The file 'ALL' on the evaluation disk re-compiles all the library modules supplied in this Eprom set. The order of the files is important and takes care of the IMPORT dependencies. It is worth making a file for each project so that any library changes can be made knowing that the batch file will always re-compile the whole job. Time for a cup of tea!

## 1.16 Hardware Description

### 1.16.1 Digital I/O

A library is provided allow easy use of the digital I/O ports on the Micro-Module. To output to a port try:

```
F1  
OutPort(0); WritePort(0, 55h)
```

The 'h' implies a hexa-decimal number. Hex numbers beginning with a letter must be preceded by a 0.

```
WritePort(0, 0AAh)
```

Ports

This program shifts a 1 round all the digital bits in port 0.

```
MODULE Shiftp;  
FROM ADIO IMPORT WritePort, OutPort;  
FROM System IMPORT Delay;  
  
CONST p=0;  
  
VAR x : INTEGER;  
  
BEGIN  
  OutPort(p);  
  LOOP  
    x := 1;  
    REPEAT WritePort(p, x);  
      x := x << 1;  
      Delay(20)  
    UNTIL x>128  
  END  
END Shiftp.
```

## Channels

The digital line can be used on a channel basis as well. Here is the same program written using channels:

```
MODULE Shiftc;
FROM ADIO IMPORT WriteCh, OutPort;
FROM System IMPORT Delay;

CONST p=0; t=7;

VAR x : INTEGER;

BEGIN
  OutPort(p);
  LOOP
    FOR x:=0 TO t DO
      IF x=0 THEN
        WriteCh(t, FALSE)
      ELSE
        WriteCh(x-1, FALSE)
      END;
      WriteCh(x, TRUE);
      Delay(20)
    END
  END
END Shiftc.
```

### 1.16.2 Serial Port

The Micro-Module has one serial port on board. This serial port is buffered with RS-232 devices which generate the required voltage levels using on chip charge pumps. The serial port is a 10 way polarised boxed header. The Starter pack contains a ribbon cable that connects between this connector and a 9 pin D type connector commonly found on P.C. serial ports. Software is provided in the standard Micro-Module.

### 1.16.3 The Terminal

The main terminal is used for communication to the PC. This port has two special purposes other than transmitting and receiving characters. It will detect an Escape character (1Bh) and will deal with the loading and saving of programs.

The escape effect can be turned on or off with a command:

```
Escape(FALSE)      off
Escape(TRUE)       on
```

After turning off it will not be possible to escape from a running program. Only shorting out the Reset link will work. When using the port as a filing stream the system looks after all the byte transfers and all 256 characters are transmitted or received intact. The descriptor name for Port A is ':S1'. Note that, in the program below, the colon before the name specifies a device and not a file.

When using the serial port for an application where pure 8-bit binary characters are required (nearly all cases), the serial ports special filing functions will have to be disabled as well as the Escape effect. There is a PROCEDURE in the Osi library to do this which turns off the filing functions on a defined path.

```
Osi.File (path, FALSE);

MODULE Serial;
FROM Fio IMPORT Open, Close;
FROM InOut IMPORT WriteString, WriteLn;

VAR path : INTEGER;

BEGIN
  path := Open(':TERM');      (* open path *)
  WriteString(path, 'Hello from Micro-Module');
  WriteLn(path);
  Close(path)
END Serial.
```

It should not be necessary to do this, but is included for completeness.

#### **1.16.4 The Reset Link**

The reset link, as it suggests, resets the Micro-Module. It has the same effect as switching on.

#### **1.16.5 Expansion Connector**

The expansion connector contains all of the signals necessary to interface the on board microprocessor to the outside world. It is described more fully in the hardware manual. The evaluation board uses this connector to drive the two serial ports as previously described.

If the user wishes to use the expansion connector as well as the serial ports the cable can be replaced with one with an extra connector for the user. It uses standard 64 way IDC type ribbon cable and its pin out is listed later.

When using the expansion connector the extension cable should be kept as short as possible to avoid signal cross coupling. We would normally recommend up to 10 cm using standard ribbon cable.

### **1.17 Terminal Escape Sequences**

Characters received by the terminal are placed on the screen at the current cursor position and the cursor is incremented ready for the next character. The terminal is capable of handling some special control functions. These are accessed by an escape sequence. An escape sequence is a string of characters beginning with the character Escape (27). The second character is a command byte and is usually followed by parameter bytes. The command bytes are shown in decimal. For example to move the cursor to column 20, row 5 the following sequence would be sent. The commas are shown to make the reading clear and are not part of the data:

Esc, 13, 20, 5

In Modula:

```
WriteChar(1Bx); WriteChar(0Dx); WriteChar(14x);  
WriteChar(5x)
```

The 'x' denotes a hexa-decimal character constant.

As a Procedure:

```
PROCEDURE Tab(column,line : INTEGER);
VAR
  Buffer : ARRAY[0..3] OF CHAR;
BEGIN
  Buffer[0] := 1Bx; Buffer[1] := 0Dx;
  Buffer[2] := CHAR(column); Buffer[3] := CHAR(line);
  WriteBin(stdout,Buffer,4)
END Tab;
```

All of these routines are available in a Library file Video.MOD on the evaluation disc provided.

#### **1.17.1 Colour**

Esc, 10, colour

where colour is a number from 0 to 15

#### **1.17.2 Background Colour**

Esc, 11, colour

where colour is a number from 0 to 15

#### **1.17.3 Clear Screen**

Clears the current active window. It will leave other windows displayed.

Esc, 12

#### **1.17.4 Cursor Move**

Move the cursor to the line and column specified.

Esc, 13, column, line

#### **1.17.5 Define a Window**

Esc, 14, column1, line1, column2, line2

where column1, line1 is the top left corner  
column2, line2 is the bottom right corner

#### **1.17.6 Enter a Window**

Write next character to this window. Moving between windows marks the last position used and will return to it with this command.

Esc, 15, W

#### **1.17.7 Hide a Window**

Stop displaying this window. If characters are written to a hidden window they will not be displayed but are still recorded so that the hidden text will appear when the window is entered.

Esc, 16, W

#### **1.17.8 Display this Window on Top**

Moves this window onto the top. The last window using this command is on view. Other portions that overlap will be hidden.

Esc, 17, W

#### **1.17.9 Insert a Line**

Insert a line at the current cursor position scrolling down the screen below to make space.

Esc, 18

#### **1.17.10 Delete a Line**

Delete a line at the current cursor position scrolling the lines below up. This leaves the bottom line clear.

Esc, 19

**1.17.11 Clear to End of Line**

Clear the line to the right of the cursor.

Esc, 20

**1.17.12 Cursor on/off**

Turn the cursor on or off.

Esc, 21, c

where           c=0 off  
                  c=1 on

This page is intentionally left blank

## 2 The Modula-2 Programming Language

### 2.1 Introduction

Modula-2 was designed in 1980 and it is in fact one of the most recent implementations of a high level programming language. It was developed in Zurich by Professor Niklaus Wirth, the author of Pascal. Throughout this manual Modula-2 will be referred to as Modula.

The language has a deserved reputation for its provision of many high-level principles of software engineering. Modula presents itself very well on paper and is very legible. Programs are on the whole are self documenting with clear descriptive words. It is true to say that no programming language can make a bad programmer into a good one, but the Modula structure forces a good technique right from the start. This allows mortal engineers to use the language efficiently as well as computer scientists. And for those requirements where access to the system is required, Modula provides for legal low level programming.

Modula is a small core language. The actual Modula standard can be implemented very efficiently in a relatively small amount of memory. It is ideally suited to microprocessor technology with its advanced but simple architecture and is equally suitable to control and instrumentation as it is to science. As a bonus it is not a difficult language to master but its flexible control and data structures can produce some very ingenious algorithms. It is suitable for the engineer as well as the computer scientist.

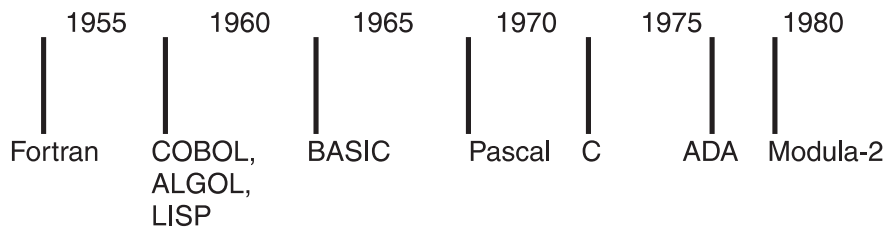
One of the key features of the language, if not the best, is its ability to take programs or routines that have been previously written, compiled and tested by yourself or by others and use them in your current design. Using them with the knowledge that they cannot be effected by your new program and will work exactly as they did when they were first designed. The program module is an entity in itself with a wall around it and nothing can go in or out without following a strict discipline.

Because of its modularity, Modula is very suitable for the development of large programs where several software engineers write separate sections. In saying this, it is also suitable for small programs with one engineer. The governing factor is that the modular approach encourages a tidy, well

maintained, working method. Not only will you be able to understand your own work when it requires a future modification, but so will a new engineer who may take over the job. It is our hope that in using this language you will be able to construct effective and elegant solutions to a whole range of problems, not just now, but in the many years to come.

## 2.2 History

When professional programmers discuss the many programming languages the following hold the main topics of conversation: FORTRAN, ALGOL, LISP, BASIC, Pascal, C and Modula-2. As can be seen from the time chart Modula-2 is the most recent in a line of language milestones. In many ways Modula-2 combines the very best ideas that are found in its predecessors.



## 2.3 Other Languages

### 2.3.1 Pascal

Overall, Modula and Pascal appear to be very similar. In fact Modula can be described as an enhanced and improved Pascal. Enhancements include libraries and system dependent routines and improvements have been made to tidy up loose ends such as ELSIF and the use of BEGIN END. The main difference is Modula's ability for separate compilation of individual Modules. Large programs can be broken down into small parts and information transfer between these parts is strictly controlled. Another important difference is that Modula is designed to work effectively on multi-tasking systems.

### 2.3.2 'C'

'C' is a structured language like Modula and is very similar in a lot of ways. The main difference lies in the Languages dialect. Do you prefer French to German. Modula is more 'wordier' than 'C'. Whereas 'C' uses operators in places, Modula would use a key word. Because of this Modula is less cryptic. The main differences are as follows:

- i Modula has strong type checking, 'C' does not. It is quite legal to assign an INTEGER to a character in 'C', i.e. store a 32-bit number into an 8-bit result.
- ii Modula is formally block structured whereas 'C' is not. Block structure is the languages ability to create logically connected units of code that can be referenced together. In Modula, procedures can have local procedures.
- iii In 'C' all references to procedures or identifiers outside the main program are resolved at link time. In Modula, they are always done at compile time. The big difference here is that at compile time the compiler can check the compatibility of the procedures with the main program, e.g. do they have the correct number of parameters and are the parameters of the correct type? At link time, only the procedure name is checked.

The big similarity between Modula and 'C' is that they both support and encourage separate compilation and hence break down large programs into manageable sections which is more than can be said of BASIC.

### 2.3.3 BASIC

BASIC is a not a structured language and bears little similarity to Modula. Standard BASIC does not offer the procedure with its own local variables but many new, non standard, versions exist with this as an extension. In the UK, BBC BASIC is well recognized as a very good implementation and extension of the BASIC standard. Although it does not enforce the use of good programming techniques many of the good BASIC programmer find little difficulties in using a structured language. The inexperienced BASIC programmer can occasionally get frustrated by the strict rules imposed by structured languages but soon learns that these are a blessing in disguise.

## 2.4 About this Section

This section is not a full description of the Modula standard. It attempts to give a brief description of all of the key words and their use and/or actions. Modula has no I/O facilities as standard. These are supplied as libraries and it is these libraries and their uses that form the main topic of this section.

We assume that the reader has a little experience at programming using BASIC or Pascal for example.

This section should be used in conjunction with its complementary sections for the Module. Please refer to the Operating system and Hardware section for more detailed hardware descriptions.

## 2.5 Versions

Modula is a very portable language so that any program using the standard key words should run on any other Modula systems. This assumes that the various compatible libraries are available to deal with the I/O. Having said this a few problems remain and one is that of which version of Modula is to be run. The latest edition, published in 1988, is the 4th and it is the edition that this implementation is based on. The main changes in this edition is the recommended use of the variable type INTEGER as opposed to CARDINAL. The standard functions accept INTEGERS as their parameters. Finally strings can only be assigned to character arrays with enough space for the string and its terminator. The 3rd edition introduced in 1983 saw the removal of export lists from definition modules.

The CMS Modula-2 has been released as a sub-set of the Modula standard. A section in the appendix explains which parts are missing and gives a precise history of the software issues showing details of any bug fixes or additional features added. The issue numbering system uses two numbers separated by a decimal point. The number to the left of the point is the version number and changes when extra features are added. Bug fixes leave the version number alone but increment the issue number to the right of the decimal point. A new version will reset the issue to 1. The first version was 1.1.

## 2.6 A Simple Program

Programs in Modula are called Modules. The program module can be considered as the main program that directs how other modules are to be used. Here is the simple program that does nothing but show the structure of all modules:

```
MODULE Name;                                module heading

TYPE                                         |
CONST                                       |
VAR                                          |
PROCEDURE                                   |

BEGIN
    < statement list >                       program body
END Name.
```

The compiler is a one pass compiler so that only back references are allowed. The order in the MODULE example above is recommended for general usage.

All I/O words have to be imported from library modules before use. There are a few resident system words which can be seen by typing 'Help' from the command line. Typing 'List' will show the current module directory displaying program and library modules.

This page is intentionally left blank

### 3 Syntax and Conventions

#### 3.1 Identifiers

The identifiers are the names that are given to variables and PROCEDURES when a program is being written. The Modula rules for identifiers are simple. They should only contain alpha-numeric characters and must start with an alpha character. For example:

lamp	jim	count	a1
FindMax	Square	The2nd	x

CMS Modula-2 allows the underline character as an extension.

lamp_post	Time_Out
-----------	----------

#### 3.2 Case Sensitivity

The convention for identifiers is based on the use of upper and lower case characters. Note that a convention is not a written rule but it is a recommended method that is used throughout the Modula community.

##### 3.2.1 Variables and Constants

Variables and constants should use lower case throughout and nouns where possible. If a double barreled word is used the second should have a capital letter.

VAR x, y,	count : INTEGER;		
CONST	maximum	=	100;
	minimum	=	10;
	lightHouse	=	7;

##### 3.2.2 Procedure names

PROCEDUREs should start with a capital and then precede with lower case letters. If a double barreled word is used the second word should also have a capital letter. Also PROCEDUREs should use verbs where possible.

WriteChar	ReadChar	Sort	Proper
WriteLn	OutPort	TurnOn	TurnOff

### 3.2.3 Types

Types should use the same convention as PROCEDURES but use the plural form.

```

TYPE Apples      = ARRAY [1..99] OF INTEGER;
     Rows        = [1..24];
     Columns     = [1..80];
     Screens     = ARRAY Rows, Columns OF CHAR;
    
```

### 3.2.4 Reserved words

The set of standard Modula reserved words MUST be in capital letters.

## 3.3 Standard Modula-2 reserved Words

The following is a list of the 40 reserved words used by Modula-2.

Reserved words used in Modula-2

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	<i>SET</i>
BY	<i>EXPORT</i>	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMEN-	PROCEDURE	VAR
	TATION		
DO	IMPORT	<i>QUALIFIED</i>	WHILE
ELSE	IN	<i>RECORD</i>	<i>WITH</i>

see note 1.

Note: The 1983 version 3 release of Modula does not require the words QUALIFIED EXPORT in definition modules.

### 3.4 Separators

For program legibility spaces can be freely placed anywhere within the program text. This of course precludes splitting up any of the key words including the double character operators. The compiler actually rejects all control characters of ordinal values 0 to 32 inclusive. This includes the Tab, New Line and Carriage Return characters.

The following two statements are equivalent.

```
x:=x+2;
x := x + 2 ;
```

The carriage return is a separator and will be treated like a space character. The carriage return will still act as a line end when commands are typed directly from the keyboard in immediate mode.

```
(* this is a valid, and silly, program *)
MODULE Separator; VAR x
: INTEGER; y
: CHAR; BEGIN x:=56;
y:=999 END Separator.
```

### 3.5 The Semi-colon

The semi-colon is used as a statement separator. The key word here is SEPARATOR. The semi-colon separates statements and is not used to terminate lines. This allows statements to occupy more than one line so that a programmer can format the text in any way that suits, to improve legibility. If a semi-colon is added to the end of a line, where it is not required, the compiler will just ignore it. This is usually called a null statement.

