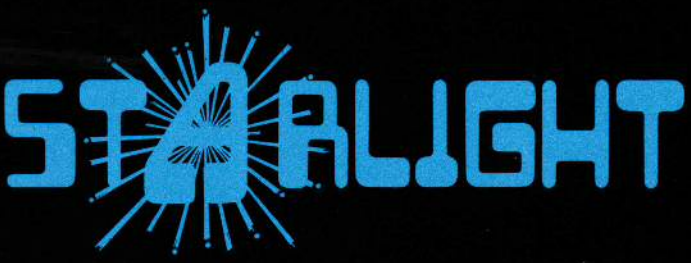


AN INTERGALACTIC
EDUCATIONAL
EXPERIENCE

GAMES CREATOR

FOR THE

ATARI®

TDI
Software, Inc.

M2S
Limited

NOTE: Any reference in this manual to TDI will apply to TDI Corporation, M2S, Ltd., and TDI Software, Inc.

While every precaution has been taken in the preparation of this manual, no responsibility is assumed for errors or omissions nor is any liability assumed for loss or damage resulting from the use of the information it contains.

Atari ST is a trademark of Atari Corp.

Published by:

TDI Software, Inc.
P.O. Box 550279
Dallas, Texas 75335-0279
U.S.A.

M2S, Ltd.
Box 393
Bristol BS99 7WU
U.K.

CONDITIONS OF USE

TDI programs contain material in which TDI retains proprietary rights. TDI wants these programs to be fully usable by you for the purpose for which they are supplied. No infringement of TDI's rights will occur provided that the following conditions are observed with respect to each program:

1. The programs are used only on a single machine at any one time.

2. The program is copied into machine-readable or printed form only for backup or modification purposes only in support of a single machine.

3. The copyright notice is reproduced and included in any copy or modifications made of the program and in any portion merged into other programs.

4. If this program package is transferred to another party, all copies and modifications made of the program must be transferred or destroyed. You do not retain any right with respect to the transferred package. The other party must agree to observe all the TDI conditions of use.

Any other act involving reproduction of or use of, or other dealing in the programs is prohibited.

No statements contained in this package shall affect the statutory rights of consumers.

TDI Software, Inc. Limited Warranty

TDI is committed to providing quality products and has established the following warranty for TDI products:

For a period of 90 days from the date of license of the software to the retail customer, TDI warrants to the customer that the materials of the disk on which the licensed program is recorded and the User Manual (the "Product") are not defective and that the licensed program (the "Program") is properly recorded on the disk. TDI also warrants for such 90-day period that the Program operates substantially as described in the User Manual and that the User Manual contains all the information which TDI and its software suppliers deem necessary for the use of the licensed program. THIS WARRANTY DOES NOT APPLY TO DEFECTS DUE, DIRECTLY OR INDIRECTLY, TO MISUSE, ABUSE, NEGLIGENCE, ACCIDENT, REPAIRS OR ALTERATIONS OUTSIDE OF TDI'S FACILITY.

TDI LIMITS ALL IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY, PERFORMANCE, AND FITNESS FOR A PARTICULAR PURPOSE, TO A PERIOD OF 90 DAYS FROM THE DATE OF THE LICENSE OF THE SOFTWARE TO THE RETAIL CUSTOMER, AS ESTABLISHED BY THE CUSTOMER'S PAID INVOICE. SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

Limitations of Remedies:

TDI SHALL IN NO EVENT BE LIABLE FOR INCIDENTAL, CONTINGENT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING FROM USE OF THE PRODUCT OR PROGRAM, EVEN IF TDI OR AN AUTHORIZED TDI DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. TDI's entire liability and the customer's exclusive remedy shall be the replacement of any Product or Program not meeting TDI's "Limited Warranty" and which is returned to TDI or an authorized TDI dealer with a copy of the customer's paid invoice, within the 90-day Limited Warranty period. You agree that TDI's liability arising out of contract, negligence, strict liability in tort or warranty shall not exceed any amounts paid by you for the particular Product or Program licensed to you. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTY DAMAGES, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

The Limited Warranty is implemented solely through the TDI Product Replacement Program. This Limited Warranty gives you specific legal rights, and you may also have others which may vary from state to state.

CONTENTS

Modula-2/STarlight User Manual

Modula-2/STarlight User Manual

LIST OF FIGURES

1. Introduction and Startup Guide

This manual describes the implementation of the Modula-2 development system Modula-2/STarlight. It is neither a reference manual nor a course about programming in Modula-2.

Modula-2/STarlight has been developed jointly by Modula-2 Software Ltd. in the United Kingdom, and TDI Software, Inc. in the United States. The goal was to produce a development environment that was easy to work within, and a compiler that was efficient.

1.1 System Description

The system contains an editor and a compiler. These are loaded once from disk and remain in memory throughout a development session (which gives fast program switching.)

1.1.1 The Compiler

- Single-pass Modula-2 compiler
- Full Modula-2 language supported, as described in Revision 3 of Wirth's book
- Fast: typically compiles at a speed of 7000 lines per minute
- Generates native MC68000 code
- Object code is relocatable; no linking is necessary

1.1.2 The Editor

- Displays compiler detected errors
- Compile directly from edit buffer
- Familiar WordStar(TM) commands

1.2 Hardware Requirements

The Modula-2/STarlight system requires at least a 520ST with TOS in ROM and a 350K single sided disk drive. For serious program development, a 1040ST and one or two double-sided disk drives is recommended; a 1040ST and 20MB hard disk will provide the most powerful and responsive system.

1.3 Backing up Modula-2/STarlight

It is vital that you back-up the Modula-2/STarlight system. If you have not read your "Atari 520ST Owner's Manual" do so now. Pay particular attention to Chapter 4 page 41 about making backup disks and Chapter 5 page 44 about formatting disks. After you are totally familiar with making backup copies of disks, follow those steps to backup the Modula-2/STarlight system. Put the Modula-2/STarlight master disk in a safe place where it won't get damaged. The copy you have made of the disk will be your working copy.

1.4 Starting Modula-2/STarlight

Take the following steps to enter the Modula-2/STarlight system:

1. Turn on the power to the Atari ST, disk drive (if you have an external drive), and monitor.

2. Place your working copy of the Modula-2/STarlight disk into the disk drive.

3. Double click on the disk icon marked "A".

4. Locate the "STARLITE.PRG" icon in the window.

5. Double-click on the "STARLITE.PRG" icon.

The disk drive will whirr for a few seconds, and then you will be placed in the Modula-2/STarlight system.

1.5 Contents of the Distribution Diskette

The Modula-2/STarlight system is distributed on a single sided diskette with accompanying manual. The diskette contains:

STARLITE.PRG The Modula-2/STarlight system.

2. The Modula-2/STarlight Environment

The Modula-2/STarlight system consists of an editor and a compiler. These are totally integrated within one environment and are not separate programs.

The following chapters give detailed instruction on the various components of the system; here, we present the overall structure of the system.

When the system is started, you are presented with a blank desktop with the menu bar across the top of the screen. To start work upon a program, select the Open item from the File menu (for details, see the Editor chapter.) Once the module has been loaded, you can compile it by pressing the F1 key, or by selecting the Compile option from the Modula menu. After the compiler has finished, you will be put back into the editor. If there were compilation errors, the cursor will be at the first one showing you the error message above the program text. You can then correct the errors and recompile the module. When the module compiles without error, you are ready to test the program. Simply type F2 and the program will be loaded and run. The program is loaded alongside the Modula-2/STarlight system (which is always resident.) If the program terminates abnormally you will be presented with a dialog prompting you to continue, abort, or debug the program. If the program terminates normally (i.e., no execution errors) you will be placed back in the editor.

3. The Editor

The Modula-2/STarlight editor is a full screen text editor which provides a wide range of commands for entering and deleting text, moving text, finding and replacing particular text strings, and storing text in files. The STarlight Editor can be operated using the mouse and menu interface, or entirely from the keyboard. The keyboard commands are WordStar(TM) compatible; the editor also makes use of some of the special purpose keys on the ST keyboard.

3.1 Starting Off

The editor is the primary means of entering the Modula-2 program to be compiled. To start editing, select the "Open" item from the File menu. You will be presented with a file selector box.

3.2 Cursor Control

All cursor movement commands wrap onto the next or previous line. For example, typing cursor right when the cursor is already at the right of a line causes the cursor to be moved to the beginning of the next line. If your intention was actually to insert a space at the end of the line, you should use the space bar instead.