The semi-colon is one of the points that really bother the newcomer to the language. Think of the role of the semi-colon as a separator rather than a terminator. Example:

```
x:=1; y:=2; z:=3
```

Take two PROCEDURE bodies.

```
BEGIN WriteString('Hello ') END name
```

```
BEGIN WriteString('Hello '); WriteString('World') END name.
```

The second body has two statements in it and requires a separator. Let us rewrite it.

```
BEGIN
  WriteString('Hello ');
  WriteString('World')
END name
```

This is identical to the example above but now it looks as though a semi-colon is missing. Hence the apparent problem with newcomers. If an extra semi-colon was added to the second line the compiler would remove it anyway.

### 3.6 Comments

Comments can be freely placed anywhere in the program text by using the comment braces. These are the special double characters (\* and \*). Anything in between these will be ignored. Comment braces can be placed anywhere and are treated just like separators. Example:

```
x := y ; (* assignment *)
IF x = 10 THEN WriteString('Ten')
END (* IF *)
```

## 4 Modules

The MODULE is the very object that gives Modula-2 its name. The Module approach derives itself from the old saying “divide and conquer”. Large programs should be broken down into small manageable groups or Modules. The Latin word ‘modulus’ means a small measure.

### 4.1 Programs

The simplest and most common program is a self contained module. This program MODULE runs on its own without any references to anything outside of itself. In theory this is possible in a Modula system, but very unlikely, because there are no built in I/O words. The I/O libraries have to be IMPORTed into the MODULE for use. The libraries can be brought in from the PCs disc or can be stored in the Eproms. Our libraries are already in the EPROMs. Program example:

(\* assigns a two dimensional array of characters the days of the week and displays them \*)

```
MODULE Days;
FROM Terminal IMPORT WriteString, WriteLn;
VAR   x      : INTEGER;
      day    : ARRAY [1..7], [1..10] OF CHAR;

PROCEDURE Setup;          (* local procedure *)

BEGIN
    day[1] := 'Monday';
    day[2] := 'Tuesday';
    day[3] := 'Wednesday';
    day[4] := 'Thursday';
    day[5] := 'Friday';
    day[6] := 'Saturday';
    day[7] := 'Sunday'
END Setup;
```

```
BEGIN                                (* main body *)
    Setup;
    FOR x:=1 TO 7 DO
        WriteString( day[x] ); WriteLn
    END
END Days.
```

## 4.2 Libraries

One of the main features of Modula is that programs can be subdivided into lots of small parts. The parts once proven and tested become Libraries that can be used over and over again. Library MODULEs appear similar to program MODULEs but their main body is optional. If present it will be run before the program MODULE and can be used to initialise the library MODULE. The contents of library MODULEs contains predominantly PROCEDURES that will be used by the main program MODULE. Library MODULEs can also define TYPEs, CONSTants and VARIABLEs which can be IMPORTed if required. A library MODULE can be identified in the source code by the word IMPLEMENTATION at the start of the file. It has a sister called the DEFINITION MODULE. It is the definition MODULE that specifies what is and what is not available from the IMPLEMENTATION MODULE. This gives enough information to use the library. The source code is found in the implementation file. The Terminal and Video libraries included on the evaluation disc are good examples and these can be freely inspected. One note worth mentioning here. It is not necessary to supply the source code of a library for it to be used. Only the definition file and its compiled code are required. This gives a supplier the option of hiding the workings of a library. The compiled code is stored as a binary file, on disc or in the EPROMs, and can be IMPORTed directly. When a library is IMPORTed the system checks if it is already in memory. If not, it will load it into the RAM just like a normal MODULE. All of the library MODULE is loaded, not just the parts listed in the IMPORT statement.

The following listing is from the Video library and then follows an example using it.

```
DEFINITION MODULE Video;
CONST      black = 0; blue = 1; green = 2; cyan = 3; red = 4;
           magenta = 5; brown = 6; lightgray = 7;
PROCEDURE Colour(colour : INTEGER);
PROCEDURE Background(colour : INTEGER); PROCEDURE Cls;
PROCEDURE Tab(column,line : INTEGER);
END Video.
```

```
IMPLEMENTATION MODULE Video;
FROM Terminal IMPORT pathout;
FROM Osi IMPORT WriteBin;
CONST      black = 0; blue = 1; green = 2; cyan = 3; red = 4;
           magenta = 5; brown = 6; lightgray = 7;
PROCEDURE Colour(colour : INTEGER);
VAR  Buffer : ARRAY[0..2] OF CHAR;
BEGIN
    Buffer[0] := ESC; Buffer[1] := 0Ax;
    Buffer[2] := CHAR(colour);
    WriteBin(pathout,Buffer,3)
END Colour;
PROCEDURE Background(colour : INTEGER);
VAR  Buffer : ARRAY[0..2] OF CHAR;
BEGIN
    Buffer[0] := ESC; Buffer[1] := 0Bx;
    Buffer[2] := CHAR(colour);
    WriteBin(pathout,Buffer,3)
END Background;
```

```
PROCEDURE Cls;
VAR Buffer : ARRAY[0..1] OF CHAR;
BEGIN
    Buffer[0] := ESC; Buffer[1] := 0Cx;
    WriteBin(pathout,Buffer,2)
END Cls;

PROCEDURE Tab(column,line : INTEGER);
VAR Buffer : ARRAY[0..3] OF CHAR;
BEGIN
    Buffer[0] := ESC; Buffer[1] := 0Dx;
    Buffer[2] := CHAR(column); Buffer[3] := CHAR(line);
    WriteBin(pathout,Buffer,4)
END Tab;

END Video.
```

```
(*
Kaleidoscope of colour dots. This program will produce random coloured
dots on the terminal.
*)
```

```
MODULE Dots;
FROM Video IMPORT Colour, Background, Cls, Tab, black;
FROM Terminal IMPORT WriteChar;
CONST dot = DBx;
      x0 = 40;      y0 = 12;      (* screen centre *)
      Dx = 34;      Dy = 11;      (* size *)
VAR x ,y ,z : INTEGER;
BEGIN
    Background(black); Cls;
    LOOP
        FOR z := 0 TO 1 DO
            x := Rnd(Dx);
            y := Rnd(Dy);
            IF z THEN
                Colour( Rnd(15) )
            END IF
        END FOR
    END LOOP
END Dots;
```

```
        ELSE
            Colour(black)
        END;
        Tab( x0+x,y0+y ); WriteChar( dot );
        Tab( x0+x,y0-y ); WriteChar( dot );
        Tab( x0-x,y0+y ); WriteChar( dot );
        Tab( x0-x,y0-y ); WriteChar( dot );
    END
END
END Dots.
```

#### 4.2.1 Qualified Identifiers

IMPORT, when used on its own, requires the use of qualified identifiers. The source module name must be added to the front of the identifier and separated by a full stop.

```
    IMPORT Video;
    Video.Cls;
    Video.Tab(1,10);
```

#### 4.2.2 Unqualified Identifiers

The declaration FROM <module> IMPORT <id list> uses unqualified identifiers.

```
    FROM Video IMPORT Cls, Tab;
    Cls;
    Tab(1,10);
```

Both methods have the same result. The whole of the 'Video' module is imported and the statement shows which identifiers are to be used in this module.

### **4.3 Local**

See note 2.

These MODULEs are nested within the main body of program or library MODULEs. The local MODULE is used for special initialisation purposes and is only visible to its surrounding MODULE. Local MODULEs are best viewed as a way of encapsulating variables and procedures so as to hide the inner workings from inadvertent access. Local modules cannot be compiled separately and do not have a corresponding definition module. They are part of the module surrounding them.

#### **4.3.1 Export**

Identifiers within local MODULEs must be EXPORTed before they can be used in the surrounding MODULE and variables outside will require IMPORTing.

### **4.4 Version Control**

The very powerful concept of separately compiled MODULEs is not without problems, although these are small by comparison. The problem occurs when a library MODULE is edited and re-compiled. There is no way that an old program will know that the library routine has been modified and its call to it will probably be catastrophic. The only way out of this is to keep a check of what uses what and to re-compile all MODULEs that are effected by the change to the original library. To this effect the system word 'M2' will accept a list of MODULE names for compilation and saving.

#### 4.5 Battery backed RAM

Some modules have battery backed RAM available for program storage. To make matters easy any programs or libraries can be loaded into the RAM and Locked in. This effectively makes them part of the system and they will remain in the memory until UnLocked. This means that as a program is being developed its associated library programs can be added one by one and stored in the battery backed RAM. The following shows how the above example would work.

M2 Video	(* Compile and Save the Video library *)
Lock Video	(* Lock in battery backed RAM *)
C Dots	(* Compile the Dots main program *)
Dots	(* Execute the program Dots *)

This page is intentionally left blank

## 5 Errors

### 5.1 Error Types

It is the role of the compiler to try and indicate all of the errors that a programmer may write when constructing a program. An admirable statement! Unfortunately the compiler can only detect syntactical errors in the program text. These errors in textual structure are called Compile-time (static) errors. The system outputs error messages when the program files are being compiled into memory. The system will output a maximum of 8 errors before aborting the file. This will fill the screen with messages. In practice the first error usually causes an avalanche of following errors so it is best to tackle this one first.

The second type of errors occur at run time and are called Run-time (dynamic) errors. The expression evaluator will HALT the program if an arithmetic overflow occurs. e.g. division by zero. Also the sub-range types will produce an error if an assignment is out of range. Note that all mathematics are calculated at 32-bit precision and it is only the final assignment that will error if out of range. Another, less common, run time error is the system stack overflow. This can occur with a non terminating recursive PROCEDURE. After excessive calls, the systems memory for local variables gets completely used up and a stack overflow error is reported.

When the system halts at a run time error it will display, on the terminal, which module the error occurred in. If this is the current module, that has just been compiled, it will also show the name of the procedure taken from the recently compiled symbol table. If the error is in another module '.'\*' is added to the module name to show that the system cannot detect the actual procedure name. If this is the case the module in question can be re-compiled and the original program run again. This time the error message will show the name of the procedure.

There are also many errors that do not cause the program to HALT. These are known as soft errors and it is up to the programmer to deal with them. In many of the libraries provided there is a variable called 'Error'. This can be

## Modula-2

---

IMPORTed into a program and tested. The value has been returned from an operating system call and if it is non zero it indicates an error condition. There is a list of error numbers and their meanings in the Operating System Manual.

The final error, and the one the hardest to find, is that of the programmer. The program is not doing what it is supposed to.

Good luck!

## 6 Modula-2

### 6.1 Data Types

In Modula each variable or constant must be declared at the beginning of a program before use in the main body. The declaration will specify both its identifier name and its type. The identifier name is the programmers description of that variable. e.g. x, y, valve etc. The type specifies the form of data that will be held. e.g. INTEGER, CHAR, REAL.

#### 6.1.1 Simple Types

Modula provides several standard types that can be used to hold data in a programming application. The compiler will check for type compatibility in a program and flag errors if for example an INTEGER is assigned to a CHAR type. In this implementation all whole number mathematics are carried out to 32-bit precision and floating point numbers are to 64-bit precision.

##### 6.1.1.1 INTEGER

Integer types are whole signed numbers. This implementation uses 32-bits of storage for each integer giving a number range from -2147483648 to 2147483647.

```
VAR   x           : INTEGER;
      count, top  : INTEGER;
      weeks : [ 1 .. 52 ];
```

Integer constants can be declared in two ways. Decimal or Hexa-decimal. A Hexa-decimal number must always begin with a digit character and end with an h.

```
CONST   blue       = 4;
        maxcount   = 199;
        pattern    = 55h;
        pat2       = 0aah;
```

### 6.1.1.2 CARDINAL

A CARDINAL type is an unsigned number of the same bit length of the INTEGER above. This gives a range from 0 to twice that of the INTEGER .0 - 4294967295

Two other cardinal types have been defined. These are WORD and BYTE. They can be particular useful in minimizing the amount of storage required in a system, particularly in arrays.

WORD = [ 0 .. 65535 ];  
BYTE = [ 0 .. 255 ];

Byte and word types do not have any operators. Integer and Cardinal operators can be from the following:

Assignment

:=

Unary (\* INTEGER only \*)

+ -

Binary

+ - \* DIV MOD

Relationship operators ( BOOLEAN result )

= < > >= <= <> #

### 6.1.1.3 CHAR

The data type CHAR defines the computers basic character set including the printable symbols and the non-printing control codes. These are the normal ASCII characters associated with the computer keyboard. The ordinal values of the numbers in this system occupies the range 0..255 so that control characters and semi-graphic characters can be used.

```
VAR  chr          : CHAR;
     key          : CHAR;
     alpha       : ['a'..'z'];
```

Characters can be declared as constants and are specified in quotes.

```
CONST  alpha = 'A';
       beta  = "B";
```

The outdated octal numbers are not implemented. In their place character constants can be specified in hexa-decimal notation by following the number by an X. This is a CMS extension. Note that numeric constants use an h.

```
CONST  esc   = 1bx;
       ctlZ  = 1ax;
       LF    = 0ax;
       CR    = 0dx;
       num   = 12345678h;
```

Note: Strings are dealt with by character arrays

### 6.1.1.4 BOOLEAN

BOOLEAN is the truth type. It is either TRUE or FALSE. The main use for the BOOLEAN type is as a flag variable. Flags can be cleared at the beginning of a routine and tested during or after the routine. In Modula the results of relationships are BOOLEAN and not babies!

```
VAR flag : BOOLEAN;
```

There are no BOOLEAN constants.

```
flag := TRUE;
flag := FALSE;
```

**6.1.1.5 BITSET**

The type BITSET is included to allow bit manipulation. This is particularly useful when dealing with bits within bytes in hardware. There is a good example of its use in the setting of the parity of a serial port in file LIB\Fio.MOD. The BITSET operators provided are standard for SETs.

	SET	Logical Equivalent
Union	+	a OR b
Difference	-	a AND NOT b
Intersection	*	a AND b
Symmetric Difference	/	a EOR b
Equality	=	
Non-Equality	<> ( or # )	
Membership	IN	a = a AND b
Include	INCL( <id>, <bit> )	
Exclude	EXCL( <id>, <bit> )	
Constant	{0,6,7}	

6.1.1.6 REAL

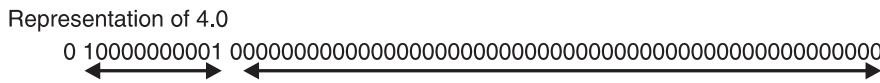
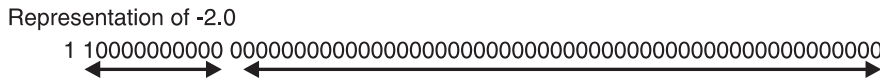
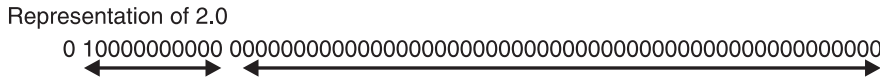
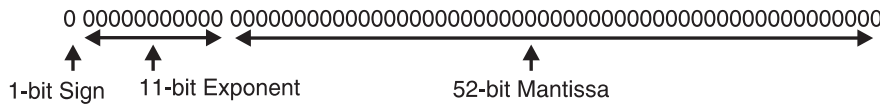
Mathematical real numbers can only be approximated on the digital computer. However the accuracy lost is very, very small when using 64-bit digital precision.

Format

Double precision IEEE floating point format

64-bits

52-bit mantissa  
 11-bit signed exponent  
 1-bit sign  
 $real = \langle 1.Mantissa \rangle * 2^{\langle Exponent - 1023 \rangle}$



Range

-1.7976931348623155e+308	minimum
1.7976931348623155e+308	maximum
2.2250738585072019e-308	smallest number

This is gives 17 significant decimal digits

Entry

real = [+ | -] digit{digit}.{digit} [scale factor]

scale factor = e | E [+ | -] digit{digit}

There must be no spaces before or inside the number. Conversion will end when a non digit is found. Note that the digit after the decimal point is optional.

Good real numbers:

1.2	3.	-2.1234	+5.12
12345.0	12e2	12E2	123e-10
123e+3	-2e+3		

Bad real numbers:

23	2 3	2 . 3	2 e3
- 3	+ 3	123e 2	5Ah
5A.0h			

Constants

Real number format must be used:

CONST	start = 2.0;	(* REAL *)
	end = 2;	(* ORDINAL *)

(\* numbers with 17 significant digits provide the full 52-bit accuracy for the mantissa. \*)

```
pi = 3.1415926535897958;  
e = 2.7182818284590446;
```

Operators

Binary

+	addition
-	subtraction
*	multiplication

/	division
=	equal
<>	not equal
#	not equal (synonym)
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

**Unary**

+	plus
-	minus

Testing for equality is not recommended with REAL numbers because of the approximate mathematical quantities. Testing if they are nearly equal is better.

e.g. IF x = y THEN

should be replaced by

IF x-y <= 0.0000001 THEN

The tolerance 0.0000001 can be chosen to suit the tolerance of x and y.

**Built in procedures**

**PROCEDURE ABS ( r : REAL ) : REAL;**

Return the absolute value of r.

ABS ( -1.23 )            return 1.23

**PROCEDURE FLOAT ( i : INTEGER ) : REAL;**

Convert an integer into real form

FLOAT ( 1234 ) returns 1234.0

**PROCEDURE TRUNC ( r : REAL ) : INTEGER;**

Convert a real number into an integer with truncation if necessary.

TRUNC ( 123.45 ) returns 123

**PROCEDURE Sgn ( r : REAL ) : REAL;**

Return the sign of a real number.

Sgn ( -123.8 ) returns -1.0

**Transcendental**

Trigonometric functions are provided from the MathLib0 library as follows:

DEFINITION MODULE MathLib0;

VAR precision : INTEGER;

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

PROCEDURE tan ( x : REAL ) : REAL;  
PROCEDURE arcsin ( x : REAL ) : REAL;  
PROCEDURE arccos ( x : REAL ) : REAL;  
PROCEDURE log10 ( x : REAL ) : REAL;  
PROCEDURE cube ( x : REAL ) : REAL;  
PROCEDURE power ( x, y : REAL ) : REAL;

END MathLib0.

**Sub-Range**

At run time when a variable is assigned the resultant value is checked for an overflow or underflow error for that particular type. It is possible to use the overflow and under flow detection by defining a sub-range for a variable. For example if a variable representing the days of the week is only allowed to have the values from 1 to 7, we could define this as a sub-range as follows:

VAR days : [ 1 .. 7 ];

If during execution the value of days falls outside the range specified a run time error will be produced. Sub-ranges are used in arrays so that the array bounds are checked at run time so that it is impossible to write over the end of an array. The type of a sub-range is obtained by inspection. It can be either INTEGER or CHAR. If another type is required it can be added to the declaration.

```
VAR  i      :      [0..15];          (* integer *)
      ch     :      ['A'..'Z'];      (* character *)
      crd    :      CARDINAL [0..255]; (* cardinal *)
```

### Expression compatibility

1. Only with the same TYPE
2. With ordinal constants

### Operators

=	equals
<>	not equals
#	not equals (synonym <>)
+	addition
-	subtraction
<	less than
>	greater than
<=	less than or equals
>=	greater than or equals

### Assignment compatibility

1. Must be the same TYPE
2. Compatible with all POINTER TYPEs

### Constants

any ordinal constant

#### 6.1.1.7 ADDRESS

The data type ADDRESS represents the microprocessors addressing range and is used to access memory or hardware. It is used strongly in connection with POINTERS.

### 6.1.1.8 POINTER

The POINTER TYPE is used to store the address of a variable of any other TYPE. There main use is for accessing memory or memory mapped I/O devices. With I/O devices the POINTER is permanently set to the address of that device and the device can be read or written via the POINTER. When used with memory, dynamic variables can be created with data structures such as trees and linked lists. The Library MODULE Storage has been provided to aid in the use of POINTERS and contains PROCEDURES for claiming blocks of memory and for Allocating and De-allocating these blocks into smaller sections for the individual variables.

#### Expression compatibility

1. Only with the same TYPE

#### Operators

=	equals
<>	not equals
#	not equals (synonym <>)

#### Assignment compatibility

1. Must be the same TYPE
2. Compatible with TYPE ADDRESS
3. When de-referenced with the same TYPE

#### Constants

NIL	unused POINTER
-----	----------------

If NIL POINTER is de-referenced it will produce a run time error

'NIL Pointer'

From the above strict rules it looks as though it is impossible to assign a POINTER to a constant address. This is true! The storage MODULE has a special built in PROCEDURE to do just this called AssignPointer. The above rules prevent inadvertent use of POINTERS which if assigned incorrectly can have a devastating effect on the system by writing all over its private parts. If the PROCEDURES from the MODULE Storage are used all should be well.

### De-Referencing POINTERS

If a POINTER variable is used on the left hand side of an assignment, or as an actual VAR parameter in a procedure call, it would refer to a variable of the TYPE specified in its declaration. The Unary operator ^ can be used to access that variable. It is known as a de-referencing POINTER. It is used after the POINTERS name.

```
e.g.  VAR  ptr    : POINTER TO INTEGER;
        ptr^   := 100;      (* INTEGER assignment *)
        ptr    := adr;      (* POINTER assignment *)
```

For those who wish to have direct memory access, this is how it is done.

```
VAR  adr    : ADDRESS;
     ptr    : POINTER TO INTEGER;
     adr    := 90000h; (* assign an ADDRESS a constant *)
     ptr    := adr;   (* assign a POINTER an address *)
```

Examples:

```
VAR  x, y    : POINTER TO INTEGER;
     c       : POINTER TO CHAR;
     i       : INTEGER;
     a       : ADDRESS;
```

(\* good POINTER expressions \*)

```
x := y;          (* same TYPE *)
x^ := y^ + i;    (* same TYPE *)
IF x=y THEN END;
x := a;
a := x;
```

(\* bad POINTER expressions \*)

```
x := c;          (* not same TYPE *)
a := x + y;      (* bad POINTER operator *)
x := 90000h;     (* no POINTER constants *)
```

There is a program example on the evaluation disc to demonstrate the use of POINTERS called 'Sort'. This inputs names from the keyboard and stores

them in memory using POINTERS. In this way only the names are stored and a large name array is not required. The POINTERS to the names are kept in an ARRAY of POINTERS. Any number of names can be entered showing that the storage requirement is expanding as required. After all the names are in, the program uses the 'Quick Sort' algorithm to sort the names into alphabetical order. Here the names remain fixed in position and it is the POINTERS that are exchanged in the POINTER array. Hence the movement of data is minimal. Finally the names are written in alphabetical order.

Note: ARRAY [] OF POINTERS is not implemented

This is no problem as ARRAY [] OF ADDRESS is ok and ADDRESS is compatible with POINTER.

POINTER TO ARRAY is ok.

POINTER TO POINTER is not allowed.

#### **6.1.1.9 PROCEDURE**

See note 3.

## 6.1.2 Structured Types

### 6.1.2.1 ARRAY

A variable which is defined as an array has many values which are stored one after another. Arrays are used where there are many variables of the same type that are treated in the same way. They are particularly suitable for tables and lists of data. The individual elements of an array can be accessed using an index for the start.

```
VAR <id> : ARRAY <simple type>{,<simple type>} OF type;
```

Examples:

```
intary   : ARRAY [1..99] OF INTEGER;
string   : ARRAY [0..15] OF CHAR;
flags    : ARRAY [1..4096] OF BOOLEAN;
screen   : ARRAY [1..79], [1..24] OF CHAR;
```

To access the array an index is used for example in the above integer array declaration the first element would be index 1, the next 2, up to 99. An index outside this range will produce a run time error.

```
intary [ 1 ] := 31; intary [ 2 ] := 97;
intary [ 99 ] := 0;
screen [ 1, 5 ] := 'A';
```

Strings are special cases of CHAR arrays. Here strings can be assigned a string constant independent of its length. The string will be terminated by the null character 0x. If the string constant is too big the compiler will report an error. Modula does not provide any string operations as standard. The libraries 'String' and 'Conv' provide all the necessary string operations.

```
string := 'David';      WriteString ( string );
```

Array assignment is possible by individual element, by slice or by the whole array assuming type compatibility.

Example:

```
VAR day, s : ARRAY [1..7], [0..15] OF CHAR;
day[1,0] := 'M';      (* element *)
```

```
day[2] := 'Tuesday'    (* slice of 15 elements *)
s := day;              (* whole array *)
```

### Open ARRAYS

Open ARRAYS are used when a PROCEDURE has to accept ARRAYS of differing lengths. These are particularly useful when dealing with strings of characters.

```
PROCEDURE OpenArray ( a : ARRAY OF CHAR );
```

Please observe that an open ARRAY always starts with element zero and ends with element HIGH(<id>). If it is called with an ARRAY [c..d] it will appear in the procedure as ARRAY [0..d-c].

#### 6.1.2.2 SET

See note 4.

#### 6.1.2.3 RECORD

See note 5.

#### 6.1.2.4 Enumeration

See note 6.

## 6.2 Type Compatibility

Modula is a strongly typed language. It was designed like this to try to detect as many errors as possible at compile time so relieving the burden of having to debug a program that is misbehaving without reporting any compile or run time errors. For example it is possible in the 'C' language to pass a character to an integer or to pass 2 parameters to a routine requiring 3. Both of which could produce some undesirable effects. This is not possible in Modula.

The following is a summary of the rules of type compatibility.

### 6.2.1 Expression

Expression compatibility defines the rules for mixing types in expressions.

e.g.  $x + 2 * y$

With  $x$  as the host type, the rules for  $y$  are:

1. Identical type as the host
2. Constant type of the host
3. Sub-range type of the host
4. Pointer to the type of the host

Examples:

```
VAR  i : INTEGER;   c : CARDINAL;
      si : [0..15];  sc : CARDINAL [0..15];
      pi : POINTER TO INTEGER;

i+i;   same type
i+2;   constant type
i+si;  sub-range type
i+pi^; pointer type
i+c;   expression incompatible with cardinal type
i+sc;  expression incompatible with cardinal sub-range
i+'a'; expression incompatible with character constant
i+3.4; expression incompatible with REAL constant
```

### 6.2.2 Assignment

In an assignment the type of the assignor is compared with the expression, i.e. the left hand side must be the same type as the right hand side. In the assignment  $x := y$  the following rules apply:

1.  $x$  and  $y$  are expression compatible
2. INTEGER is assignment compatible with CARDINAL and visa versa
3.  $x$  and  $y$  are identical structured types

Examples:

```

VAR   i : INTEGER;   c : CARDINAL; ch : CHAR;
      s1, s2 : ARRAY [0..15] OF CHAR;
      s3     : ARRAY [0..15] OF CHAR;

      i      := 2 + i;           same type
      i      := 2 + c;           INTEGER/CARDINAL
      s1     := s2;             same structure
      ch     := s1[5];          same type

```

assignment incompatible

```

      i      := ch;             type miss match
      ch     := s1;             type miss match
      s1     := s3;             different structure

```

The last error is one to ponder as the structures are inherently the same but have been defined separately. Because of this they have different type and are incompatible. When using arrays it is best to declare the array as a type at the beginning and use that definition to create new variables. In this way all the structures refer back to the same type.

```

TYPE  String = ARRAY [0..15] OF CHAR;

VAR

      s1 : String;
      s2 : String;

      s1 := s2                 assignment compatible, same structure

```

### 6.2.3 Parameter

When a procedure is called, the parameters of the calling routine must be compatible with the procedure itself. The rules of parameter compatibility are as follows:

1. If the parameter is a value parameter, the types can be assignment compatible.
2. If the parameter is a variable parameter, the types must be identical.

3. An open array is compatible if:  
    It is a single dimensioned array  
    and its base type is the same.

### 6.3 Type Conversion

Type conversion is accomplished using the built in type conversion functions.

PROCEDURE CHR (i : INTEGER) : CHAR

PROCEDURE ORD (c : CHAR) : INTEGER

PROCEDURE TRUNC (r : REAL) : INTEGER

PROCEDURE FLOAT (i : INTEGER) : REAL

Also the use of any standard type word can be used to 'cast' a conversion from any type to its own type. This in fact just defeats the compilers type checking so that for example the ordinal value of a character can be placed in a cardinal variable.

REAL( x )

INTEGER( x )

CARDINAL( x )

ADDRESS( x )

WORD( x )

CHAR( x )

BITSET( x )

BYTE( x )

x is any type.

## 6.4 Declarations

The previous descriptions of data types introduced the VAR and CONST declarations. The following shows how these can be entered into a program. The TYPE, CONST, VAR and PROCEDURE declarations are optional and can be in any order.

```
MODULE Name;
  [ TYPE <id>    = TYPE;
    {<id>      = TYPE; } ]
  [ CONST <id>  = <constant expression>;
    {<id>      = <constant expression>; } ]
  [ VAR <id list> : TYPE;
    { <id list>: TYPE; } ]

  { PROCEDURE Name; <body> END Name; }

BEGIN
  <statements>
END Name.
```

A constant expression can use integer maths.

```
+      -      DIV      MOD      ()
```

Example:

```
MODULE      Setup;
FROM        Terminal IMPORT WriteInt, WriteLn;
TYPE        Numbers = INTEGER;
CONST       bottom = 10;
            top = bottom + 256;
VAR         x : Numbers;

PROCEDURE OutAB(a,b:INTEGER);
BEGIN
  WriteInt(a,4); WriteInt(b,4); WriteLn
END OutAB;
```

```
BEGIN
    x := 999;
    OutAB(bottom,top)
END Setup.
```

## 6.5 Control Structures: Selection

All programs consist of three basic constructs:

- Statement sequences
- Path Selection
- Repetition

A program would contain several statement sequences. The control structures are responsible for determining which order the sequences are executed. Path selection occurs when a program needs to make a decision which statement sequence is to be executed next and repetition controls the repeated execution of the same sequence of statements. A statement sequence is constructed from one or more individual statements.

In Modula the path selection can be achieved by one of two methods. These are the IF and CASE structures. Repetition is provided by four methods, namely the FOR, REPEAT, WHILE and LOOP structures.

### 6.5.1 IF THEN ELSIF ELSE END

```
IF      <conditional expression> THEN <statement sequence>
{ ELSIF <conditional expression> THEN <statement sequence> }
[ ELSE <statement sequence> ]
END;
```

If the conditional expression is true then the following statement sequence is executed until an ELSIF, ELSE or END statement is reached. Then the program continues with the statement following the END. The ELSIF is optional. One or several ELSIF clauses can be included if required and will be tested in sequence if the preceding conditions are false. Finally if all the conditions are false the optional ELSE clause will be executed.

Examples:

```
IF x=y THEN
    WriteString('Equal')
END;

IF x=y THEN
    WriteString('Equal')
ELSE
    WriteString('Not equal')
END;

IF x=y THEN WriteString('Equal')
ELSIF x<y THEN WriteString('Less than')
ELSE WriteString('Greater than')
END;
```

### 6.5.2 CASE OF | END

```
CASE <ordinal expression> OF
    <label> : <statement sequence>
    { | <label> : <statement sequence> }
    [ ELSE <statement sequence> ]
END;
```

The labels must be of the same type as the expression and take the form of a constant list.

The expression following the CASE statement is evaluated to produce a selector value. This value is compared against the individual case labels. If a match is found the statements belonging to that label are executed until an ELSE or a | is reached whereupon execution will jump to the case END and continue from there.

Examples:

```
CASE x OF
    1 : WriteString('one')
    |
    2 : WriteString('two')
    |
    3, 5 : WriteString('Three or five')
```

```
|
    0, 4 : (* do nothing *)
|
    6..9 : WriteString('Six to Nine')
ELSE
    WriteString('Greater than 9')
END;
CASE ch OF
    'a'..'z' : WriteString('Lower case letter')
|
    'A'..'Z' : WriteString('Upper case letter')
|
    '1'..'9' : WriteString('Digit')
ELSE
    WriteString('non alpha character')
END;
```

## 6.6 Control Structures: Iteration

These four control structures deal with the repetition of statement sequences.