3.2.1 Cursor Left Move the cursor one character to the left. If the cursor is already at the beginning of a line, typing cursor left will cause the cursor to wrap to the end of the previous line.

3.2.2 Cursor Right Move the cursor one character to the right. If the cursor is already at the end of a line, typing cursor right will cause the cursor to wrap to the beginning of the next line.

3.2.3 Cursor Up Move the cursor up one line. If the cursor is at the top of the screen, the screen scrolls down one line.

3.2.4 Cursor Down Move the cursor down one line. If the cursor is at the bottom of the screen, the screen scrolls up one line.

3.2.5 Cursor Home Move the cursor to the top-left corner of the screen.

3.2.6 Line Left Move the cursor to the left of the line.

3.2.7 Word Left Move the cursor left one word. The cursor will cross the end of the line if it needs to. A word is defined as any sequence of characters up to the first space.

3.2.8 Word Right Move the cursor right one word. The cursor will cross the end of the line if it needs to.

3.2.9 Line Right Move the cursor to the right of the line.

3.2.10 Scroll Up Scroll the screen up. The cursor remains on its line until the cursor reaches the bottom of the screen.

3.2.11 Scroll Down Scroll the screen down. The cursor remains on its line until the cursor reaches the top of the screen.

3.2.12 Page Up The cursor is moved back one page; it has no effect at the top of the file.

3.2.13 Page Down The cursor is moved forward one page; it has no effect at the bottom of the file.

3.2.14 Top of File The cursor is moved to the start of the file, i.e., the first character of text, and that page is displayed.

3.2.15 End of File The cursor is moved to the end of the file, i.e., the last character of text, and that page is displayed.

3.3 Deleting and Undeleting

Text can be deleted by character, word, line, or block. All text deleted by successive deletions in the same direction (i.e., left, right, or down) is saved internally and can be restored by the Undo key. This means that accidental deletions can be restored immediately. The deleted text remains saved (even after being undeleted) until a new deletion is made.

3.3.1 Delete Line Delete the line the cursor is on. All lines below the cursor are moved up one line, and the cursor is moved to the left of the screen.

3.3.2 Delete Line Left Delete all characters from the cursor position to the left of the screen. The cursor is moved to the left of the screen.

3.3.3 Delete Word Left Delete the word to the left of the cursor. The cursor will delete across the end of the line if it needs to.

3.3.4 Delete Word Right Delete the word to the right of the cursor. The cursor will delete across the end of the line if it needs to.

3.3.5 Delete Line Right Delete all characters from the cursor position to the right of the screen. The cursor is not moved.

3.3.6 Delete Character Right The character under the cursor is deleted; the text to the right of the cursor is moved back one space. Deleting at the end of a line will cause the line below the cursor to be appended to the line the cursor is on.

3.3.7 Delete Character Left The character to the left of the cursor is deleted; the text to the right of the cursor, and the cursor itself is moved back one space. Deleting at the start of a line will cause the line the cursor is on to be appended to line above the cursor.

3.3.8 Delete Word The word the cursor is sitting on is deleted. The cursor does not have to be at the start or end of a word for the word to be deleted.

3.3.9 Insert Line A blank line is inserted at the cursor line; the line the cursor is on is moved down one line, and the cursor is moved to the left of the blank line.

3.3.10 Undelete The Undo button places back into the text (at the cursor position) the latest deletions put into the internal delete buffer. The deleted text remains saved even after being undeleted until a new deletion is made.

3.3.11 Tab Pressing the Tab key inserts a number of spaces starting at the current cursor position. Spaces are inserted up to the beginning of the next word on the previous line. If the preceding line is not full screen width, tab positions are defined by default at every eighth character position in the blank part of the line. This method of tabbing provides a simple means of entering aligned columns of information on successive lines by tabbing at the start of each new line. It is also a convenient way to enter indented Modula-2 programs.

3.4 Block Commands

Block commands allow you to quickly and easily copy and move text around the edit buffer. The block commands work between two positions: the mark position and the cursor position. For all block commands other than Mark Block Begin, the mark must be set either before or after the cursor for the command to work. If the mark is not set, the block commands will simply ignore the request after issuing a warning.

3.4.1 Mark Block Begin This marks the start of the block at the cursor position. If the mark was already set at some location, it is simply moved to the cursor position. The mark remains set until an insertion or deletion takes place, after which it becomes unset.

3.4.2 Delete Block This deletes the block between the mark and the cursor position. It places the deleted text into an internal buffer so that it can be recovered. This allows you to copy and move large chunks of text around the edit buffer without resorting to continual delete-line and undelete operations.

3.4.3 Read Block from Disk This command prompts for a filename with the standard file selector box. Once a file has been selected, clicking on the OK button will read that file into the edit buffer at the cursor position. Clicking on the CANCEL button will abort the command with no action.

3.4.4 Write Block to Disk This writes the block of text between the marked position and the cursor position onto a named file. A file selector box is presented as with the Read Block from Disk command; type the name of the file and click OK to write the block, CANCEL to abort the write. The

block of text is not deleted from the edit buffer.

3.4.5 Print Block The block of text between the marked position and the cursor position is printed on the printer. This command takes account of the current settings of the 'Install Printer' desk accessory, so the text may be diverted both to parallel and serial printers. A form-feed is sent at the end of the text.

3.4.6 Start of Block This moves the cursor to the marked position if it has been set. If it is unset, this command does nothing.

3.5 File Commands

3.5.1 List File to Printer As Print Block, but the whole of the edit buffer is printed on the printer, ignoring any marked block.

3.5.2 Save File and Continue The edit buffer is written to disk, and the editor is reentered for you to continue editing.

3.5.3 Save to Named File As Write Block, but the whole edit buffer is written to disk, ignoring any marked block.

3.5.4 Save and Edit New File Writes the edit buffer to disk, and then prompts for a new file to edit.

3.5.5 Save File and Exit Writes the edit buffer to disk, and then closes the edit window.

3.5.6 Abandon File The changes you have made to the file since it was last loaded or saved will be abandoned. Before abandoning the changes, a dialog is presented asking whether you really wish to abandon the modifications. Clicking OK will throw away the modifications you have made and close the edit window; clicking CANCEL will put you back into the editor.

3.6 Summary of Editor Commands

Cursor Up	^E	Cursor Up
Cursor Down	^X	Cursor Down
Cursor Left	^S	Cursor Left
Cursor Right	^D	Cursor Right
Cursor Home		Home
Line Left	^Q-D	F1
Word Left	^A	F2
Word Right	^F	F3
Line Right	^Q-S	F4
Scroll Up	^W	
Scroll Down	^Z	
Page Up	^R	- on keypad
Page Down	^C	+ on keypad
Top of File	^Q-R	* on keypad
End of File	^Q-C	Enter on keypad

3.6.1 Insert and Delete Commands

Insert Line	^N	
Delete Line	^Y	
Delete Line Left		Shift F1
Delete Word Left		Shift F2
Delete Word Right		Shift F3
Delete Line Right	^Q-Y	Shift F4
Delete Character Right	^G	Delete
Delete Character Left	^H	Back Space
Delete Word	^T-T	
Tab	^I	Tab
Undelete		Undo

3.6.2 Block Commands

Mark Block Begin	^K-B
Delete Block	^K-Y
Read Block from Disk	^K-R
Write Block to Disk	^K-W
Print Block	^K-P
Start of Block	^Q-B

3.6.3 File Commands

List File to Printer	^K-L
Save File and Continue	^K-S
Save to Named File	^K-N
Save and Edit New File	^K-D
Save File and Exit	^K-X
Abandon File	^K-Q

3.6.4 Other Commands

Repeat Last Find/Replace	^L	
Find	^Q-F	
Replace	^Q-A	
Find Next Error	^Q-X	
Find Previous Error	^Q-E	
Toggle Word Case	^T-U	
Toggle Word Capitalisation	^T-C	
Toggle Auto-Indent	^Q-I	
Compile		F1
Compile and Run		F2

Note that ^L repeats the last find, or the last replace, or the last find error. A find error is done automatically by the editor when the compiler detects an error, so ^L may be used right away in those cases (until a normal find or replace operation has been carried out).

4. The Compiler

This chapter describes the use of the Modula-2 compiler. The Modula-2 language was designed by Prof. Niklaus Wirth at the Swiss Federal Institute of Technology (ETH) in Zurich. This language compiler conforms to the specification laid down in the document "Report on the Programming Language Modula-2" in the book "Programming in Modula-2", 3rd Revised Edition. The syntax of the language can be found in Section 10.