### 6.6.1 FOR TO BY DO END

```
FOR <var> := <start exp> TO <end exp> [ BY <step exp> ] DO
    <statement list>
END;
```

FOR structures are the most convenient to use to repeat a statement sequence an exact number of times. The structure has an inherent counting mechanism. The variable used must be an INTEGER, CARDINAL or CHAR type or a sub-range of. When the structure is entered the variable is assigned with a starting value. A final value and an optional step value are also evaluated. The statement sequence is then executed until the END is reached whereupon the step value is added to the variable and a test is made to see if the result is greater than the final value. If not the statement sequence is executed again. The value of the variable can be used within the structure.

The optional step value can be negative and if not included the step value will default to 1.

Examples:

```
FOR x:=1 TO 10 DO WriteInt(x) END;

FOR x:=1 TO 9 DO
  FOR y:=1 TO 9 DO
    WriteInt(x*y,4)
  END;
  WriteLn
END;

FOR reverse:=10 TO 1 BY -1 DO WriteInt(reverse,4)

END;
```

As an extension in CMS Modula-2, EXIT can be used from within a FOR loop.

### 6.6.2 REPEAT UNTIL

```
REPEAT <statement list> UNTIL <conditional expression>;
```

This structure will execute the statement list after the REPEAT word until it reaches the UNTIL clause where it will test the following expression. If the expression is true the program will continue. If false the statement list will be repeated.

Examples:

```
x:=1;
REPEAT WriteInt(x,0); x:=x*2 UNTIL x>256;
REPEAT UNTIL Ready();
```

### 6.6.3 WHILE DO END

```
WHILE <conditional expression> DO <statement list> END;
```

The WHILE structure is similar to the previous REPEAT structure except that the condition to stop repeating the statements is done at the start of the list

instead of the end. If the condition is true the statement list is executed. If false the program execution commenced after the END clause.

Examples:

```
x:=1;
WHILE x<=256 DO
  WriteInt(x,0);
  x:=x*2
END;
```

#### 6.6.4 LOOP EXIT END

```
LOOP <statement list> END;
```

This forms a continuously repeating sequence of statements. This is the usual loop that the main program takes. Even so, it is still possible to exit from this loop with the EXIT statement. This will cause an exit from the current loop only. Note that the Escape key will break a program from anywhere on the CMS Module.

Examples:

```
LOOP ReadString(ans);      WriteString(ans) END;
LOOP
  ReadChar( key );
  IF key=1Ax THEN EXIT END; (* Ctrl Z to exit *)
  WriteChar(key)
END;
```

### 6.7 PROCEDURES

PROCEDURES are used to sub-divide a program into small manageable units or sub-programs. This has two main uses. The first allows repetitive actions to be written as one PROCEDURE that can be called from several places and the second, and probably the more important, allows the program structure to group together all those operations that are function related. It is quite acceptable that a PROCEDURE is only called once. The PROCEDURE declaration is very similar to a MODULE.

PROCEDURES can take any parameter type. The actual variables and expressions used in the PROCEDURES call must always match those in the formal definition. The number of parameters must always be correct. There is comprehensive error checking built into the compiler. VARIABLE and value parameters are allowed and ARRAYS can be open or closed.

```
PROCEDURE <procedure name> [ ( <parameter list> ) ];
[ TYPE { <type declarations> } ];
[ CONST { <constant declarations> } ];
[ VAR { <variable declarations> } ];
{ < local procedure declarations> };
BEGIN
    <statement list >
END <procedure name>;
```

### 6.7.1 Visibility

The above definition shows a PROCEDURE declaration within a PROCEDURE. These inner declarations are local to the enclosing PROCEDURE and can only be called from within it. They are said to be only visible within the enclosing PROCEDURE. The overall rule is as follows:

1. An identifier can be used within the PROCEDURE that it was declared in and within any local PROCEDURES belonging to the first PROCEDURE. Subject to rule 2.
2. If a new identifier is defined, in a local PROCEDURE, with an identical name to a previous one, that is still visible by rule one, then the new identifier takes over as in rule 1.

The rules of visibility apply to all identifiers.

### 6.7.2 Local and Global Variables

The difference between MODULES and PROCEDURES shows up when we consider local and global variables. Any variable declared within the main part of a MODULE is said to be global to the whole program. Subject to rule 2 above. These variables are permanent. Once the MODULE starts to run they will remain resident in the computers memory throughout the execution of the program. This is true of local MODULES too. A PROCEDURE on the other hand has temporary variables. These variables only exist during the

execution of that PROCEDURE. If the same PROCEDURE is called more than once, it cannot be assumed that the variables left after the first call will be valid. They almost certainly will not be.

### 6.7.3 Parameters

All PROCEDURES can take optional parameters. There are two forms of parameters. The value and the variable parameter.

#### VALUE

The value parameter is evaluated before entry to the PROCEDURE and its value is passed to the PROCEDURE where it behaves exactly like a local variable. Its value can be used, including a new assignment, within the PROCEDURE. When the PROCEDURE ends the value will be lost.

Example:

```
PROCEDURE Send ( x : INTEGER );           (* declaration *)
Send ( y + 2*z );                         (* call *)
```

#### VARIABLE

The variable parameter is actually a pointer back to another variable. The PROCEDURE will use this variable during execution. The effect of the PROCEDURE on that variable is permanent. Another simpler way of looking at variable parameters is to imagine that they are passed into the PROCEDURE on entry and out of it on exit. The variable parameter can be much faster in execution than the value parameter, especially with arrays, as no copying is required. Only a pointer is passed to the PROCEDURE.

Example:

```
PROCEDURE Modify (VAR x : INTEGER );     (* declaration *)
Modify ( y );                             (* call *)
```

### Open Array

If an ARRAY is defined, as a PROCEDURE parameter, and its elements are not present it is said to be an open array. At run time the calling routine will use the size of the calling array. The word HIGH can be used to find its size.

```
PROCEDURE Oarray ( ary : ARRAY OF CHAR )
```

Try to avoid value parameters for arrays as there will be a lot of time spent copying them into the PROCEDURE. Variable parameters are much faster as they pass pointers.

### Order of Evaluation

The order of evaluation of parameters is from left to right as they appear in the definition.

## 6.7.4 Function PROCEDURES

PROCEDURES can be defined as functions and used in expressions. In this case they will return a value to the expression. Care must be taken in the programming of function procedures to avoid the use of global variables and variable parameters as these can lead to unusual side effects. It is possible that a function procedure can change a global variable that itself uses later. A golden rule is do not use global variables or variable parameters in function procedures.

```
PROCEDURE Heat () : INTEGER;      (* declaration *)  
x := Heat();                      (* use *)
```

The brackets on the end of the function procedure are necessary even if there are no parameters present. This serves to identify a function procedure from a variable in a program listing.

### 6.7.5 Recursion

The PROCEDURES of Modula are fully recursive in that they can call themselves. Each time a PROCEDURE is called it creates a new work space for its local variables. This is maintained until the END of that PROCEDURE. If a PROCEDURE calls itself the existing work space remains untouched and a new work space is created. This is best demonstrated by an example.

```
(* This program will enter keys from the keyboard until a Return
is pressed. It will then display them in reverse order. *)
MODULE Reverse;

FROM Terminal IMPORT WriteChar, WriteLn, ReadChar;
PROCEDURE Rev;
CONST CR = 0Dx;
VAR key : CHAR;
BEGIN
    ReadChar(key); WriteChar(key);
    IF key=CR THEN
        WriteLn
    ELSE
        Rev;
        WriteChar(key)
    END
END Rev;

BEGIN
    Rev; WriteLn
END Reverse.
```

When Rev is called the new PROCEDURE will create a new work space so that the existing value of key will remain intact. The new called PROCEDURE will then get a new key value. This will go on until RETURN is pressed. Now the PROCEDURES will return, one by one, each writing out their own local value of key. Hence the order of calling is reversed on return.

```
(* Real representation of integers
   i      number
   d      number of digits to right of decimal point
   f      print field width *)

PROCEDURE WrReal(i,d,f : INTEGER);
VAR v : INTEGER;
BEGIN
  IF d>0 THEN
    v := i MOD 10;
    WrReal(i DIV 10,d-1,f-1);      (* recursion *)
    WriteInt(v,0)                  (* returns to here *)
  ELSE
    WriteInt(i,f-1);
    WriteChar('.')
  END
END WrReal;
```

#### 6.7.6 BYTE and WORD parameters

See note 7.

## 7 Language Summary

### 7.1 Key Words

This section briefly describes all of the key words. The word is listed in bold type and is followed by an example of its use and then a description. The following syntax is used.

```

|           or
[ option ]
{ repeated option }
```

Example:

& is a synonym of AND and would appear in the text as :

```

AND | &
FOR <id> := <start> TO <finish> [ BY <exp> ] DO
    BY <exp> is optional
VAR <id> list : <type>;
    { <id list> : <type>; }
```

there can be as many extra lines of declarations as required.

#### 7.1.1 ABS

```
PROCEDURE ABS ( x : INTEGER ) : INTEGER;
```

```
PROCEDURE ABS ( x : REAL ) : REAL;
```

Function procedure that converts a signed numeric expression into a positive number.

Example:

```

ABS ( -123 )      returns 123
ABS ( 99 )       returns 99
ABS ( -1.23 )    return 1.23
```

Note. A REAL constant must have a '.' or an 'E' in the word otherwise it will be treated as INTEGER.

### 7.1.2 ADR

PROCEDURE ADR ( VAR x : <any type> ) : ADDRESS;

ADR is a function procedure that returns the absolute address of where a variable is stored. The variable can be of any type. The value returned is of type ADDRESS and can be used by POINTERS. ADR can also return the start address of a machine code PROCEDURE. This is very useful in debugging.

Example:

```
VAR x : INTEGER;
ADR ( x )      returns the address where x is stored
```

### 7.1.3 ADDRESS

VAR <id list> : ADDRESS;

ADDRESS is a type of variable which specifies a whole number of 32-bits which represents a system address. It requires 4 bytes of storage space and represents a value of from 0 to 4294967295 or 0 to 0FFFFFFFFH.

Examples:

```
VAR   adr   : ADDRESS;
      aryadr : ARRAY [0..255] OF ADDRESS;
```

#### 7.1.4 AND or &

<boolean term>        AND        |        &        <boolean term>

AND is a BOOLEAN operator that requires two BOOLEAN values. It can be used in expressions that yield a BOOLEAN result.

Example:

```
IF (x>9) AND (x<99) THEN
```

#### 7.1.5 ARRAY

```
VAR <id list> : ARRAY <simple type> OF <type>;
```

ARRAY is used to specify the structured data storage of variables of identical types.

Example:

```
VAR a, b        : ARRAY [ 1..100 ] OF INTEGER; (* 2 arrays *)  
VAR line       : ARRAY [ 1..15 ] OF CHAR;
```

multi-dimensional arrays

```
VAR c : ARRAY [ 1..255 ] OF ARRAY [ 0..15 ] OF CHAR;
```

this can be abbreviated to

```
VAR c : ARRAY [ 1..9 ], [ 0..15 ] OF CHAR;
```

### 7.1.6 BEGIN

BEGIN <statement list> END <name>;

BEGIN marks the end of the declaration section of a MODULE or PROCEDURE and the beginning of the statements.

Example:

```
MODULE Hello;
FROM Terminal IMPORT WriteString, WriteLn;
BEGIN
    WriteString('Hello');
    WriteLn
END Hello.
```

### 7.1.7 BITSET

VAR <id list> : BITSET;

BITSET is used in type declarations. It represents a value containing 8 set members. This, of course, is 1-byte of memory storage. The BITSET type is for use when bit manipulation of bytes is required. This is of particular use in low level programming of hardware devices. The BITSET type only requires 1-byte of storage space. As the minimum storage space in this implementation is 2 bytes, a single BITSET value will require 2 bytes. When used in arrays all bytes are used.

Examples:

```
VAR    pattern :    BITSET;
       bitary  :    ARRAY [ 1 .. 256 ] OF BITSET;
```

### 7.1.8 BOOLEAN

```
VAR <id list> : BOOLEAN;
```

BOOLEAN is used in type declarations. It represents a value of either TRUE or FALSE. The BOOLEAN type only requires 1-bit of storage space. As the minimum storage space in this implementation is 2 bytes, a single BOOLEAN value will require 2 bytes. When used in arrays the 2 bytes will hold 16 values so that large BOOLEAN arrays are very efficient in their storage requirements.

Examples:

```
VAR   flag           : BOOLEAN;
      flags          : ARRAY [ 1 .. 8192 ] OF BOOLEAN;
```

### 7.1.9 BY

```
FOR <id> := <start> TO <finish> [ BY <increment> ] DO <statement list>
END;
```

BY specifies the optional increment, or decrement if negative, in the FOR statement. The following expression is evaluated to form an INTEGER. This whole signed number will be added to the control variable and the result tested to determine the end of the FOR structure. The BY expression produces a constant that cannot be changed during the execution of the structure.

Example:

```
FOR x:=0 TO 10 BY 2 DO <statement list> END;
FOR x:=10 TO 0 BY -2 DO <statement list> END;
FOR x:=start TO finish BY step DO <statement list> END;
```

### 7.1.10 BYTE

VAR <id list> : BYTE;

BYTE is a system word which specifies a type of 8-bits or a byte. It represents a value of from 0 to 255. The BYTE type only requires 8-bit of storage space. As the minimum storage space in this implementation is 2 bytes of storage, a single BYTE value will require 2 bytes. When used in arrays the 2 bytes will hold 2 values so that BYTE arrays are more efficient in their storage requirements.

BYTE variables can only be assigned. There are no BYTE operators.

Examples:

```
VAR  b      : BYTE;
     ba     : ARRAY [0..255] OF BYTE;
```

### 7.1.11 CAP

PROCEDURE CAP ( x : CHAR ) : CHAR;

This function procedure returns the upper-case equivalent of a variable or constant of type CHAR. If the variable or constant is not lower-case the character is unaffected.

Examples:

```
CAP ( 'a' )  returns 'A'
CAP ( 'B' )  returns 'B'
CAP ( '7' )  returns '7'
```

### 7.1.12 CARDINAL

VAR <id list> : CARDINAL;

CARDINAL is a type of variable which specifies a whole number of 32-bits requiring 4 bytes of storage space. It represents a value of from 0 to 4294967295.

Examples:

```
VAR   c      : CARDINAL;
      ca     : ARRAY [0..255] OF CARDINAL;
```

### 7.1.13 CASE

```
CASE <expression> OF
  <label> : <statement sequence>
  { | <label> : <statement sequence> }
  [ ELSE <statement sequence> ]
END;
```

The CASE statement is a selector structure, described more fully in a previous section, that tests conditions and routes the program flow accordingly.

The CASE expression is evaluated to produce a case selector. This is compared against the case labels. If a match is found the statements after the colon are executed until an optional ELSE clause or a | case separator is reached when the execution continues after the END.

Example:

```
CASE x OF
  1 : WriteString('one')
  |
  2 : WriteString('two')
  |
  3 : WriteString('three')
  ELSE
    WriteString('not 1 to 3')
  END;
```

#### 7.1.14 CHAR

VAR <id list> : CHAR;

CHAR is the variable type that specifies the use of ASCII characters. It represents a value of printable symbols and non-printable control codes. There are 128 characters using 7 or 8 bits of storage. This represents an ordinal value of 0 to 255 or a byte. The CHAR type uses 8-bits of storage space. As the minimum storage space in this implementation is 2 bytes, a single CHAR value will require 2 bytes. When used in arrays the 2 bytes will hold 2 values so that CHAR arrays are more efficient in their storage requirements.

When used in an array the characters form strings that can be used as normal English words. A character array can be used as a string variable.

Examples:

```
VAR c      : CHAR;
VAR ca     : ARRAY [0..127] OF CHAR;
```

#### 7.1.15 CHR

PROCEDURE CHR ( x : INTEGER ) : CHAR;

CHR is a function procedure that converts a number into a character. This is done at run time and can take an expression as its parameter.

Examples:

```
CHR ( 65 )      returns 'A'
CHR ( 65 + x )  is valid
```

Note:  $x = \text{CHR} ( \text{ORD}(x) )$

### 7.1.16 CONST

```
CONST <id> = <constant expression>;  
    {<id> = <constant expression>; }
```

The CONST declaration is used to declare constants at the beginning of MODULEs or PROCEDUREs. No type definition is required because the form of the constant is sufficient to determine the type. Constant expressions are allowed in numerical constants using the operators +, -, \*, DIV and MOD.