4.1 Glossary

Compilation Unit	Unit accepted by compiler for compilation, i.e., definition module or program module.
Definition Module	Part of a separate module specifying the exported objects.
Program Module	Implementation part of a separate module (called an implementation module) or main module.
Source	Input to the compiler, i.e., a compilation unit in the edit buffer.
Symbol File	Compiler output file with symbol table information. The information is generated during compilation of a definition module; it is read when the corresponding implementation module is compiled, or when it is imported into another compilation unit.
Reference File	Compiler output file with symbol table information. The information is generated during compilation of a program module.
Object File	Compiler output file with generated native MC68000 code in Modula-2/STarlight loader format; it is read when it is loaded for execution, or when a program that imports it is loaded.

5. The Implemented Language

Modula-2 is an evolving language and has gone through some agreed revisions. A brief description of the implemented language follows.

5.1 Standard Types

For a complete description of the storage layout and data formats, please see Section 9. The standard types are summarised in Figure 1.

INTEGER	The value range of the type INTEGER is -32768 to 32767 represented in 16-bit two's complement format. The compiler does not allow the direct definition of -32768, so it must be computed indirectly by -32767-1.
CARDINAL	The value range of the type CARDINAL is 0 to 65535 represented in 16-bit unsigned format.
REAL	Values of type real are represented in four bytes in IEEE format. The value range is -1.7014E38 to +1.7014E38.
CHAR	Values of type char are represented in one byte in ASCII format.
BITSET	The type BITSET is defined as SET OF [0..15] (2 bytes). Consider that sets are represented from the low order bits to the high order bits, i.e., (0) corresponds to the ordinal value 1.
LONGINT	The value range of the type LONGINT is -2147483648 to 2147483647 represented in 32-bit two's complement format. The compiler does not allow the direct definition of -2147483648 so it must be computed indirectly by -2147483647-1.
LONGCARD	The value range of the type CARDINAL is 0 to 4294967295 represented in 32-bit unsigned format.
LONGREAL	At present this type is not supported.

Figure 1. Modula-2 Standard Types

5.2 Standard Functions

Standard functions are predefined (i.e., need not be imported). Some are generic procedures that cannot be explicitly declared, i.e., they apply to several classes of operand type or have several possible parameter list forms. Standard functions are summarised in Figure 2:

ABS(x)	Return the absolute value of x, i.e., if x < 0.0 returns -x, otherwise returns x. x is of type INTEGER or LONGINT; result type = argument type.
CAP(ch)	If ch is a lower case letter, the corresponding uppercase letter; otherwise the same letter. ch is of type CHAR.
CHR(x)	The character with ASCII code x. x is of type INTEGER or CARDINAL.
FLOAT(x)	x of type INTEGER or LONGINT represented as a value of type REAL. x must be >= 0.
FLOATD(x)	x of type INTEGER or LONGINT represented as a value of type LONGREAL. x must be >= 0.
MAX(T)	T is any scalar type (including real). Result is the type's maximum value.
MIN(T)	T is any scalar type (including real). Result is the type's minimum value.
ODD(x)	TRUE if x is odd, otherwise FALSE. x of type INTEGER, LONGINT, CARDINAL, or LONGCARD.
ORD(x)	Ordinal number (of type INTEGER) of x in the set of values defined by type T of x. T is any enumeration type, CHAR, INTEGER, or CARDINAL.
SIZE(x)	Number of bytes (INTEGER) the variable occupies in memory.
SIZE(T)	Number of bytes (INTEGER) the type occupies in memory.

TRUNC(x)	Real number x (of type REAL or LONGREAL) truncated to its integer part (of type INTEGER). x must be >= 0.0.
TRUNCD(x)	Real number x (of type REAL or LONGREAL) truncated to its integer part (of type LONGINT). x must be >= 0.0.

Figure 2. Modula-2 Standard Functions

5.3 Standard Procedures

Standard procedures are predefined (i.e., need not be imported). Some are generic procedures that cannot be explicitly declared, i.e., they apply to several classes of operand type or have several possible parameter list forms. Standard procedures are summarised in Figure 3:

DEC(x)	x := x - 1. x must be a scalar.
DEC(x, n)	x := x - n. x must be a scalar, n must be of type INTEGER, LONGINT, CARDINAL, or LONGCARD.
EXCL(s, i)	s := s - {i}.
HALT	Terminates program execution, which can be used as a programmed breakpoint in Modula-2/STarlight.
INC(x)	x := x + 1. x must be a scalar.
INC(x, n)	x := x + n. x must be a scalar, n must be of type INTEGER, LONGINT, CARDINAL, or LONGCARD.
INCL(s, i)	s := s + {i}

Figure 3. Modula-2 Standard Procedures

5.4 The SYSTEM Module

Explicitly system-dependent features are imported from the module SYSTEM. Although this module cannot be standard for all implementations of Modula-2, it typically exports the type ADDRESS which is compatible with all pointer types. The generic function VAL(T, x) is effectively a replacement


```
DEFINITION MODULE SYSTEM;

  TYPE ADDRESS = POINTER TO BYTE;
               (* compatible with LONGCARD and
                  all pointer types *)
       BYTE;   (* smallest addressable unit,
                  uninterpreted; TSIZE(BYTE) = 1.
                  ARRAY OF BYTE is compatible
                  with everything. *)
       WORD;   (* two uninterpreted consecutive
                  bytes beginning at an even address,
                  assignment compatible with
                  all types of size 2. *)

  (* subsequently T denotes any
     simple type <= 4 bytes *)

  PROCEDURE ADR(VAR x: AnyType): ADDRESS;
    (* return storage address of variable x *)

  PROCEDURE CODE(x: WORD);
    (* place x directly into instruction stream *)

  PROCEDURE REGISTER(reg: INTEGER): LONGINT;
    (* return MC68000 register contents;
       reg must be a constant.
       reg: 0..7 = D0-D7; 8..15 = A0-A7 *)

  PROCEDURE SETREG(reg: INTEGER; val: T);
    (* set register to value; reg as REGISTER *)

  PROCEDURE TSIZE(AnyType): INTEGER;
    (* return number of bytes required
       for storage of AnyType *)

  PROCEDURE LONG(hi, lo: INTEGER): LONGINT;
    (* construct LONGINT from two INTEGERs;
       hi is the high 16 bits,
       lo the low 16 bits *)

  PROCEDURE LONG(x: T): LONGINT;
    (* coerce x to a LONGINT *)

  PROCEDURE SHORT(x: LONGINT): INTEGER;
    (* truncate x to 16 bits, no checks *)
```

```
    PROCEDURE SHIFT(x: T; n: INTEGER): T;
      (* x shifted by n bits;
         n > 0, x shifted left,
         n < 0, x shifted right *)

    PROCEDURE VAL(AnyType0; x: AnyType1): AnyType0;
      (* x is converted to have type AnyType0 *)

END SYSTEM.
```

Figure 4. Definition of the SYSTEM Module

for type-transfer functions T(x). Its value is x, interpreted as type T. No code is generated for this pseudo-procedure. Its explicit import is to make the use of machine-dependent type transfers explicit and more easily locatable.

5.5 Differences and Restrictions

The implementation of Modula-2 on the Atari ST has some differences and restrictions from the original multi-pass compilers:

5.5.1 Assignment Compatibility The following types are assignment compatible with each other (overflow is checked):

INTEGER, CARDINAL, LONGINT, and LONGCARD
REAL and LONGREAL

5.5.2 Procedures No forward references are permitted, except in definitions of pointer types and in forward procedure declarations. Procedures referenced before declaration must be declared before by a forward declaration. The format is:

```
PROCEDURE P(parameter list); FORWARD;
```

or

```
PROCEDURE P(parameter list): result type; FORWARD;
```

The corresponding procedure must have the full header repeated and must lie at the same nesting level as the forward declaration.

5.5.3 Function Procedures The result type of a function procedure must be 1, 2, 4, or 8 bytes. This includes all simple types.

5.5.4 Data Size The maximum total global data size must be less than 32KBytes per compilation unit. There is no limit on total data size over all modules.

5.5.5 Code Size The maximum code size must be less than 24000 bytes per compilation unit. The total code size over all modules has no limit.

5.5.6 Index Types in Array Declarations The index type must be a subrange type.

5.5.7 Standard Functions, Procedures, and Types The procedures NEW, DISPOSE, TRANSFER, IOTRANSFER, NEWPROCESS, and the type PROCESS are not implemented (although defined as standard objects in earlier reports on the language.) The procedure SIZE is a standard procedure and is identical to the function TSIZE in the module SYSTEM. To implement NEW, simply replace NEW(p) by Storage.Allocate(p, SIZE(T)), where p is declared as POINTER TO T. Similarly, DISPOSE(p) should be replaced by Storage.Deallocate(p, SIZE(T)).

5.5.8 Subranges The bounds of a subrange must be less than 2**15 in absolute value, and the difference MAX(Subrange) - MIN(Subrange) must be less than 2**15.

5.5.9 Opaque Types If a type T is declared in a definition module to be opaque, it cannot (in the corresponding implementation module) be declared as equal to another, named type.

5.5.10 Enumeration Types An enumeration may have at most 256 elements.

5.5.11 Sets A set may have at most 16 elements.

5.5.12 Procedures Declared in Definition Modules If a procedure (heading) is declared in a definition module, its body must be declared in the corresponding implementation module proper; it cannot be declared in an inner, local module.

6. Modula-2 Tutorial

This chapter describes each major feature of the Modula-2 language with simple examples.

6.1 The Elements of Modula-2 Programs

Modula-2 programs consist of a sequence of symbols, each of which have a specific meaning.

6.1.1 Identifiers Identifiers are programmer defined names that may be associated with constants, data types, procedures, modules, and variables. An identifier starts with a letter followed by any number of letters or digits. Unlike other languages all characters in an identifier are significant; upper and lower case are considered to be distinct. Modula also provides a range of predeclared identifiers for standard procedures and functions.

Some examples of valid identifiers are:

```
Modula2
aVeryLongIdentifier
notfound
```

Some examples of invalid identifiers are:

```
2BorNot2B       - cannot start with a number
The_time        - underlines not permitted
todays date     - spaces not allowed
```

6.1.2 Numeric Constants Numeric constants can be divided into two distinct groups, those that represent whole numbers and those that may have a fractional part. In Modula these are classified as integers and reals. These groups may be further subdivided to produce different ranges of numbers, although not all compilers support them.

Numeric constants are represented by a sequence of digits with no intervening spaces. Some examples of constants are:

```
1986    102     453     74
```

By using a suffix you can express constants in bases other than decimal. The suffix "B" denotes an octal constant, and the suffix "H" denotes a hexadecimal constant. For example,


```
22B07b  6F2EH   0FFD2H  3FH
```

Letters and the suffix must always be entered in uppercase; lowercase will give compilation errors.

To prevent ambiguity, however, hexadecimal numbers that start with a letter, for example FFD2H, must be preceded by a leading zero. This ensures that the constant is taken as a number and not an identifier. So, the hexadecimal constant FFD2 would be written as 0FFD2H.

The suffix "D" denotes a decimal long constant , i.e., a constant that is compatible with LONGINTs, LONGCARDs, and ADDRESSs. A hexadecimal constant greater than the 0FFFFH will also be interpreted as a long constant.

Real constants contain a decimal point (period), an optional fractional part, and an optional scale factor. The scale factor is specified by the letter E (in uppercase) followed by an integer which may be preceded by a sign. Examples of real constants are:

```
0.0003   0.4   7.0E+3   3.14E-4   0.003E-6
```

The E is short for "times ten to the power of". So, for example, the following reals all represent the number 1024.0:

```
0.1024E4    1024.0    102400.E-2    10.24E+2
```

Note that commas, apostrophes or spaces cannot appear in a numeric constant; also the scale factor for a real constant must be constant, so 1.4En is not allowed.

6.1.3 String Constants String constants are sequences of characters surrounded by quotes or apostrophes. This allows quotes or apostrophes (but not both) to be contained within the string. Examples of string constants are:

```
"Modula-2"
'Pascal-type string'
"What's up?"
''
```

Note that the last example contains no characters and is given a special name, the null string.

String constants that contain a single character have a special property: they may be considered to be a character constant as well as a string constant.

6.1.4 Operators and Delimiters These are either special characters or reserved words. Reserved words must not (and cannot) be used as identifiers; they are written in upper case.

6.1.5 Reserved Words The reserved words for Modula are:

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

Figure 5. Modula-2 Reserved Words

In addition the reserved word FORWARD is added to the above list for the single-pass compiler; it is not needed (and is not available) in multi-pass compilers.

6.2 Declarations

All identifiers in a Modula program must be declared. Data types, variables, and symbolic constant declarations are introduced by the reserved words TYPE, VAR, and CONST, respectively.

6.2.1 Constant Declarations Constant declarations, as their name implies, associate a constant with an identifier. An identifier declared this way is named a symbolic constant. This symbolic constant may then be used in the program in place of the constant itself. So, wherever a constant is legal, so is a symbolic constant.

```
CONST
  RowsOnVDU = 24;
  ColsOnVDU = 80;
```


This defines the symbolic constants RowsOnVDU and ColsOnVDU to be 24 and 80. Symbolic constants are not limited to integers; they can be integers, strings, sets, or characters:

```
CONST
  FourStars = "****";
  Pi = 3.1415926;
  Primes = {1,2,3,5,7,11,13};
  CR = 15C;
```

In fact, any constant expression will do the following:

```
CONST
  OneHundred = 50 + 25 + 25;
  CharsDisplayableOnVDU = RowsOnVDU * ColsOnVDU;
```

The last declaration shows how previously defined constants can be used to define more constants.

Functions returning constant values known at compile-time can be used in constant declarations. So, for example, the following is legal:

```
CONST
  MAXINT = MAX(INTEGER)
```

6.2.2 Type Declarations Type declarations are introduced by the reserved word TYPE. This facility allows you to define your own data types to augment the standard types supplied by Modula.

6.2.3 Variable Declarations Variable declarations are introduced by the reserved word VAR. All variables must be declared. Declaring a variable associates it with a data type and storage address. For example,

```
VAR
  CursorX: INTEGER;
  ChebyshevCoeff: REAL;
```

The variable CursorX is only able to hold an integer; ChebyshevCoeff is only able to hold a real. If you try to assign a real number, say, to CursorX, the compiler will detect it and report an error.

You can declare several variables of the same type at the same time as follows:

```
VAR
  CursorX, CursorY: INTEGER;
```

6.3 Data Types

Modula supplies a wide range of data types that allow you to structure your data in the most appropriate way. Data types may be divided into two groups, namely simple and structured. Simple data types represent a single value, whilst structured data types represent a collection of values.

The simple data types are sometimes referred to as scalar types. Scalar types are:

```
INTEGER, LONGINT
CARDINAL, LONGCARD
CHAR
BOOLEAN
User-defined scalar types
Subrange types.
```

The REAL and LONGREAL types are omitted from the above list, although they are simple types. Real values are not handled in the same way as scalars, so, for example, you cannot define a subrange of real values.

6.3.1 Simple Data Types Simple data types represent a single value. Modula provides several numeric types: integers and cardinals represent whole numbers, and reals represent numbers with fractional parts. In addition to these there are types to store characters, results of comparisons, and user defined types.

This section will describe some of the available data types; the more specialised types will be left for later.

6.3.1.1 The INTEGER Data Type Integer values are signed, whole numbers. The exact range of values offered depends on the implementation and the underlying hardware.

6.3.1.2 The REAL Data Type Real numbers are signed quantities that represent a number that may contain a fraction. Again, the range and accuracy offered depends on the implementation.

6.3.1.3 The CARDINAL Data Type A cardinal is an unsigned integer. Therefore, the minimum value a cardinal can hold is zero; the largest is defined by the implementation, but it is never less than the maximum integer value.

6.3.1.4 The CHAR Data Type This type represents a character of the hosts character set. Most computer systems use the ASCII (American Standard Code for Information Interchange) or ISO (International Standards Organisation) code; others, notably IBM, use EBCDIC (Extended Binary Coded Decimal Interchange Code).

6.3.1.5 The BOOLEAN Data Type Boolean values are either TRUE or FALSE, denoting logical truth or logical falsehood, respectively. Such values are the result of comparisons and are mainly used to alter the flow of control in a program.

6.3.1.6 User Defined Scalar Types Modula allows you to define a new unstructured data type that consists of an ordered list of values. These types are scalars, but may also be called enumerated types. For example,

```
TYPE
  ComputerType = (Cray1, CrayXMP, Cyber205, Lilith,
                  PDP11);

VAR
  Computer: ComputerType;
```

The variable Computer may only take one of the possible values enumerated in ComputerType. Thus, to set the variable Computer to the value Lilith we write:

```
Computer := Lilith;
```

Trying to set it to anything that isn't listed in ComputerType raises an error.