Example:

```
CONST      top    =    99;  
           hexa   =    55AAH;  
           alpha  =    'a';  
           LF     =    0Ax;  
           name   =    'Jumbo';  
           middle =    (top - bottom) DIV 2;
```

### 7.1.17 DEC

```
PROCEDURE DEC ( VAR x : <ordinal type> [; y : INTEGER] );
```

This is a proper procedure that will decrement any ordinal type such as INTEGER, CARDINAL, CHAR or sub-ranges of. The second parameter is optional and allows a larger decrement value.

```
i:=25 ; DEC ( i );      results in i = 24  
c:='C' ; DEC ( c );    results in c = 'B'  
i:=5 ; DEC ( i,3);     results in i = 2
```

### 7.1.18 DEFINITION

The DEFINITION MODULE holds all the PROCEDURE, VARIABLE, TYPE and CONSTANT declarations that are available to other modules as library calls. The DEFINITION MODULE is usually an edited down version of its equivalent IMPLEMENTATION MODULE with perhaps some extra comments added to give fuller descriptions to the user.

```
DEFINITION MODULE File;  
VAR error : INTEGER;
```

(\* This procedure opens a file. It is passed the name of a device and returns a handle if the file is opened successfully. If there is an error the error number is written to the variable error, else error is zero \*)

```
PROCEDURE Open ( string : ARRAY OF CHAR ) : INTEGER;  
END File.
```

### 7.1.19 DIV

<numeric term> DIV <numeric term>

DIV is the numeric operator that performs the division of whole numbers. The result of the division is a whole number and the remainder is discarded. Division by zero will produce an arithmetic overflow. Division by negative numbers is not defined by the Modula standard but a truth table of what happens in this implementation is presented below. As can be seen the result is the same as for REAL numbers.

x	y	x DIV y
+	+	+
+	-	-
-	+	-
-	-	+

**7.1.20 DO**

FOR <id>:=<start> TO <finish> DO

WHILE <testable exp> DO

DO is a separator used in FOR and WHILE structures. It isolates the last expression from the following statement.

Examples:

```
FOR x:=0 TO 10 DO <statement list> END;
```

```
WHILE x<10 DO <statement list> END;
```

**7.1.21 ELSE**

```
IF <testable expression> THEN <statement list>
  { ELSIF <testable expression> THEN <statement list> }
  [ ELSE <statement list> ]
END
```

```
CASE <expression> OF
  <label> : <statement sequence>
  { |<label> : <statement sequence> }
  [ ELSE <statement sequence> ]
END;
```

ELSE is the default clause used in the IF and CASE selector structures. If all of the tests in a structure are FALSE then the statements after the ELSE clause are executed.

Example:

```
IF x>10 THEN <statement list>
ELSE <default statement list>
END;
```

### 7.1.22 ELSIF

```
IF <testable expression> THEN <statement list >
    { ELSIF <testable expression> THEN <statement list> }
    [ ELSE <statement list>]
END
```

The ELSIF clause is an optional method of adding more testable conditions to the IF selector structure. If the first IF condition is FALSE the following ELSIFs will be tested until a TRUE condition is found.

Example:

```
IF x>10 THEN <statement list >
ELSIF x>100 THEN <statement list >
ELSIF x>1000 THEN <statement list>
END;
```

### 7.1.23 END

END is the clause that, as it sounds, ends MODULEs, PROCEDUREs and most structures. The name of the MODULE or PROCEDURE is required after the END clause. The final END should be terminated by a full stop.

Examples:

```
MODULE Exam;
<declarations>
BEGIN
    <statements>
END Exam.
```

**7.1.24 EXCL**

```
PROCEDURE EXCL ( set : <set type>; bit : INTEGER );
```

This SET procedure excludes members from a SET. Note that BITSET members are from 0 to 7 inclusive.

```
    VAR bit : BITSET;  
    EXCL( bit, 1 )
```

**7.1.25 EXIT**

```
LOOP <statement list>; EXIT; <statement list> END;
```

The EXIT clause can optionally be used in a LOOP structure to terminate a loop and jump to the statements after the LOOPs END. The exit will be from the current LOOP in nested LOOPS.

Example:

```
    LOOP  
        <statement list>;  
    IF x>10 THEN EXIT END;  
        <statement list >  
    END;
```

### 7.1.26 FALSE

FALSE is the BOOLEAN constant representing NOT TRUE. In this implementation its ordinal value is 0.

Example:

```
flag := FALSE; (* flag is BOOLEAN *)
IF flag THEN <statement list> END;
```

### 7.1.27 FLOAT

```
PROCEDURE FLOAT ( i : INTEGER ) : REAL;
```

Converts an integer into real form

```
    FLOAT ( 1234 )      returns 1234.0
```

### 7.1.28 FOR

```
FOR <id> := <start> TO <finish> [ BY <increment> ] DO
    <statement list >
END;
```

The FOR statement, described more fully in a previous section, is a control structure which deals with counting loops. It has a starting value, a finishing value and a increment value. A variable known as the control variable is used to do the counting and this is available within the structure.

Example:

```
FOR x:=1 TO 9 DO WriteInt(x,0) END;
```

**7.1.29 FROM**

FROM <module name> IMPORT <id> {,<id>};

FROM is used with IMPORT to specify the use of unqualified identifiers from a library MODULE. The identifiers can be PROCEDURE names or constants, variables or types.

Up to 64 imports are allowed in any one module.

**7.1.30 HALT**

HALT is a proper procedure that will end the program prematurely. It is normally used when a unrecoverable error has been detected.

Example:

```
IF temperature >99 THEN
    ShutDown;
    HALT
END;
```

The message Halt will be displayed on the terminal output.

### 7.1.31 HIGH

PROCEDURE HIGH ( VAR a : ARRAY OF <any type> ) : INTEGER;

This function procedure will return the value of the last element of an array. It is for use with open array parameters where an array of variable length is passed to a procedure. Note that the first array element will always be zero when in the procedure.

Example: To clear any INTEGER array use:

```
PROCEDURE Zeroarray( VAR anyarray : ARRAY OF INTEGER );
VAR x : INTEGER;
BEGIN
    FOR x:= 0 TO HIGH(anyarray) DO
        anyarray[x]:=0
    END
END Zeroarray;
```

### 7.1.32 IF

```
IF <testable condition> THEN <statement list>
    { ELSIF <testable condition> THEN <statement list> }
    [ ELSE <statement list> ]
END;
```

The IF statement, described more fully in a previous section, is a control structure which deals with selection of the program path. The following condition is tested to determine which statements are to be executed next.

Example:

```
IF x=y THEN <statement list> END;
```

### 7.1.33 IMPLEMENTATION

An IMPLEMENTATION MODULE is the source code for a library that is to be compiled separately from the main programs or libraries that will use it. This MODULE, together with the DEFINITION MODULE, form the basic building blocks that make up a Modula system.

```
IMPLEMENTATION MODULE File;
```

### 7.1.34 IMPORT

```
IMPORT <module name> {,<module name>;
```

IMPORT specifies a list of library modules that are required by a MODULE. As there are no identifiers specified, identifier use will have to be qualified with the name of the parent MODULE.

```
IMPORT Terminal, Video;  
  
Terminal.WriteInt( .... );  
Video.Cls;
```

### 7.1.35 IN

<set1> IN <set2>

The SET operator IN can be used to test whether the members of one set are in another set. The operator returns a BOOLEAN result.

```
bit := {1};
IF bit IN {1,3,5,7} THEN
```

This would be TRUE.

(\* Write Binary version of a BITSET called set \*)

```
FOR x:=7 TO 0 BY -1 DO
  bit:={}; INCL(bit, x);
  IF bit IN set THEN
    WriteChar('1')
  ELSE
    WriteChar('0')
  END
END
```

The following are some interesting equivalents that may take on a practical meaning at a later date:

UNION	$x \text{ IN } (s1 + s2) = (x \text{ IN } s1) \text{ OR } (x \text{ IN } s2)$
DIFFERENCE	$x \text{ IN } (s1 - s2) = (x \text{ IN } s1) \text{ AND NOT } (x \text{ IN } s2)$
INTERSECTION	$x \text{ IN } (s1 * s2) = (x \text{ IN } s1) \text{ AND } (x \text{ IN } s2)$
SYMMETRIC DIFFERENCE	$x \text{ IN } (s1 / s2) = (x \text{ IN } s1) \text{ <> } (x \text{ IN } s2)$

**7.1.36 INC**

```
PROCEDURE INC ( VAR x : <ordinal type> [; i : INTEGER] );
```

This is a proper procedure that will increment any ordinal type such as INTEGER, CARDINAL, CHAR or sub-ranges of. The optional second parameter will change the increment value.

```
    i:=25 ; INC ( i );           results in i = 26
    c:='C' ; INC ( c );         results in c = 'D'
    i:= 5 ; INC (i,6);          results in i = 11
```

**7.1.37 INCL**

```
PROCEDURE INCL ( set : <set type>; bit : INTEGER);
```

This SET procedure allows the inclusion of new members into a SET. Note that BITSET members are from 0 to 7 inclusive.

```
    VAR bit : BITSET;
    INCL( bit, 1 )
```

**7.1.38 INTEGER**

```
VAR <id list> : INTEGER;
```

INTEGER is a type of variable which specifies a whole signed number of 32-bits. It represents a value from -2147483648 to 2147483647. The INTEGER type requires 4 bytes of storage space.

Examples:

```
    VAR  i      : INTEGER;
        ia     : ARRAY [0..255] OF INTEGER;
```

### 7.1.39 LOOP

LOOP <statement list> END;

The LOOP statement, described more fully in a previous section, is a control structure which creates an infinite loop. The statements within the loop will be repeated over and over again. Saying that, it is possible to break out of the loop with the EXIT clause.

Example:

```
    LOOP
        INC(x);
        IF x>99 THEN EXIT END;
        WriteInt(x,0)
    END;
```

### 7.1.40 MAX

PROCEDURE MAX ( <type> ): INTEGER;

MAX is a function procedure that returns the maximum value of a particular type.

Example:

```
    MAX ( INTEGER )    returns 2147483647
```

### 7.1.41 MIN

PROCEDURE MIN ( <type> ): INTEGER;

MIN is a function procedure that returns the minimum value of a particular type.

Example:

```
    MIN ( INTEGER )    returns -2147483648
```

**7.1.42 MOD**

<numeric term> MOD <numeric term>

MOD is the numeric operator that performs the division of whole numbers. The result of the division is discarded and the remainder is returned. Division by zero will produce an arithmetic overflow. Division by negative numbers is not defined by the Modula standard but a truth table of what happens in this implementation is presented below. As can be seen the result is the same as for REAL numbers.

x	y	x MOD y
+	+	+
+	-	+
-	+	-
-	-	-

### 7.1.43 MODULE

```
MODULE <name>;  
<declarations>;
```

```
BEGIN  
    <statement list >  
END <name>.
```

MODULE is the defining word that is the start of a program. The program MODULE is completely self contained. When activated it claims its own memory for work space. This cannot be touched by any other program MODULE unless explicit instructions are given (see IMPORT). This is unlike the PROCEDURE that can use variables in surrounding PROCEDURES or from its own enclosing MODULE.

MODULEs are completely anonymous and several can reside in memory at any one time. The only inter communication is by IMPORTs. The IMPORTing MODULE must declare its IMPORTs at the start of the program and the exporting MODULE must have the same identifiers in its DEFINITION MODULE. When this has been achieved one MODULE can read/write another's variables and call its PROCEDURES. The exporting MODULEs are said to be library MODULEs and can be recognized by the word IMPLEMENTATION at the start. An IMPLEMENTATION MODULE always has a sister DEFINITION MODULE.

### 7.1.44 NIL

NIL is a pointer constant representing a unusable address. It is used for marking the end of lists. All de-allocated pointers are assigned the value NIL. The ordinal value of NIL in this system is 0.

**7.1.45 NOT or ~**

<boolean term> NOT | ~ <boolean term>

NOT or ~ is a unary BOOLEAN operator which negates the action of the logic.

Examples:

NOT TRUE	returns FALSE
NOT 5 = 6	returns TRUE

**7.1.46 ODD**

PROCEDURE ODD ( x : <ordinal type> ) : BOOLEAN;

ODD is a function procedure that returns a BOOLEAN value. Its parameter is an expression, of any ordinal type, that is either odd or even.

Examples:

ODD ( 89 )	returns TRUE
ODD ( 100 )	returns FALSE

CMS Modula-2 provides the extension function Even()

### 7.1.47 OF

```
CASE <expression> OF
    {<label> : <statement sequence> | }
    <label> :<statement sequence>
    [ ELSE <statement sequence> ]
END;
```

OF is a separator used in the CASE structure described previously.

Example:

```
    CASE x OF
        'a' : <statement list>
        |   'b' : <statement list>
    END;
```

### 7.1.48 OR

<boolean term> OR <boolean term>

OR is a BOOLEAN operator that requires two BOOLEAN values. It can be used in expressions that yield a BOOLEAN result.

Example:

```
    IF (x>99) OR (x<9) THEN
```

### 7.1.49 ORD

```
PROCEDURE ORD ( c : CHAR ) : INTEGER;
```

ORD is a function procedure that converts a CHAR type into an INTEGER type at run time.

Examples:

```
    ORD ( 'A' )    returns 65
    ORD ( 'c' )    returns 99
```

Note            x = ORD ( CHR ( x ) );

### 7.1.50 POINTER

```
VAR ptr : POINTER TO <type>
```

The POINTER TYPE is used to store the address of a variable of any other TYPE. Their main use is for accessing memory or memory mapped I/O devices. With I/O devices the POINTER is permanently set to the address of that device and the device can be read or written via the POINTER. When used with memory, dynamic variables can be created with data structures such as trees and linked lists. The Library MODULE Storage has been provided to aid in the use of POINTERS and contains PROCEDURES for claiming blocks of memory and for Allocating and De-allocating these blocks into smaller sections for the individual variables.

### 7.1.51 PROCEDURE

```
PROCEDURE <procedure name> [ ( <parameter list > ) ];  
[ TYPE { <type declarations> } ];  
[ CONST { <constant declarations> } ];  
[ VAR { <variable declarations> } ];  
{ <local procedure declarations> };  
  
BEGIN  
    <statement list>  
END <procedure name>;
```

The defining word PROCEDURE declares both proper and function PROCEDURES. A function PROCEDURE returns a simple variable type and can be used in expressions of the same type. Both proper and function PROCEDURES can take optional parameters which are declared in the PROCEDURE definition. When calling a PROCEDURE the number of parameters passed must be the same as that in the definition and the parameters must be of the same type. The parameters can be defined as value parameters or variable parameters. Value parameters are evaluated from the calling routine and act as a local variable within the PROCEDURE body. They can be said to pass into the PROCEDURE. Variable parameters use pointers to external variables. These can be modified by the action of the PROCEDURE. They can be said to pass into and out of the PROCEDURE.



### 7.1.52 REAL

VAR <id list> : REAL;

REAL is a type of variable which specifies a floating point signed number. This system uses the IEEE double precision number format of 64-bits. It represents a number with 17 significant decimal digits. The REAL type requires 8 bytes of storage space.

-1.7976931348623155e+308	minimum
1.7976931348623155e+308	maximum
2.2250738585072019e-308	smallest number

Examples:

```
VAR   r      : REAL;
      ra     : ARRAY [0..255] OF REAL;
```

### 7.1.53 REPEAT

REPEAT <statement list> UNTIL <testable expression>;

The REPEAT statement, described more fully in a previous section, is a control structure which creates a loop that is terminated by a condition at its end.

Example:

```
x := 1;
REPEAT
  WriteInt(x, 0);
  INC(x)
UNTIL x>10;
```

### 7.1.54 RETURN

RETURN [ <exp> ];

The statement RETURN is used within a function PROCEDURE to specify what is to be returned to the calling expression. The PROCEDURE is left after evaluating the return value. RETURN can also be used from a proper PROCEDURE, in which case no value is returned. If the RETURN statement is omitted from a function PROCEDURE the value returned will be undefined.

Examples:

```
PROCEDURE Times ( x,y : INTEGER): INTEGER;
BEGIN
    RETURN x*y
END Times;
```

Times( 4, 6 ) will return 24

```
PROCEDURE Count;
VAR x : INTEGER;
BEGIN
    x := 0;
    LOOP
        WriteInt(x,0);
        INC(x);
        IF x>10 THEN RETURN END
    END
END Count.
```

**7.1.55 SIZE**

PROCEDURE SIZE ( <type> ) : INTEGER;

SIZE is a function procedure that returns the number of bytes of storage used by a particular type.