6.3.1.7 Subrange Types Subrange types are allowed to take values from a selected range of another type. For example, we could classify supercomputers as follows:

```
TYPE
  SuperComputerType = [Cray1..Cyber205];

VAR
  SuperComputer: SuperComputerType;
```

This defines SuperComputerType to be a subrange of ComputerType. The values that SuperComputer may take are Cray1, CrayXMP, and Cyber205. If you try to assign Lilith, say, to SuperComputer the compiler will report an error.

Note that you cannot define a subrange of the REAL or LONGREAL type, so the following is illegal:

```
TYPE
  WaterTemperature = [0.0..100.0];
```

6.3.1.8 The POINTER Data Type The pointer type is used for accessing variables that are created at runtime. The variable that a pointer points to (or references) is called a dynamic variable. The dynamic variable is not declared as with all other variables; instead a pointer to the variable is declared. During program execution the pointer is used to create the dynamic variable. For example,

```
VAR
  p: POINTER TO INTEGER;
```

Here, p is declared to be a pointer to an integer. Therefore, the dynamic variable is of type integer and p points to it. As yet the dynamic variable has not been created; this is done by

```
Storage.Allocate(p, SIZE(p^))
```

This creates a new dynamic variable in a piece of unused memory. This variable can then be accessed by using the dereferencing operator ^.

```
p^ := 3;  (* assign 3 to the dynamic integer *)
```

Pointers and dynamic variables are used mainly when the size of a data structure cannot be determined at compile time.

6.3.2 Structured Data Types Structured data types represent a collection of values. There are no predefined structured data types; you must define them yourself using the constructs described below.

6.3.2.1 The ARRAY Data Type An array is a collection of values, all of the same type. Each value in the array is called an array element. The number of elements in an array is fixed, i.e., it cannot change at run time.

There are two parts to an array declaration, the index type and the element type. The index type defines the number of elements in the array and how they will be accessed. The element type defines the type of all the elements of the array. For example, consider:

```
VAR AnArray: ARRAY [1..7] OF INTEGER;
```

The ARRAY keyword introduces the array declaration. After this comes the index type, in this case the subrange 1 to 7. The reserved word OF separates the index type from the element type, the element type being INTEGER.

The elements in the above array are accessed as follows:

```
AnArray[1] AnArray[2] ... AnArray[7]
```

The index type is not just restricted to subrange types, it can be any scalar type. So, for example, we could say:

```
VAR
  ComputerPrices: ARRAY ComputerType OF REAL

BEGIN
  ComputerPrices[CrayXMP] := 4.E6;
  ...
```

6.3.2.2 The RECORD Data Type The record type usually groups together related pieces of information about an object. Unlike the array type, elements of a record variable do not have to be of the same type. Instead of using an index to identify the piece of data, record variables use identifiers. The declaration

```
VAR
  City: RECORD
          latitude, longitude: REAL;
          altitude: REAL;
          population: CARDINAL
        END
```

creates a record variable called City which contains four elements. The elements of the record are accessed by using "dot" notations:

```
City.latitude
City.longitude
City.altitude
City.population
```

As you can see, there are three real values in the record and one cardinal. The order of record elements does not matter, but it is good practice to place related items, (such as latitude and longitude) together.

6.3.2.3 The SET Data Type A set is a collection of values, all of which are called elements or members. A member is either present in a set, or it is not; no member may be present twice. A set declaration is introduced by the words SET OF, then the base type is given. The base type defines what members the set will have. For example,

```
TYPE
  LetterSet = SET OF ["A".."Z"];

CONST
  Vowels = LetterSet{'A','E','I','O','U'};
  Consonants = LetterSet{'A'..'Z'} - Vowels;
  ...
```

This defines LetterSet to have as its members the letters A to Z. The first statement after BEGIN defines the variable Vowels to be the set of letters that represent vowels. The second statement defines the set of consonants; this is defined to be the set of all letters (LetterSet{'A'..'Z'}) with the set of vowels removed (so leaving just consonants.) Set operators will be described later.

6.4 Statements

Modula-2 statements constitute the executable part of a program; they do all the work. Executable statements are introduced by the word BEGIN and are terminated by END, and a semicolon is used to separate them. Modula provides several types of statements, such as the following:

Assignment:	Procedures, functions and assignments allow you to assign values to variables.
Looping:	These provide a means to execute a series of statements more than once. There are the simple while and repeat loops, the general loop which allows multiple exit points, and the for loop which counts the number of times the loop is executed.
Conditionals:	These allow you to alter the flow of control in a program according to conditions (tests). There is the IF statement for simple conditions and the CASE statement which allows an efficient multi-way branch.

6.4.1 The Assignment Statement The assignment statement gives a value to a variable; the old content of the variable is lost. Consider

```
r := sqrt(x*x + y*y);
```

The value of r is overwritten with the square root of x squared plus y squared (this calculates the radius of a circle, centre the origin). x, y and r are real variables.

6.4.1.1 The IF Statement The IF statement specifies that a sequence of statements are executed only if a condition is true. This condition is given by a Boolean (logical) expression. For example,

```
IF b*b = 4.0*a*c THEN
  WriteString("perfect roots")
END
```

The WriteString is executed if and only if b squared is equal to 4*a*c. The word END terminates the IF statement.

A second sequence of statements may be executed if the condition fails; these alternative statements are separated from the true statements by the word ELSE. For example,

```
IF b*b = 4.0*a*c THEN
  WriteString("perfect roots")
ELSE
  WriteString("non-perfect roots")
END
```

Multiple conditions may also be tested for by the ELSIF clause:

```
IF b*b = 4.0*a*c THEN
  WriteString("perfect roots")
ELSIF b*b < 4.0*a*c THEN
  WriteString("complex roots")
ELSE
  WriteString("real roots")
END
```

6.4.2 The CASE Statement The case statement can be seen as a multi-way branch. Given one value, one of a selection of statements will be executed. For example, we could say:

```
VAR age: CARDINAL;
CASE age OF
   0..17: category := juvenile
| 18..25: category := youngman
| 30..64: category := man
ELSE       category := pensioner
END
```

Note that the case statement is started by the reserved word CASE; then the selector for the multi-way branch is given which is terminated by the , and, finally, the reserved word OF.

6.4.3 The REPEAT Statement One of the characteristics of a programmable computer is its ability to execute a series of commands over and over again. This repetition is called a loop. Modula provides several types of loops, the repeat statement being one of them. Consider the following example:


```
REPEAT
  x := x * 2
UNTIL x > y
```

The statements between the REPEAT and UNTIL are executed at least once. The condition, x > y, is then tested; if the test fails, the loop is restarted from the REPEAT; if the test succeeds, the loop finishes and control passes to the statements following UNTIL.

6.4.4 The WHILE Statement The WHILE statement is similar to the repeat statement; the differences are that the condition is tested at the start of the loop (not at the end) and the loop is executed if the test succeeds. For example,

```
WHILE x < y DO
  x := x * 2
END
```

As the condition is tested at the start of the loop the statements within the loop may not be executed at all (unlike the REPEAT loop which is executed at least once).

6.4.5 The FOR Statement The FOR statement indicates that a statement sequence is to be repeatedly executed while assigning a progression of values to a variable. This is simpler than it sounds; for example, the following fragment will write the numbers one to ten on separate lines:

```
FOR i := 1 TO 10 DO
  WriteCard(i,0);
  WriteLn;
END
```

The numbers one and ten are the initial and final values, respectively. The FOR statement is more flexible than this; it can count up or down by any step. Note that the statements within the loop may not be executed at all if the initial value is greater than the final value.

For example, this won't write anything:

```
FOR i := 10 TO 1 DO
  WriteCard(i,0);
  WriteLn;
END
```

To count down from 10 to 1, you should use the BY keyword:

```
FOR i := 10 TO 1 BY -1 DO
  WriteCard(i,0);
  WriteLn;
END
```

The number after the BY keyword has to be a constant. You cannot use a variable. You can use BY to count up (or down) in any step. For example,

```
FOR i := 10 TO 1 BY -2 DO
  WriteCard(i,0);
  WriteLn;
END
```

The FOR loop is not limited to counting up or down by INTEGERs and CARDINALs; it can count through characters and user-defined enumerations. For example,

```
VAR ch: CHAR;
    usr: (u0, u1, u2, u3, u4, u5);
FOR ch := 'A' TO 'Z' DO ...
FOR usr := u0 TO u5 BY 2 DO ...
```

Note that the values taken by the 'usr' variable in the last example will be u0, u2, and u4, and that BY is followed by an INTEGER constant.

6.4.5.1 The WITH Statement The WITH statement provides a shorthand for referring to the fields of record variables. So,

```
town.latitude   := 29.3;
town.longitude  := 13.4;
town.altitude   := 401.7;
town.population := 1000;
```

could be rewritten using WITH as


```
WITH town DO
  latitude   := 29.3;
  longitude  := 13.4;
  altitude   := 401.7;
  population := 1000;
END
```

NOTE: Using the WITH statement will normally produce more efficient and compact code on this implementation.

6.5 Declarations

Every identifier that is used in a Modula-2 program must be declared. Declarations are used to inform the compiler of the variables you will be using, what values symbolic constants have, and to give names to data types.

Section 2 briefly described variable, constant, and type declarations. This section covers declarations in more detail.

6.5.1 Constant Declarations Modula provides a way to associate a constant value with an identifier. The constant value is determined only once during compilation. The word CONST introduces a list of constant declarations. Each declaration has the following form:

```
identifier = constant;
```

For example,

```
CONST
  Title   = "Modula-2/STarlight";
  Version = 220;
  e       = 2.7182818;
  Primes  = {1,2,3,5,7,11,13};
```

This example shows that string, integer, and real constants may be associated with an identifier. Declaring an identifier to stand for a constant has several advantages:

- It provides better program documentation.
- It avoids "magic numbers" in a program.
- If used correctly it helps program maintenance.
- It helps the compiler to reduce code size and execution time.

These points can be illustrated by the following program fragment:

```
FROM InOut IMPORT WriteString;
FROM RealInOut IMPORT WriteReal;

CONST
  radius1 = 10.0;
  radius2 = 50.0;
  area1 = 3.1415926 * radius1 * radius1;
  area2 = 3.1415926 * radius2 * radius2;

BEGIN
  WriteString("area 1 = "); WriteReal(area1,0);
  WriteString("area 2 = "); WriteReal(area2,0);
END
```

This could be rewritten with a little more thought. The number 3.1415926 is the constant pi, the ratio of the circumference of a circle to its diameter. We could define a symbolic constant, say pi, to be this number. This helps a person who has never met the constant pi before to look up the definition of "pi" in a textbook. The same person if presented with 3.1415926, would have some trouble to discover what this magic number stands for.

We can now write the declarations this way:

```
CONST
  radius1 = 10.0;
  radius2 = 50.0;
  pi = 3.1415926
  area1 = pi * radius1 * radius1;
  area2 = pi * radius2 * radius2;
```

Defining a symbolic constant separates the constants value

with its uses. For example, consider

```
CONST
  RowsOnVDU = 24;
  ColsOnVDU = 80;
  CharsOnVDU = RowsOnVDU * ColsOnVDU;
```

This clearly defines that CharsOnVDU is related to the two constants RowsOnVDU and ColsOnVDU. If CharsOnVDU were declared as

```
CharsOnVDU = 24 * 80;
```

then the relationship is not as clear.

Say we now wish to rewrite our program so it runs on a 132 column VDU. Using the first set of constants we simply change ColsOnVDU to be the constant 132; the compiler takes care of updating the CharsOnVDU constant. Using the second definition of CharsOnVDU we must change the number 80 in the declarations of ColsOnVDU and CharsOnVDU. If we forget to change one, then it's likely that the program won't work.

Constant declarations are subject to the following restrictions:

- They must contain constants only.
- Any symbolic constants used to define a new symbolic constant must be declared before the new declaration.

The following example illustrates these points:

```
VAR
  z: INTEGER;

CONST
  y = z+1;            - illegal, z is not a constant
  circ = 10.0 * pi;   - illegal, pi is not yet declared
  pi = 3.1415926;     - this is ok
```

6.5.2 Variable Declarations A variable is a place that can hold an item of data. All variables must be declared before they are used. Declaring a variable associates it with a specific data type and memory address. For example, the following declares three variables, two of type cardinal and

one of type integer:

```
VAR
  i, j: CARDINAL;
  x: INTEGER;
```

The order of declaration is not important; the above could equally well have been written as follows:

```
VAR
  x: INTEGER
  i, j: CARDINAL;
```

or even

```
VAR
  j: CARDINAL;
  x: INTEGER
  i: CARDINAL;
```

There are some things to note about variable declarations:

- Each declaration is terminated by a semicolon; this separates it from the other declarations that follow.
- Two or more variables of the same type can be declared by simply separating the variable names by commas.

6.6 Simple Data Types Revisited

Every item of data in a Modula-2 program has an attribute associated with it called its type. The type determines the operations that can be performed (and how to interpret) the data.

Modula has built-in types for representing integer, real, boolean and character values. In addition it provides the ability for you to define your own types, and to build more complex data types.

Section 2 outlined the simple data types. This section covers the simple data types in more detail.

6.6.1 Numeric Data Types Modula provides three basic numeric types: integer, cardinal, and real. For bigger numbers and greater precision, long versions of these types

are available.

6.6.1.1 The INTEGER Type Integer values are signed, whole numbers. The range of integer values that are representable is -32768 to 32767.

Operator	Description
+	addition
-	subtraction, unary minus
*	multiplication
DIV	division
MOD	modulo

Figure 6. Basic INTEGER Operators

These operators are described fully in the next section. Note that integer division is denoted by the DIV operator and not the usual mathematical / operator (which is reserved for real numbers).

Modula provides two built-in functions, ODD and ABS, which operate on integers. The ODD function tests to see if its argument is odd; the ABS function takes the absolute value of its argument.

ODD(3)	=	TRUE	ABS(3)	=	3
ODD(74)	=	FALSE	ABS(74)	=	74
ODD(0)	=	FALSE	ABS(0)	=	0
ODD(-149)	=	TRUE	ABS(-149)	=	149

Figure 7. ODD and ABS Operator Examples

6.6.1.2 The CARDINAL Type Like integers, cardinals represent whole numbers, but only positive values and zero. The range offered by cardinals is 0 to 65535.

+	addition
-	subtraction
*	multiplication
DIV	division
MOD	modulo

Figure 8. Basic CARDINAL Operators

The ODD function can be applied to cardinals and its results are the same as for integers.

Note that monadic minus is not available to negate a cardinal and the ABS function cannot be used with a cardinal argument; this is because cardinals can never be negative.

A positive integer number can be converted into a real number using the FLOAT function, defined as

```
PROCEDURE FLOAT(x: INTEGER): REAL
```

```
FLOAT(0)      =  0.0
FLOAT(43)     =  43.0
FLOAT(65535)  =  65535.0
```

Figure 9. FLOAT Operator Examples

There is no predefined function to convert a negative integer to a real. To perform the conversion a function procedure like this can be used:

```
PROCEDURE IFLOAT(x: INTEGER): REAL;
BEGIN
  IF x < 0 THEN RETURN -FLOAT(-x)
  ELSE RETURN FLOAT(x)
  END
END IFLOAT
```

Figure 10. The IFLOAT Procedure

6.6.1.3 The REAL Type Real numbers represent signed numbers that can have a fractional part. The operators that are applicable to reals are:

```
+      addition
-      subtraction, unary minus
*      multiplication
/      division
```

Figure 11. Real Operators

Note that / is used to denote real division; DIV denotes integer or cardinal division. The MOD operator is not defined for real numbers in Modula.

The function ABS will return the absolute value of its argument as for integers.

To convert a positive real number into a cardinal the TRUNC function is used. It is defined as:

```
PROCEDURE TRUNC(x: REAL): INTEGER
```

It takes a real number, truncates it, and returns the cardinal representation. It can only truncate positive numbers; giving it a negative argument will produce a run-time error.

```
TRUNC(0.0)     =  0
TRUNC(3.5)     =  3
TRUNC(3.0)     =  3
TRUNC(100.9)   =  100
TRUNC(-3.5)    =  error, cannot truncate negative arguments
TRUNC(1.E20)   =  error, 1E20 is too large for cardinal range
```

Figure 12. TRUNC Examples

There is no way to convert a negative real into an integer. To perform the conversion a function procedure like this can be used:

```
PROCEDURE ITRUNC(x: REAL): INTEGER;
  VAR i: INTEGER;
      neg: BOOLEAN;
BEGIN
  neg := x < 0.0;
  i := TRUNC(ABS(x));
  IF x < 0.0 THEN RETURN -i
  ELSE RETURN i
  END
END ITRUNC
```

Figure 13. The ITRUNC Procedure

NOTE: The module MathLib0 contains a function 'entier' which will convert a real into an integer.

Using ITRUNC, any real number x in the range -32768.0 <= x < +32768.0 will be successfully converted into an integer. Anything outside this range will produce a run-time error (if arithmetic checking is on), or an invalid result (if

checking is off).