Examples:

SIZE ( INTEGER )	returns 4
SIZE ( CHAR )	returns 1
SIZE ( WORD )	returns 2

Note that the minimum storage element in the Module system is 2 bytes so that single characters or BOOLEAN values will be stored as 2 bytes per value. This is not the case for an array where the data is packed. eg a BOOLEAN array would store 8 values per byte.

**7.1.56 THEN**

```
IF <testable expression> THEN <statement list>
  { ELSIF <testable expression> THEN <statement list> }
  [ ELSE <statement list> ]
END
```

THEN is a separator used in the IF structure to mark the end of the testable condition. A statement list will follow that will be executed if the expression is TRUE.

Example:

```
IF <testable condition> THEN <statement list> END;
```

### 7.1.57 TO

```
FOR <id>:= <start> TO <finish> [ BY <increment> ] DO
    <statement list >
END;
```

TO is a separator used in the FOR structure to isolate the start condition from the end condition. The expression following the TO is evaluated to form the end value for the FOR control variable.

Example:

```
FOR x:= start TO finish DO
```

### 7.1.58 TRUE

TRUE is the BOOLEAN constant representing NOT FALSE. In this implementation its ordinal value is -1.

Example:

```
flag := TRUE;          (* flag is BOOLEAN *)
IF flag THEN <statement list> END;
```

### 7.1.59 TRUNC

```
PROCEDURE TRUNC ( r : REAL ) : INTEGER;
```

Convert a real number into an integer with truncation if necessary.

Example:

```
TRUNC (123.45)      returns 123
```

**7.1.60 TYPE**

```
TYPE <id> = <type>; { <id> = <type> ;}
```

Types can be defined under the TYPE declaration and then subsequently used in variable definitions. TYPEs can be IMPORTed from other modules.

Example:

```
(* definition of a chess board *)
TYPE chessPieces = [1..6];
   columns      = [1..8];
   rows         = [1..8];
   chessBoards = ARRAY columns, rows OF chessPieces;

VAR x      : columns;
    y      : rows;
    board  : chessBoards;
```

**7.1.61 UNTIL**

```
REPEAT <statement list> UNTIL <testable expression>;
```

UNTIL is used at the end of a REPEAT structure. The expression following UNTIL is evaluated and if it is FALSE the statement list following the REPEAT statement is executed again.

Example:

```
x:=0;
REPEAT
    INC ( x );
    WriteInt( x,4 )
UNTIL x>10;
```

### 7.1.62 VAR

```
VAR <id list> : <type>;  
    {<id list> : <type>; }
```

VAR is used in the declaration section of MODULEs and PROCEDUREs. It is used to declare the names of the variable used in that MODULE or PROCEDURE. It is also used in a PROCEDURE declaration to specify a variable parameter type.

Examples:

```
VAR x, y : INTEGER;  
PROCEDURE name ( VAR z : INTEGER )
```

### 7.1.63 WHILE

```
WHILE <testable expression> DO <statement list> END;
```

The WHILE statement, described more fully in a previous section, is a control structure which creates a loop that is terminated by a condition at the beginning of the loop.

Example:

```
x := 1;  
WHILE x<10 DO  
    WriteInt(x,4);  
    INC(x)  
END;
```

**7.1.64 WORD**

VAR <id list> : WORD;

WORD is a system word which specifies a type of 16-bits or a word. It represents a value of from 0 to 65535. The WORD type requires 2 bytes of storage space.

WORD variables can only be assigned. There are no WORD operators.

Examples:

```
VAR  w      : WORD;  
     wa     : ARRAY [0..255] OF WORD;
```

## 7.2 Operators

### 7.2.1 Precedence

When evaluating an expression the various operators have different precedence as follows:

Group	Description	Operators
0	Parentheses	()
1	Unary	NOT ~ + -
2	Multiplication	* / DIV MOD AND & > <
3	Addition	+ - OR Eor
4	Relational	= > < # <> <= >=

The group with the highest priority is evaluated first. Operators in the same group are evaluated left to right. 1 is a higher priority to 2 etc.

Examples:

$2 + 3 * 4$	will return 14
$2 * 3 + 4$	will return 10
$(2 + 3) * 4$	will return 20

### 7.2.2 Unary

**+** **Plus**

Shows that a number is positive. This operator has no effect.

Example:

$+123$  @TAB TEXT =

**-** **Minus**

Indicates a negative number.

Example:

$-123$

~                    **logical complement**

This is a synonym of NOT

Example:

```
IF ~ bool THEN <state1> ELSE <state2> END
```

The ~ operator reverses the logic of the BOOLEAN value bool.

### 7.2.3 Binary

Arithmetic operators

+	addition
-	subtraction
*	multiplication
DIV	integer division
MOD	remainder

The following shows the relationship between MOD and DIV

$$x = (x \text{ DIV } y) * y + (x \text{ MOD } y)$$

Boolean

These operate on two BOOLEAN values to produce a BOOLEAN result.

AND	logical AND
&	synonym of AND
OR	logical OR

Truth table for OR and AND

a	b	a AND b	a OR b
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

CMS Modula-2 also adds the Eor logical operator.

### Relationship

<	less than
<=	less than or equal to
<>	not equal to
#	synonym of <>
=	equal to
>	greater than
>=	greater than or equal to

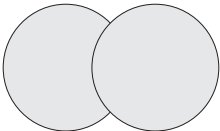
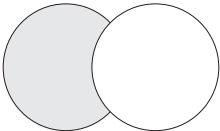
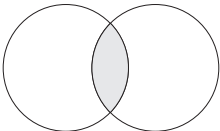
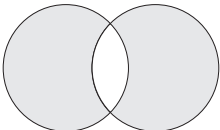
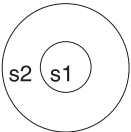
The relationship operators compare two similar types to produce a BOOLEAN result. They are frequently used as testable conditions in the IF, UNTIL and WHILE structures.

Examples:

```
IF x>10 THEN WriteString('Greater than 10') END
WHILE x#y DO <statement list> END
```

### 7.2.4 Set

+	union
-	difference
*	intersection
/	symmetric difference
=	equality
<> #	non-equality
IN	membership

		SET	Logical Equivalent
		s1    s2	
+	Union		s1 OR s2
-	Difference		s1 AND ~s2
*	Intersection		s1 AND s2
/	Symmetric Difference		s1 EOR s2
IN			s1 = s1 AND s2

### 7.3 Punctuation

“<text>” or ‘<text>’

#### Quotes

Character constants or string constants must be specified in between two similar quotes. The choice of which type of quote, single or double, allows the other type to be used within.

Examples:

```
chr := 'a';  
line := 'Hello World';  
text := "Modula-2 is a 'VERY' good Language";
```

#### ( ) expression brackets or procedure parameters

Brackets can change the order an expression is evaluated. The values in the brackets will be worked out first.

```
x := (2 + 3) * 4;
```

Brackets are also used to hold the parameters that are passed to procedures.

```
sale ( x )
```

#### , comma parameter separator

When a procedure has more than one parameter commas are used to separate the items. The same is true of CASE label lists.

```
PROCEDURE lift ( x, y : INTEGER );  
WriteInt ( x , 4 );
```

#### .. sub-range specifier

```
VAR   x       : [ 1 .. 99 ];  
      line    : ARRAY [ 0..15 ] OF CHAR;
```

#### : declaration of variables

---

```
VAR x, y : INTEGER;
```

**and CASE label separator**

```
'a' : <statement list> |
```

```
'd'..'z' : <statement list>
```

**:= assignment of variables**

```
x := 56;          (* x becomes 56 *)
```

```
x := x + 1;      (* x becomes x+1 *)
```

**; statement separator**

```
x := 1 ; y := 2 ; z := 3
```

**[ ] array braces**

```
ary [ 5 ] := 56;
```

```
x := ary [ 5 ];
```

**(\* \*) comment braces**

(\* anything written between the comment braces will be removed from the source text before compilation. The characters must be grouped as pairs. Comments can be freely nested \*)

**{ } Set braces**

Set braces for defining set constants.

```
bits := { 0, 2, 4, 6 }; (* even set *)
```

**| CASE statement separator**

```
'a' : <statement list> | 'b' : <statement list>
```

## 7.4 CMS Extensions

### 7.4.1 Indirection

?	byte indirection
??	word indirection
!	Integer indirection

These are here for BBC BASIC fans. The recommended method is to use the Modula POINTER type described elsewhere. given that i is of type INTEGER, w is of type WORD and b is of type BYTE then:

```
i = !( ADR (i) )
w = ??( ADR (w) )
b = ?( ADR (b) )
```

These are dangerous operators, as they can write to absolute memory.

```
!( 1000h ) := 12345678h;
```

will write to 1000h address in memory.

### 7.4.2 Bit Shift

<<	shift left
>>	shift right

Binary shift left or right. These can perform extremely fast multiplication or division by powers of 2.

```
x := x<< 3      =    x := x * 8;
y := y >> 4;    =    y := y DIV 16;
```

## 7.5 System Key Words

These words are built into the system and are available without importing. They should be used with care if the program is to be kept portable.

### 7.5.1 BatRam

This is a system variable that holds the start of the unused battery backed storage area in RAM. It is read only. This area is guaranteed untouched by the system or language and can be used for user variables that will survive power on/off sequences.

### 7.5.2 BatRamTop

This is a system variable that holds the end of the untouched battery backed RAM. It is read only.

### 7.5.3 BufSize

When a program is compiled a compiler buffer is claimed from the operating system. When a successful compilation is achieved the buffer is trimmed down to the required size and the remaining space is given back to the system. The current Compiler buffer size can be changed with BufSize. It defaults to 8K after a reset. An error 'Compiler Buffer Overflow' will occur if a program gets too large during compilation.

```
BufSize := 4000h;          (* change to 16K *)
```

### 7.5.4 C

C <name>

'C' is for compile and will compile a Modula source file to produce a MODULE in memory. This can then be run or saved on disc for future use.

### **7.5.5 Continue**

Continue can be used to resume program execution after the 'Debug' statement. This can be abbreviated by just pressing the carriage return key.

### **7.5.6 Debug**

The Debug statement will cause the program to temporarily halt execution. All the program state is saved and the system will prompt for action. The prompt will actually be changed and will show the name of the PROCEDURE or MODULE that the debug is in. The actions that can be performed from the new prompt are the same as that of the normal prompt. The rules of visibility apply within the PROCEDURE. This means that the local variables can be examined or changed as part of the debug process. To continue the program type Continue or just press the carriage return. The Debug statement can be used within a conditional IF statement so that conditional Debug can be used.

Example:

```
IF dbg1 THEN Debug END;
```

### 7.5.7 Eor

<boolean term> Eor <boolean term>

Eor is a BOOLEAN operator that requires two BOOLEAN values. It can be used in expressions that yield a BOOLEAN result.

Example:

```
IF (x>9) Eor (x<99) THEN
```

Truth Table Eor

a	b	a Eor b
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

### 7.5.8 ErrMax

ErrMax is a system variable that contains the number of error messages that will be reported by the compiler. It has a default value of 2.

```
ErrMax := 8 (* change to 8 errors *)
```

### 7.5.9 Escape

```
PROCEDURE Escape( onoff : BOOLEAN)
```

The Escape procedure enables or disables the effect of the escape character on a running program. By default a program can be halted when it receives an escape on its terminal port. Escape takes a BOOLEAN parameter.

```
Escape ( FALSE) (* Turn off escape effect *)  
Escape ( TRUE ) (* back on *)
```

### 7.5.10 Even

PROCEDURE Even ( x ) : BOOLEAN;

Even is a function procedure that returns a BOOLEAN value. Its parameter is an expression that is either odd or even.

Even ( 89 )	returns FALSE
Even ( 100 )	returns TRUE

### 7.5.11 Heap

This is a system variable that holds the start of the Heap storage area in RAM. It is read only. The Heap is the area that follows the program storage area and is used for the global variable storage of MODULEs.

### 7.5.12 HeapTop

This is a system variable that holds the end of the Heap storage area in RAM. It is a read only variable. The memory usage of the Heap is a constant for any given program.

### 7.5.13 Help

Help will list all of the key words and operators in the system.

### 7.5.14 List

List [ name ]

List will display a table of MODULEs that are currently held in memory whether it be EPROM or RAM. By specifying a MODULE name as a parameter, List can also be used to show what is available from any library module in memory. It Lists the identifiers that are available for IMPORT from other modules. This is a quick, convenient method. If more information is required the .DEF file can be inspected.

### 7.5.15 Load

Load <file name [.ext] >

Load is used to load previously saved program from disc into memory. The system will first search the current directory and then, if unsuccessful, the \REL directory. The terminal is intelligent and the file load uses embedded control codes to pass the data. The word 'Load' will only work with the Target terminal program supplied with this system.

Load will also load system drivers as well as Modula-2 programs.

e.g. Loading the extension serial boards device descriptors

Load MINOS\VS0

### 7.5.16 Lock

Lock [ <name> ]

Typing the word Lock will place an imaginary barrier around all of the programs currently stored in the RAM. These will be preserved by the battery back-up circuitry during a power down or after a reset. This solves the problem of having to download all of the MODULEs and MODULE libraries in a given project at every test session. Only the MODULE that is currently being developed needs to be repeatedly re-compiled. Re-compiling a 'Locked' module will fail and give an error message. The old module must be UnLocked first.

### 7.5.17 Mdir

The Mdir command will display all of the system modules that are in the Modules memory. This includes both EPROM and RAM memory. Each entry is followed by an E or R (EPROM or RAM) and a digit which is the system module type:

0	Machine code program
1	Configuration table for a device
2	Driver
3	special system parts
4	Modula-2 MODULEs
5	Data

**7.5.18 M2**

M2 <name> [ <name> ]

M2, for Modula 2, will compile a source program and then, if there are no errors, save the resultant code. M2 can optionally take multiple file names. This is very useful when a library has been edited and all the dependent MODULEs will require re-compiling.

**7.5.19 New**

New will clear all of the memory of programs except those which are locked.

**7.5.20 Program**

This is a system variable that holds the start of the program storage area in RAM. It is read only.

**7.5.21 Quit**

Quit when executed drops into the 68000 debug monitor. It passes all the 68000s registers which can be displayed. The program can be traced or break points can be added. A common use of debug is to set a break point in a machine code routine under test. Pressing 'g' from the monitor will continue after the Quit instruction as long as non of the registers have been changed.

e.g. Testing a machine code procedure called 'Mac'.

```
WriteAdr ( ADR( Mac ), 8 ); WriteLn; Debug;  
Quit;  
Mac;
```

### 7.5.22 Rnd

PROCEDURE Rnd ( x : INTEGER ) : INTEGER;

A random number is returned from the function Rnd. The number generator uses a 40 bit algorithm and returns the next number in the sequence to Rnd. Rnd returns an INTEGER which will be from 0 to the value of the parameter passed.

Example:

```
      LOOP
          Tab ( Rnd(78), Rnd(23) );
          WriteChar('*')
      END;
```

### 7.5.23 Save

Save <name>

'Save' will save a MODULE to disc. This can be subsequently loaded with 'Load'. All IMPORTed MODULEs must have previously have been saved. The 'M2' command automatically saves a program after compilation.

**7.5.24 Sgn**

```
PROCEDURE Sgn ( x : INTEGER ) : INTEGER;  
PROCEDURE Sgn ( x : REAL ) : REAL;
```

The function Sgn return the sign of the number passed to it. The result is of value 1 for positive and -1 for negative.

Example:

Sgn( 100 )	returns 1
Sgn( -23 )	returns -1
Sgn ( -123.8 )	returns -1.0

Note:

A REAL constant must have a '.' or an 'E' in the word otherwise it will be treated as INTEGER.

**7.5.25 Stack**

This is a system variable that holds the lowest value that the stack reached during the execution of a program. It is read only.

**7.5.26 StackTop**

This is a system variable that holds the start of the stack storage area in RAM. It is read only. The stack starts at the top of a block of claimed RAM memory and extends downwards. It is used by procedures to hold the local variables and parameters. To that effect it moves up and down as the programs run depending on the individual requirements of each procedure. The stack also contains intermingled with the procedures data, the normal 68000 CPUs stack information. DO NOT TOUCH THE STACK!