Rounding of a real number can be accomplished as follows:

```
PROCEDURE ROUND(x: REAL): CARDINAL;
BEGIN
  RETURN TRUNC(x + 0.5)
END ROUND
```

Figure 14. The ROUND Procedure

```
ROUND(0.0)   = 0
ROUND(3.5)   = 4
ROUND(3.49)  = 3
```

Figure 15. ROUND Examples

Similarly, using ITRUNC, reals may be rounded and returned as integers.

6.6.2 Ordinal Data Types

6.6.2.1 The CHAR Type A character value is any character from the ISO 7-bit character set. Associated with each character is an internal code which is found by using the ORD function:

```
ORD("A")   =  65
ORD('"')   =  34
```

The CHR function is the inverse of the ORD function; it takes an internal code and returns a character:

```
CHR(65)   =  "A"
CHR(34)   =  '"'   - quote mark character
```

The CHR function can be used to construct character constants that cannot be represented in the source program. These characters are called control characters as they are used to control output devices such as terminals. For example, a carriage return character, with internal code 13, can be constructed as follows:

```
CHR(13)
```

The 7-bit code is normally extended to an 8-bit code on most

computer systems. This allows the so-called international character set which includes accented letters and mathematical symbols.

6.7 Operators and Expressions

In its simplest form an expression can be a constant, a variable, or a function call.

```
nrVertices                - a constant
ModuleName[thisModule]    - an array variable
GetWord()                 - a function call
list^.next                - a dynamic variable
```

Figure 16. Simple Expressions

These elements can be combined with operators to form more complex expressions.

An operator normally takes two operands to yield a result; this type is called a dyadic operator or a binary operator.

For example,

```
age > 65
Factorial(n) + Factorial(n+1)
x - 10.3E4
```

There are operators that take only one operand; this type is called monadic operators or unary operators. Modula defines three unary operators, unary +, -, and NOT.

```
NOT ok
-pressure
+7
```

The following sections describe all operators available in Modula.

6.7.1 Arithmetic Operators The arithmetic operators, used with integer, cardinal and real types (including their long counterparts) are:

```
+     addition, unary plus (identity)
-     subtraction, unary minus (negation)
*     multiplication
/     real division
DIV   integer division
MOD   remainder of integer division
```

Some things to note about operators are:

- Both operands of binary operators must have the same base type: mixed mode arithmetic is not allowed.
- The result of an operator has the same type as its operands.
- The result of arithmetic operators must lie in the range of values representable in the argument's base type. If it lies outside this range a range error is raised.

6.7.1.1 The + Operator The + operator is used for addition of integers, cardinals, and reals (including their long counterparts). The sum has the same type as its operands. For example,

```
RealX + 3.0
Length + 100 + y.length
```

The + operator can be used with a single argument as a sign indicator. The use of + produces a result that is identical to its operand. For example,

```
+7
+Velocity
```

6.7.1.2 The - Operator The - operator is used for subtraction of integers, cardinals, and reals (including their long counterparts). The difference has the same type as its operands. For example,

```
B - 4 - factor
Z - 1.003E-2
```

The - operator can be used with a single argument as a sign inversion operator (negation). For example, assume x = +3, then -x = -3 and -(-x) = +3. When used in this context the operand must be of type integer, real, longinteger, or longreal.

6.7.1.3 The * Operator The * operator is used for multiplication of integers, cardinals, and reals (including their long counterparts). For example,

```
  3  *   4   =   12
(-3) *   4   =  -12
  3  * (-4)  =  -12
(-3) * (-4)  =   12
```

6.7.1.4 The / Operator The / operator is used for division of reals and longreals; for integer division the DIV operator is used. The quotient has the same type as its operands. For example,

```
  3.0  /   4.0   =   0.75
(-3.0) /   4.0   =  -0.75
  3.0  / (-4.0)  =  -0.75
(-3.0) / (-4.0)  =   0.75
```

Division by zero is an error: it will be trapped at compile and run time.

6.7.1.5 The DIV Operator The DIV operator is used for division of integers and cardinals (including their long counterparts). The integer quotient has the same type as its operands. The quotient is truncated towards zero. For example,

```
  7  DIV   3   =   2
(-7) DIV   3   =  -2
  7  DIV (-3)  =  -2
(-7) DIV (-3)  =   2
```

Division by zero is an error: it will be trapped at compile and run time.

6.7.1.6 The MOD Operator The MOD operator is used for finding the remainder after division of integers and cardinals (including their long counterparts). The result has the same type as its operands. For example,

```
  7  MOD   3   =   1
(-7) MOD   3   =   2
  7  MOD (-3)  =  -2
(-7) MOD (-3)  =  -2
```

A second argument of zero is an error (as we cannot divide by zero): it will be trapped at compile and run time.

NOTE: The modulo operator is defined by Wirth for positive arguments only. You may find that another implementation gives different values for negative arguments. To be safe, always give MOD positive arguments.

6.7.2 Relational Operators Relational operators are used to compare two operands for a certain condition. The two operands can be of scalar or real type; relational operators with set operands are described in Section 7.7.8.

The relational operators are:

>	greater than
>=	greater than or equal to
=	equal to
<	less than
<=	less than or equal to
<>, #	less than or greater than, i.e., unequal

NOTE: The operands of relational operators must have the same base type.

The result of a relational operator is a boolean value, true or false. A true value means that the condition is met; false means that the condition isn't met.

The results of relational operators are most useful with the flow of control statements covered in Section 7.4.

Some examples of relational expressions are:

```
3 < 4           TRUE
3 = 4           FALSE
3 >= 4          FALSE
3.5 <> 7.21     TRUE
```

The relational operators can be applied to user defined scalars:

```
VAR x: (red, green, blue);
IF x > red THEN ...
```


The statements after THEN will be executed if the value of x is green or blue.

6.7.3 Set Operators Modula manipulates sets with the set operators. These operators, used with all set types, are:

+	set union
-	set difference
*	set intersection
/	symmetric set difference
IN	set membership

The +, -, *, and / operators take two sets, both of which have the same type; the result is a set that has the same type as the operands.

The IN operator takes a scalar value and a set as arguments; it returns a boolean that indicates whether the scalar is a member of the set.

6.7.3.1 The Set Union Operator The + operator denotes set union. Given x + y the result is the set that contains all members of x and all members of y.

```
{1,3,5} + {5,7}     =  {1,3,5,7}
{1,3,5} + {1,3,5}   =  {1,3,5}
{1,3}   + {5}       =  {1,3,5}
{1,3,5} + {3}       =  {1,3,5}
{1,3,5} + {}        =  {1,3,5}
```

Some notes on set union:

- The operator is commutative, i.e., x+y = y+x
- {} is the additive identity, i.e., x + {} = {} + x = x
- x + x = x

6.7.3.2 The Set Difference Operator The - operator denotes set difference. Given x - y the result is the set that contains all members of x that are not members of y.

```
{1,3,5} - {5,7}    =  {1,3}
{1,3,5} - {1,3,5}  =  {}
{1,3}   - {5}      =  {1,3}
{1,3,5} - {3}      =  {1,3,5}
{1,3,5} - {}       =  {1,3,5}
```

Some notes on set difference:

- The operator is not commutative, generally x-y <> y-x.
- x - {} = x
- {} - x = {}
- x - x = {}

6.7.3.3 The Set Intersection Operator The * operator denotes set intersection. Given x * y the result is the set that contains all members of x that are also members of y.

```
{1,3,5} * {5,7}    =  {5}
{1,3,5} * {1,3,5}  =  {1,3,5}
{1,3}   * {5}      =  {}
{1,3,5} * {3}      =  {3}
{1,3,5} * {}       =  {}
```

Some notes on set intersection:

- The operator is commutative, i.e., x*y = y*x
- x*{} = {}*x = {}
- x*x = x

6.7.3.4 The Symmetric Set Difference Operator The / operator denotes symmetric set difference. Given x / y the result is the set that contains all members of x and all members of y, but not members of both.


```
{1,3,5} / {5,7}    =  {1,3,7}
{1,3,5} / {1,3,5}  =  {}
{1,3}   / {5}      =  {1,3,5}
{1,3,5} / {3}      =  {1,3}
{1,3,5} / {}       =  {1,3,5}
```

Some notes on symmetric set difference:

- The operator is commutative, i.e., x/y = y/x
- x/{} = {}/x = x
- x/x = {}

The / operator can be defined in terms of set union and set intersection as:

```
x / y  =  (x + y) - (x * y)
```

6.7.3.5 The Set Membership Operator The IN operator is used to test whether a particular value is a member of a set. The base type of the set must be compatible with the scalar value. A boolean value is returned: true indicates that the value is contained (is a member) of the set, false indicates that it is not.

```
TYPE
  SmallSet = SET OF [3..7];

CONST
  set = SmallSet{5,7};

 3 IN set  =  FALSE
 5 IN set  =  TRUE
 0 IN set  =  FALSE
10 IN set  =  FALSE
```

Figure 17. IN Operator Examples

NOTE: Values outside the base range of the set will not produce a run-time error; instead, they return FALSE.