### **7.5.27 StackSize**

When a new process is created it will have an amount of stack given to it. This starts off at 4K and can be changed if required. Most programs only require 1K so that if many co-processes are required the stack can be changed to suit. The minimum size is 1K. For example 20 copies of the program 'Demo' can be run concurrently before a memory overflow. With the stack at 1K this goes up to 45. Things are getting pretty slow at this point! The program 'SysMap' is very good at displaying the used and unused memory in the Module.

```
C Demo
StackSize := 400h
Demo &
```

### **7.5.28 StartProcess**

```
PROCEDURE StartProcess ( name : ARRAY OF CHAR ): INTEGER
```

This built in PROCEDURE will start running the PROCEDURE name as a co-process.

### **7.5.29 Symbol**

This is included for interest only and gives the storage address of the compilers symbol table.

### 7.5.30 Trace

A program can be traced statement by statement. The Trace output will be to the terminal and will start when it comes across a Trace (TRUE) statement in the program. The 'TRUE' can be a BOOLEAN variable so that the trace can be made conditional. The Trace will not follow procedure calls but will continue on the return. The Trace will stop at the end of the MODULE or PROCEDURE it started in. The Trace output displays where the trace comes on and the name of the statement or identifier at the start of each new statement.

```
MODULE Loop;
VAR   x : INTEGER;
BEGIN
    Trace(TRUE);
    FOR x:=1 TO 4 DO
    END
END Loop.

Loop
```

### 7.5.31 TurnKey

Turnkey <string>

Turnkey is the magical word that takes a string parameter and stores it in the battery backed RAM. When the system is reset, or from power on, the TurnKey string will be executed. Anything can be written to the TurnKey line and it will appear as if it was being typed at reset. This is very useful for starting a program after power on. Remember that a program must be Locked before it will be saved in the battery backed RAM. A secondary effect of writing a TurnKey line is that the normal output message will be inhibited after reset. The command produces a special system data module and battery backs it in RAM. On reset the system looks for the presence of this data module, now called TurnKey, in the Eprom and RAM. If it is present it will execute its contents. In Eprom applications the TurnKey module must be Saved and then included in the list of modules for the Eprom.

### 7.5.32 UnLock

UnLock [ <name> ]

UnLock removes all data from the battery backed RAM. Use with care. An old MODULE will be deleted whenever a new one of the same name is compiled. If a MODULE is Loaded it will not replace an existing MODULE. Only Compiling a New MODULE will replace the old one. There is a quick trick to delete an old MODULE, although it will probably never be required.

(\* Delete MODULE old from the directory \*)

Lock

UnLock old

MODULE old; END

New

## 8 Co-routines

There are three methods of starting up co-routines or co-processes from the language.

### 8.1 PROCEDURES

The first to be described is the traditional Modula-2 method. Here any of a MODULEs PROCEDUREs may be run as a co-process. These procedures are identical to normal procedures so that they have access to any of the MODULEs global variables. In this way the co-routines can communicate. Reading and writing INTEGERS, CARDINALs, ADDRESSes, POINTERs, WORDs, BYTEs, BITSETs and CHARs requires no special attention as they are non interruptable. Care is required when dealing with ARRAYs or REAL numbers when a semaphore system will have to be employed.

The program on the evaluation disc shows an example of multi-tasking and is called 'Task'. This also shows how signals work.

### 8.2 Built in functions

#### **StartProcess( name : ARRAY OF CHAR ) : INTEGER;**

Starts up a co-routine PROCEDURE called name. This returns a unique process identifier.

PROCEDUREs from System.a library.

#### **PROCEDURE Wait(): INTEGER;**

This command causes the Co-routine to give up its time slot and wait for a signal. When received the program will resume after the wait instruction. The function returns a message value.

#### **PROCEDURE Signal( processId, message : INTEGER );**

This can signal an awaiting process to re-commence. It must pass the process id of the waiting process and a message. If a message is not required it can be set to 0.

### **8.3 MODULEs**

The second form of Co-processing is done at the MODULE level. Here further MODULEs can be run under the systems multi-tasking kernel. The MODULEs will be completely independent and have their own work-space. In fact the same MODULE can be run more than once with each having its own work-space. There is no way these MODULEs can communicate with each other. They are completely independent. The interesting thing here is that they are still using the languages common resources so that there is only one copy of the language environment in memory.

To achieve this form of multi-tasking just add an & to the end of the MODULEs name. This can be done from the command line or from within another MODULE.

Demo &

Note that MODULEs cannot be run from another MODULE without the background &.

### **8.4 Machine Code**

The last form of multi-tasking is the starting up of a completely independent machine code program. This must be in the memory in 'program module' format. In fact a second copy of the language can be started up on another serial port to give a two user Module. To do this another system PROCEDURE is used. The program 'User' on the evaluation disc will start up a second copy of the language on the CPU serial port.

## 9 Assembly Code

The following is a specification of how to call 68000 machine code programs from the Modula-2 language. The calls, when used from a Modula-2 program, will look exactly the same as a normal language procedure with a name to identify the call and with optional parameters.

### 9.1 Introduction

The system requires two files to be prepared. The first is called the definition file and the second the assembly file. These are both text files. The assembly file is assembled to produce a binary file. This will contain the machine code routines and some extra labels and offsets at the start to enable it to be linked with the definition file.

Name.DEF definition file  
Name.ASM assembly source  
Assemble Name.ASM to produce Name.BIN

The .DEF and .BIN extensions are obligatory but the .ASM can be changed to suit the assembler being used. When Loading into the Module, the system will first look into the current directory for the file Name.DEF. If it is not present it will search the ASM sub-directory. It is recommended for tidiness to keep all the assembly files in the ASM sub-directory. The same commands are used for machine code files that are used for language files (e.g. C, M2, Load and Save). The system will first search for Program or Library files. If this fails it will then look for the assembly files. Hence the use of the file extensions. Once compiled the machine code procedures can be saved as a single Module for future use. This behaves the same as a normal library module.

The M2 command can be used as before to compile and save in one command, e.g. M2 Name. The resultant file is stored in the REL sub-directory and it is this file that is used by the IMPORT statement. This file can also be included in the EPROM to become a resident library or even the main program.

## 9.2 File Search Order

When a C (Compile) or M2 (Compile and Save) command is entered the following shows the file search order:

Directory	Extension
MODULE	.MOD
MODULE\LIB	.MOD and .DEF
MODULE\ASM	.DEF and .BIN

File not found

Files are Saved and Loaded from the REL sub-directory.

## 9.3 Implementation

### 9.3.1 Definition File

First prepare the Definition file. This is a formal specification of the Procedure. This is exactly the same as for a normal DEFINITION file as for a Modula-2 program.

```
DEFINITION MODULE Port;
VAR   space : WORD;      (* space can be IMPORTed *)

(* Write to a port *)
PROCEDURE WritePort( port : [0..7]; value : [0..255]);

(* Read from a port *)
PROCEDURE ReadPort( port : [0..7]; VAR value : INTEGER );

(* No code block in definition modules *)

END Port.
```

### 9.3.2 Assembly File

The assembly file is written as normal and a section is required at the beginning to allow the compiler to attach the correct addresses to the appropriate routines. The code produced must be position independent and have no absolute references. The system is dynamic and the code must be capable of running anywhere in memory, including EPROM. The following example is written to work with the above Definition file. The full file can be found in the MODULE\ASM sub-directory called ADIO.A. The order of the routines is irrelevant.

```

    dc.b   'WritePort',0
    align
    dc.l   WritePort-Start

    dc.b   'ReadPort',0
    align
    dc.l   ReadPort-Start

    dc.b   'Port',0
    align
    dc.l   Port-Start

    dc.w   0           list terminator
    dc.l   2           Heap space required

Start

WritePort   .           *           68000 code
            .
            .
            rts

ReadPort    .           *           68000 code
            rts

Port        .           *           68000 code Module initialisation
            rts

```

Note:

'Port' will be executed before the start of the main module that IMPORTed 'Port'. This initialisation routine must always be present although 'rts' on its own will suffice. The number of bytes claimed from the heap is the last entry on the table and is 2 in the above example.

The above example may be different on other assemblers. The declarations use dc.b and dc.l. These are define constants, 'b' for bytes and 'l' for long words. 'align' adds an extra \$00 if the string was odd in length. This produces a table that the compiler will use to reference the start of the procedures.

### 9.3.3 Assembly Only Programs

An assembly program can be the main module if its initialisation section is in fact the main program. As such the saved program can be loaded and run in the same way as a Modula-2 program:

Load Name

Name

## 9.4 Parameter passing

The Modula-2 compiler will check that the programs calling the machine code match the specification of the Definition files. At run time the language will prepare any necessary parameters ready for the user machine code programs. The routines are called with the following register definitions.

a7	stack pointer
a6	parameter pointer
a4	heap pointer
d0	used to return function results a7 must be preserved all other registers can be used.

### 9.4.1 Heap Pointer

a4 is the heap pointer. This points at the RAM space that was reserved for the VAR statements in the definition file. The amount of heap space was specified at the end of the name table in the assembly file. In the above example two bytes of heap were allocated. This is now reserved for this machine code module and will not be touched by any other program unless it has been specifically IMPORTed. The Heap is valid for the whole lifetime of the program. For example a random number seed could be kept there. \* for the above definition

```
move.w    (a4),d0      read heap
move.w    d0,(a4)     write heap
```

The order the compiler assigns heap space is the same order that it appears in the definition file.

Example:

```
VAR  x,y :  INTEGER;
     a :   ARRAY [0..15] OF CHAR;

     0(a4)      x
     4(a4)      y
     8(a4)      a
```

total heap used 24 bytes

```
dc.l    24          in assembler file
```

Caution:

Do not use the heap if it has not been declared in the assembly file and do not use any more or you will write on another modules heap space!

### 9.4.2 Parameter Pointer

a6 is the parameter pointer. Parameters are placed below a6 in descending order.

#### Value Parameters

For the above example WritePort:

```
a6  >>      .
      -1      ls      port INTEGER 4-bytes
      -2      .
      -3      .
      -4      ms
      -5      ls      value INTEGER 4-Bytes
      -6      .
      -7      .
      -8      ms
```

\* pre-decrement addressing mode

```
      move.l   -(a6),d0      port
      move.l   -(a6),d1      value
```

or

\* offset addressing mode

```
      move.l   -4(a6),d0     port
      move.l   -8(a6),d1     value
```

#### VARiable Parameters

Variable parameter (so called VAR-parameters) pass the address of the variable rather than the value.

For the above example ReadPort:

```
a6  >>      .
      -1      ls      port INTEGER 4-bytes
      -2      .
      -3      .
```

```

-4      ms
-5      ls      variable address 4-Bytes
-6      .
-7      .
-8      ms

```

To Write the result of the routine back to the variable is simply achieved by using the address.

```

move.l   -(a6),d0      port
move.l   -(a6),a0      address of variable
.
.                      get port value to d1
move.l   d1,(a0)       write to variable

```

### Arrays

Both normal and Open arrays pass the same form of parameters. Value and Variables appear the same. All of the work has been done already by the language. A value array will have been copied into local work-space. A Variable array is the original copy as normal. Consider the following definition:

```

TYPE array = ARRAY [0..7] OF CHAR;
PROCEDURE Name(x : array);

```

This would produce the following

```

a6    >>      .
      07      number of elements - 1
      00
      ls      address of array start 4-bytes
      .
      .
      ms

```

The following Open definition would produce the same parameters as above but the length would reflect the calling array.

```

PROCEDURE Name( x : ARRAY OF CHAR );

```

### 9.4.3 Type storage requirements

The following tables specifies the storage requirements for each type of variable.

TYPE	Bytes
INTEGER	4
CARDINAL	4
POINTER	4
WORD	2
BYTE	2
CHAR	2
BITSET	2
BOOLEAN	2
ARRAY	(TYPE)*(no. of elements)

It will immediately be noticed that the byte types actually use 2 bytes of storage space. This is because the 68000 uses an even stack and can only push and pull words (2-bytes). When used in arrays byte and boolean types are packed so that program storage is efficient. This poses the question which of the two bytes is the actual byte that is required and which bit of the 16-bits is the boolean value. The following examples serve to illustrate this.

#### Bytes

PROCEDURE Name( b,c : CHAR );

```
a6    >>      .
      00      undefined
      bb      character
      00      undefined
      cc      character
```

```
      move.b  -(a6),d0
      move.b  -(a6),d1
```

or

```
      move.b  -2(a6),d0
      move.b  -4(a6),d1
```

**Boolean**

PROCEDURE Name ( b : BOOLEAN );

```

a6    >>      .
      00      undefined
      01      boolean

      move.b   -(a6),d0
      btst    #0,d0      lsb of byte
or
      move.w   -(a6),d0
      btst    #8,d0

```

**Array of Integer**

PROCEDURE Name ( ary : ARRAY OF INTEGER );

The address and length of the array is obtained as described above.

```

      move.w   -(a6),d0      number of elements
      move.l   -(a6),a0      address of the first
      ls      n
      .
      .
      ms      n
      .
      .
      .
      ls      1
      .
      .
      ms      1
      ls      0
      .
      .
a0    >>      ms      0

```

```

        move.l    (a0),d1    first element
        move.l    4(a0),d2   second element
etc

```

### Array of byte

As above but the storage is in consecutive bytes.

```

        move.b    (a0),d1    first element
        move.b    1(a0),d2   second element
etc

```

### Array of boolean

Boolean arrays are stored as an array of bytes where each byte holds 8 boolean values. The first bit of the array is the least significant bit of the first byte.

PROCEDURE Bool ( b : ARRAY OF BOOLEAN );

```

        move.l    -6(a6),a0   start
        btst     #0,(a0)     bit 0
        btst     #7,(a0)     bit 7
        btst     #0,1(a0)    bit 8

```

Routine to test the nth bit:

```

entry:
        d0      bit
exit:
        N      equal or not-equal

tstbit  move.l   -6(a6),a0     array start
        move.l   d0,d1
        lsr.l    #3,d1        byte position
        andi.l   #7,d0        bit position
        tst.b    d0,(a0,d1.l)
        rts

```

## 9.5 Debugging

A machine code program can be debugged using the machine code debugger that is included in the system. Two things are required to start debugging. Firstly, where has the machine code been stored in memory, and secondly, how do we invoke the debugger. The address of a machine code procedure can be found by using the 'ADR' function in Modula-2.

```
MODULE GetStart;
FROM Machine IMPORT Code;
FROM Terminal IMPORT WriteAdr, WriteLn;
BEGIN
    WriteAdr( ADR( Code ), 0 );
    WriteLn
END GetStart.
```

After running this module the machine code library 'Machine' will be Loaded into memory and the address of its routine 'Code' will be displayed. This can now be 'Locked' so that its position in memory is preserved. Now the application program can be compiled which makes a call to 'Code'. Invoking the debugger is simply achieved by executing a Quit instruction or by pressing the 'ABORT' button on the front of the evaluation board. (Note: Evaluations boards below Issue 4 used the 'IRQ 8' button. The issue number can be found in the bottom left hand corner and is the last 2 digits of the board code). The debugger can now be used to place a break point or dis-assemble the code from the start address of the 'Code' procedure. Typing 'g' will return to the language exactly where it left off.

### Caution:

Do not change any of the registers at this point or it will not return correctly!

When the application program is run it will now stop at the break point and enter the debugger. Normal debugging can now take place. If an error occurs and the reset button has to be pressed the break point may still be loaded. In which case the modules check-sum will be wrong and it will not appear in the module directory shown by 'List'. If this is the case the module 'GetStart' will have to be compiled, run and locked once more.

This page is intentionally left blank

## 10 The Debugger

### 10.1 Introduction

The Minos debugger allows programs designed to run in a Minos system to be tested at the 68000 assembler level. It also provides a number of commands for displaying and changing memory which are very useful for checking out new pieces of custom built hardware. The debugger operates in either command mode or run mode. In command mode any of the commands may be used except those directly associated with program execution like trace, goto, and register. In run mode all of the debugger commands are available and the stored registers stack pointers etc. are all maintained for the program under test so that it can be traced or executed up to a break point etc..

### 10.2 Using the Debugger

The Minos debugger is not actually a program in its own right and you cant run the debugger in the same way as you run your application program. The debugger functions are called as if they were functions within your program or they are called by the operating system in response to a 68000 exception (like trace or illegal instruction etc.). When a Minos system is reset the debugger is initialized in command mode, commands can be sent to the debugger from the shell prompt by preceding the debugger command with a d character for example you can display the start of the EPROM memory by typing

```
d      dump      0
```

Or disassemble the start of the system code by typing

```
d      dis       30c
```

The shell program simply calls the appropriate debugger function then returns to its command prompt. The debugger itself has a command entry function which allows commands to be typed in without the preceding d. This mode is entered by typing a d on its own from the shell command line. At this point the prompt will change showing that you are at the debuggers command entry level.

Note the debugger actually runs with the 68000 interrupt mask raised to 7 to allow it to be used to debug interrupt service routines but this means that the Minos task swapping and the normal interrupt driven IO will be suspended while debug functions are being performed.

Debugging a user program is simply a case of downloading the program in the normal way using the load command then setting a breakpoint somewhere in the program using the debugger from the shell command line e.g.

```
d      break      180042
```

The actual address for the breakpoint will of course depend on the program and where in memory it has been loaded. Using the symbol table information provided by the compiler will allow you to use function names instead of addresses. Now run the program in the normal way by typing its name. The program will start to run as normal but when the breakpoint is encountered the debugger will be called, copies of the stack pointers and registers will be saved and the debugger's command entry function is called with the debugger in run mode ready to trace or run the rest of your application. At this stage the register store reflects the state of the registers just before the breakpoint.

### 10.3 Abbreviating Commands

Each of the Minos debugger functions has a meaningful word associated with it i.e. dump for dumping out an area of memory change for changing the contents of memory. This makes the commands easier to remember but it does make for rather a lot of typing especially for commands like disassemble. To overcome this the debugger will recognize commands by the first few letters i.e. dump can be replaced by a single d, change by a single c etc.. In some cases where the first letter is the same a slightly longer abbreviation must be used e.g. disassemble can be shortened to di. The detailed command descriptions give the minimum abbreviation allowed for each command.

#### **10.4 Data Sizes**

The Minos debugger does all of its memory specific functions on a byte by byte basis i.e. if you display a block of memory it will be shown with 16 bytes across a line. If you try to change a location you can only affect the byte at the given address etc.. This is not always the most convenient size especially when dealing with high level programs where different types have different lengths. In order to allow data to be displayed in a more appropriate way the debuggers data size can be changed using the byte, word, or long commands. After one of these commands has been used subsequent use of commands like dump and change will display and change 8, 16, or 32 bits at a time.

#### **10.5 Volatile Registers**

When the change command is used to check and manipulate new hardware devices it will read and display the current value of the register before allowing you to change it. This can be a problem with devices where the read and write function of a register are different. This is particularly important with devices such as UARTs where the transmitter and receiver registers are in the same place and the reading of the receiver affects other status registers. In order to prevent this problem it is possible to disable the reading of a location before allowing you to change its value. This is done with the commands noread to disable it and read to enable it again.

#### **10.6 Odd And Even Addresses**

Many of the peripherals used with 68000 systems only have 8 data lines, many system designers connect these lines to the lower data lines of the 68000 and leave the higher lines alone this works well but it means that the internal registers of the peripheral only appear at odd addresses in the memory map. The debuggers odd command will force any subsequent use of the change command to only look at odd addresses and to move forward and backwards by two each time.

## 10.7 Options

In some debugging tasks especially when new hardware is involved it is possible to get a situation where the size of data being worked on and the need for odd addressing is constantly changing. In order to save continually changing these modes a number of commands can be used as options in which case their effect only lasts for the duration of that command. When they are being used to temporarily change the mode of operation the modifying command should be placed directly after the main command i.e. if the debugger is working with byte sized data you can display some memory as 32 bit numbers by typing

```
dump long [address]
```

or change some peripheral registers by typing

```
change odd [address]
```

Note All the normal command abbreviations work with options as well.

## 10.8 Debugger Command Summary

### 10.8.1 BREAK

syntax:           BREAK [address]

abbreviation:    B

purpose:          Display or set break-points.

This command is used to set a break-point or to display all the break-points which are currently set. There can be up to 8 break-points set. If the address is omitted the addresses of all the currently set break-points are listed. If no break-points are set the message No breaks set is displayed. If an address is specified it is added to the list of currently set break-points. When a break-point is set the instruction to which the address points is saved but the break-points are only inserted into the applications program after a GOTO command. When a break-point is executed the registers from the application program are saved and all of the break-points are removed before returning to the Debugger. At this stage the applications program is intact. i.e. there are no break-points in it and the register table is set ready to execute the instruction which was replaced by the break-point. If the break-point is removed the program will continue as normal. If it is not, a TRACE instruction will be required to move passed this instruction. The TRACE will leave the register table ready to execute the program as normal.

Note:

The break-points are actually implemented with an illegal instruction so all the 68000 software traps are available to the applications program.

### **10.8.2 BYTE**

syntax:            BYTE

abbreviation:    BY

purpose:           Set the operating data size to bytes.

This command is used to change the data size used by some of the memory commands like dump and change to 8 bits. Byte can be used as a command on its own in which case it affects all subsequent memory commands or it can be used as an option in which case it only affects the current command.

### 10.8.3 CHANGE

syntax:           CHANGE [<options>] [<address>]  
abbreviation:    C  
purpose:          Examine and change the contents of memory.

This command is used to change the contents of memory. The address and current contents of the memory are displayed before waiting for an input line which is used to change the contents of the memory. If the input line is blank, just a carriage return, the address is incremented by an appropriate amount and the next address is displayed. If the line starts with a single minus sign the address is decremented by an appropriate amount and the previous memory location is displayed. If the line starts with any other character except a double quote mark the line is assumed to contain numbers which are to be inserted starting at the current address. The numbers may be either straight numbers or expressions as described earlier. Numbers which are smaller than the current data size will be padded out with leading zeros while numbers which are larger are truncated. When the end of line is reached the Debugger will return with the next address to be changed. If a bad number is encountered an error message is printed and the command is terminated. If the input line starts with a double quote it is assumed to be an ASCII string and the characters will be inserted into the memory without being changed until another double quote mark is encountered when the Debugger will return with the next byte to be changed. To insert a double quote into memory put two adjacent double quotes on the input line.

#### Note

The string option will only work if the Debugger is in BYTE mode. The CHANGE can be stopped at any time with the ESC key.

#### **10.8.4 COMPARE**

syntax: COMPARE <address 1> < address 2> <length>

abbreviation: CO

purpose: Compare two blocks of memory.

This command compares one area of memory with another. Bytes starting at address 1 are compared to the corresponding byte starting at address 2 for length number of bytes.

#### **10.8.5 DISASSEMBLE**

syntax: DISASSEMBLE [address]

abbreviation: DI

purpose: Disassemble object code back to mnemonics.

This command prints out a page of addresses followed by the mnemonic for the instruction at that address. All branch offsets are calculated and displayed as the absolute address to which they point. If no address is specified the disassembly starts at the current address of the stored copy of the P.C. register. If the debugger has been given a symbol table they addresses will be displayed in function name + offset format.

**Note:**

As the disassembler works by decoding the micro-code of the instructions it will always have a go at disassembling even if the instruction turns out to be illegal. This can cause it to print out some slightly strange mnemonics when it passes over areas of data rather than instructions.

**10.8.6 DUMP**

syntax: DUMP [<options>] <address>

abbreviation: D

purpose: Display memory in hex and ASCII.

A page of memory is displayed with each line starting with an address printed as a 24-bit hex number followed by either 16 bytes, 8 words or 4 long words, printed in hex and separated by spaces. Each line ends with 16 bytes printed as their ASCII equivalent. If a byte is outside the normal printable range it is replaced with a full stop. The command will stop after 24 lines have been displayed. The next page will be displayed after any key is pressed. The command is terminated by pressing the ESC key.

### 10.8.7 FILL

syntax: FILL [<options>] <start> <end> <data>

abbreviation: F

purpose: Fill a block of memory with data.

This command fills the block of memory from the start address to end address minus one with the specified data. The fill command will pad data which is less than the current size with zeros. The start address will be rounded up to the next even address if in word or long mode. Likewise, the end address will be rounded down. NB filling areas of memory can be dangerous. Make sure that the area you are going to fill is not being used by something else.

### 10.8.8 FIND

syntax: FIND <address 1> <address 2> <data>

abbreviation: FIN

purpose: Find a data pattern in a block of memory.

This command is used to find all the occurrences of a data pattern between two addresses. The search data can be specified either as a list of numbers or expressions or as an ASCII string enclosed by double quote marks and can be up to 256 bytes.

**10.8.9 GOTO**

syntax: GOTO [<address>]

abbreviation: G

purpose: Start executing the applications program.

This command is used to restart execution of an applications program. The execution starts at the address given on the command line or at the current value of the stored program counter if no address is specified. At the start of execution the register table is passed to the applications program and the break-points are inserted.

**10.8.10 HELP**

syntax: HELP

abbreviation: H

purpose: Display a list of commands

This command is used to display a list of executable commands in the debug monitor. It gives the command name and any parameters required.

### **10.8.11 KILL**

syntax:           KILL <address>

abbreviation:    K

purpose:          Remove a break-point

This command is used to remove a break-point from the break-point list. A specific break-point can be removed by specifying its address after the kill command. If the address specified is zero then all the break-points are removed.

### **10.8.12 LONG**

syntax:           LONG

abbreviation:    LON

purpose:          Set the operating data size to long words.

This command is used to change the data size used by some of the memory commands like DUMP and CHANGE to 32 bits. Long can be used as a command on its own in which case it affects all subsequent memory commands or it can be used as an option in which case it only affects the current command. If an odd address is specified then the address is rounded down to the next even address.

**10.8.13 MOVE**

syntax: MOVE <address 1> <address 2> <length>

abbreviation: M

purpose: Move a block of memory to a new address.

This command is used to move a block of memory from one address to another. The byte at address 1 is copied to address 2 until the number of bytes specified by length have been copied. If address 2 is inside the source memory area the copy is done in reverse so as not to overwrite the source block. NB overwriting areas of memory can be dangerous make sure that nothing else is using the memory before you copy into it.

**10.8.14 NOREAD**

syntax: NOREAD

abbreviation: N

purpose: To prevent the CHANGE instruction from reading data.

This command prevents the CHANGE command from reading the data at an address. This is particularly useful when using peripheral devices whose registers have different functions depending on whether they are read or written. Noread can be used as a command on its own or as an option with the change command.

### **10.8.15 ODD**

syntax: ODD

abbreviation: O

purpose: Set the data size to bytes on odd addresses.

This command allows the CHANGE command to look at peripheral devices which only appear at odd addresses. Many 68000 peripherals only have an 8-bit data bus which is connected directly to the first 8 data bits of the CPU (d0-d7). This has the effect of making the peripherals registers appear only on odd addresses. In order to be able to access these devices the ODD command forces the size to bytes, sets the address given in the CHANGE instruction to the next odd address and forces the address to increment by two for each byte. This command can be used on its own in which case it affects all subsequent CHANGE commands or it can be used as an option.

### **10.8.16 READ**

syntax: READ

abbreviation: REA

purpose: Allow the change instruction to read data.

The READ command allows the CHANGE command to read and display the contents of a location before changing it, it is used in conjunction with NOREAD. Read can be used as a command on its own or as an option with the change command.

**10.8.17 REGISTER**

syntax: REGISTER [<name> <value>]

abbreviation: R

purpose: Display or modify the stored registers.

This command is used to display and change the CPU's registers. These registers are passed to and recovered from the application's program under test. If the command is used without a register name the complete register set is displayed as 3 lines. One line for the 8 data registers, one for the 8 address registers, and one for the program counter, status register and the two stack pointers.

Note:

The value of a7 reflects either the Usp (User stack pointer) or Ssp (System stack pointer) depending on the state of the system state bit in the status register. The pc (Program counter) contains the address of the next instruction to be executed.

Note:

The Ssp value passed to the application's program is not the same as the one used by the Debugger. If the command is followed by a register name and a value, this value is loaded into the register and will be passed to the application's program next time it is executed (GOTO). The values can be any 32-bit numbers but there are some things to watch out for with the address and status registers. The two stack pointers and the program counter must always contain even numbers and if possible all the other address registers should also be even. This will prevent address traps from occurring. The status register is only 16-bits long and care should be taken when setting the system state bit or lowering the interrupt mask.

### **10.8.18 TEST**

syntax: TEST <address> <address> [<pattern>]

abbreviation: TE

purpose: Test a block of RAM.

This command can be used to check that an area of RAM is working correctly. The test starts by filling the area to be tested with a pattern which contains the given pattern byte in the first two of every three bytes and the ones compliment of the pattern in the third. If this pattern is stored correctly another pattern is written into the memory. This one is made up of the pattern byte followed by the ones compliment followed by the pattern byte. If this pattern is stored correctly a final pass is made using a pattern made up of the ones compliment of the pattern followed by two copies of the pattern byte. Any errors which occur during any of the passes will be reported. If the pattern byte is omitted from the command line a pattern byte of FF is used.

**Note:**

The test is destructive. All the data in the RAM area being tested will be lost.

**10.8.19 TRACE**

syntax: TRACE [<address>]

abbreviation: T

purpose: To trace a program one instruction at a time.

This command is used to trace through the applications program instruction by instruction, updating the register table and disassembling the program as it goes. If an address is specified the tracing starts at that address otherwise it starts at the current value of the stored program counter in the register table. When the tracing starts, the register table is passed to the applications program. After each instruction the address and the mnemonic of the next instruction is printed. Pressing the space key will trace the next instruction while an ESC will save the applications programs registers in the register table and return to the Debugger command line. At this point the register table is ready to execute the next instruction so tracing can be restarted with just a T.

**Note:**

The break-points will not be inserted during a trace. If the applications program uses the 68000 trace vector this command will not work!

### **10.8.20 WORD**

syntax: WORD

abbreviation: W

purpose: Set the operating data size to words.

This command is used to change the data size used by some of the memory commands like DUMP and CHANGE to 16 bits. Word can be used as a command on its own in which case it affects all subsequent memory commands or it can be used as an option in which case it only affects the current command. If an odd address is specified then the address is rounded down to the next even address.

## **10.9 Option Summary**

The following are a list of commands that can be used as options to other commands. For a description of how to use them and their function please see the command summary section:

BYTE  
LONG  
NOREAD  
ODD  
READ  
WORD

## 11 Appendix

### 11.1 ASCII Character Codes

	MSB	0	1	2	3	4	5	6	7
LSB									
0		00	10	SP	0	@	P	'	p
1		01	11	!	1	A	Q	a	q
2		02	12	"	2	B	R	b	r
3		03	13	#	3	C	S	c	s
4		04	14	\$	4	D	T	d	t
5		05	15	%	5	E	U	e	u
6		06	16	&	6	F	V	f	v
7		07	17	'	7	G	W	g	w
8		08	18	(	8	H	X	h	x
9		09	19	)	9	I	Y	i	y
A		0A	1A	*	:	J	Z	j	z
B		0B	1B	+	;	K	[	k	{
C		0C	1C	,	<	L	\	l	
D		0D	1D	-	=	M	]	m	}
E		0E	1E	.	>	N	^	n	~
F		0F	1F	/	?	O	_	o	DEL

## 11.2 References

Motorola for MC68307

Programming in Modula-2  
Niklaus Wirth  
Springer-Verlag

ISBN 0-387-50150-9 4th Edition

Modula-2 Primer  
Stan Kelly-Bootle  
Howard W.Sams & Company

ISBN 0-672-22560-3

Portable Modula-2 Programming  
M Woodman, R Griffiths, J Souter and M Davies  
McGraw-Hill International Series in Software Engineering

ISBN 0-07-707201-4

Software Development with Modula-2  
David Budgen  
Addison Wesley

ISBN 0-201-18482-6

Modula-2 made easy  
Herb Schildt  
McGraw-Hill

Advanced Modula-2  
Herb Schildt  
McGraw-Hill

ISBN 0-07-881245-3

Language exception notes:

- 1 The Modula-2 reserved words in italics are not implemented.
- 2 Local MODULEs are not supported
- 3 PROCEDURE types are not implemented
- 4 SET type not implemented
- 5 RECORD type not implemented
- 6 Enumeration types are not implemented
- 7 BYTE and WORD parameters will not accept other types

This page is intentionally left blank