6.7.4 Logical Operators The logical operators take boolean operands and return boolean results. These operators can be used to combine the results of relational operators to form

more complex conditions.

The logical operators are:

NOT, ~	logical negation
AND, &	logical conjunction
OR	logical disjunction

Note that the ampersand "&" is a synonym for AND, and the tilde "~" is a synonym for NOT. The results of the operators are summarised in Figure 18.

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	FALSE

Figure 18. Summary of Logical Operators

6.7.4.1 The AND Operator The result of AND is true if and only if both operands are true.

The exact definition of AND is slightly different from the above table, although the results are identical:

```
p AND q   =   IF p THEN q ELSE FALSE
```

This implies that if p is FALSE, q does not need to be (and will not be) evaluated. This type of operator is called a short-circuit operator.

This is most useful when the result of the right-hand argument would be undefined. Consider the following:

```
IF (x <> 0) AND (300/x > 2) THEN ...
```

If both operands are evaluated when x is 0, a divide by zero exception will occur. However, using short-circuit evaluation the second operand is not evaluated if x is 0 as the whole expression is known to be false.

6.7.4.2 The OR Operator The result of OR is true if either of its operands (or both) are true.

The OR operator is a short-circuit operator like AND. However, the definition of OR is:

```
p OR q   =   IF p THEN TRUE ELSE q
```

If the first operand is true the second operand is not evaluated as the whole of the expression is known to be true.

6.7.4.3 The NOT Operator The NOT operator takes one operand; if the operand is true the result is false, and vice versa.

Note that if x = FALSE, NOT x = TRUE, and NOT (NOT x) = FALSE, i.e., NOT (NOT x) = x.

Here is an example of using the logical and relational operators together. We can test the variable x being in the range 1 to 10 by using:

```
(1 <= x) AND (x <= 10)
```

To test for x being outside the range 1 to 10 we can logically negate the above test using NOT, i.e., x outside the range 1 to 10 is the same as NOT (x in the range 1 to 10). This can be written as:

```
NOT ((1 <= x) AND (x <= 10))
```

This is not the most elegant way to write the test; we will simplify the above expression in the next section.

6.7.5 The NOT Operator and Relations The NOT operator is redundant when used with a relational argument. For example, NOT (x=1) is the same as x<>1. The relations and their images under NOT are summarised in Figure 19.

relation	NOT relation
x = y	x <> y or y <> x
x > y	x <= y or y >= x
x < y	x >= y or y <= x
x <= y	x > y or y < x
x >= y	x < y or y > x
x <> y	x = y or y = x

Figure 19. The NOT Operator Used with Relations

6.7.6 De Morgan's Law A useful rule that relates AND, OR, and NOT is De Morgan's Law which states the equivalences:

```
(NOT p) AND (NOT q)   =  NOT (p OR q)
(NOT p) OR (NOT q)    =  NOT (p AND q)
```

Using the above two tables we can simplify the expression

```
NOT ((1 <= x) AND (x <= 10))
```

By using De Morgan's Law we can rewrite the expression as:

```
NOT (1 <= x) OR NOT (x <= 10)
```

And by using the table of relations under NOT this simplifies to:

```
(x < 1) OR (10 < x)
```

6.7.7 Relational Operators With Boolean Operands Modula provides the operators AND, OR, and NOT for manipulating boolean expressions; there are no operators for implication and equivalence. Remember that the boolean type can be defined as

```
TYPE BOOLEAN = (FALSE, TRUE)
```

Therefore, FALSE < TRUE. This relationship can be used to simulate the missing operators; the results are summarised in Figure 20.

x	y	x=y	x<>y	x<y	x>y	x<=y	x>=y
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE

Figure 20. Relational Operators with Boolean Operands

These can be interpreted as:

```
p <= q     implication     (p implies q)
q >= y     implication     (q implies p)
x = y      equivalence     (p is equivalent to q)
x <> y     not equivalent  (exclusive OR)
```

Although the following is legal:

```
IF Error <> FALSE THEN ...
```

it is better (and clearer) to write it as

```
IF NOT Error THEN ...
```

The following table gives a guideline for alternative expressions:

```
p = TRUE        p
p = FALSE       NOT p
p <> TRUE       NOT p
p <> FALSE      p
```

6.7.8 Relational Operators with Set Operands The relational operators can take set operands. The results are defined as:

```
x = y     x is the same set as y
x <> y    x is not the same set as y
x <= y    x is included in (a subset of) y,
          or y includes x
x >= y    y is included in (a subset of) x,
          or x includes y
```

6.7.8.1 The Set Equality Operator The = operator is defined as the set equality operator; it returns true if and only if the sets x and y contain exactly the same members:

```
{1,3,5} = {1,3,5}       TRUE
{1,3,5} = {1,3,5,7}     FALSE
{1,3}   = {1,3,5}       FALSE
```

6.7.8.2 The Set Inequality Operator The <> is the set inequality operator; it returns true if the set x contains one or more members that are not members of y (and vice versa):

```
{1,3,5} <> {1,3,5}      FALSE
{1,3,5} <> {1,3,5,7}    TRUE
{1,3}   <> {1,3,5}      TRUE
```

6.7.8.3 The Improper Set Inclusion Operators The <= operator is the improper set inclusion operator. Given x <= y, the result is true if every member of x is also a member of y. Note that y may contain members that are not in x.

```
{1,3,5} <= {1,3,5}      TRUE
{1,3,5} <= {1,3,5,7}    TRUE
{1,3,5} <= {1,3}        FALSE, 5 not a member of {1,3,5}
```

The >= operator denotes improper set inclusion as well, but x >= y means "y is included in (a subset of) x".

```
{1,3,5} >= {1,3,5}      TRUE
{1,3,5} >= {1,3,5,7}    FALSE, 7 not a member of {1,3,5}
{1,3,5} >= {1,3}        TRUE
```

The empty set is contained in all sets, i.e.,

```
{} <= x  is always true
x >= {}  is always true
```

6.7.8.4 The Proper Set Inclusion Operators The proper set inclusion operators < and > are not available in Modula. The main difference between proper and improper set inclusion is the case when the two sets being compared are the same. In this situation, improper set inclusion gives true, and proper set inclusion gives the value false. When the two sets differ, proper and improper set inclusion give the same results.

Case where sets are the same:

```
{1,3,5} <  {1,3,5}      FALSE
{1,3,5} <= {1,3,5}      TRUE
{1,3,5} >  {1,3,5}      FALSE
{1,3,5} >= {1,3,5}      TRUE
```

Case where sets differ:

```
{1,3,5} <  {1,3,5,7}    TRUE
{1,3,5} <= {1,3,5,7}    TRUE
{1,3,5} >  {1,3,5,7}    FALSE
{1,3,5} >= {1,3,5,7}    FALSE
```

Definition: A is a proper subset of B if and only if all members of A are contained in B and B differs from A.

The proper set inclusion operators can be simulated by:

```
x < y   as   (x <> y) AND (x <= y)
x > y   as   (x <> y) AND (x >= y)
```

6.8 Scope and Visibility

Modula-2, like many other modern programming languages, is a block-structured language. Block structure has proved to be useful for organising programs; it allows identifiers to be declared within a procedure body, and for those identifiers to be invisible outside the procedure.

Modula-2 allows you to define the scope and visibility of identifiers. Scope is the lifetime of the identifier, and visibility is where the identifier can be used. The following sections describe the ways in which identifiers can be controlled.

6.8.1 Local Identifiers The simplest way we can control the scope and visibility of an identifier is by a local declaration. This type of declaration can be used in modules and procedures, but we will only consider procedures here. An example of a local declaration is:

```
PROCEDURE Foolish;
  VAR i: INTEGER;
BEGIN
  i := 7
END Foolish
```

The declaration of i lies within the procedure, and is said to be a local variable of procedure Foolish.

We can now define the visibility of i. This variable is visible only between the BEGIN and END; outside this range any reference to i will be flagged as an error.

When Foolish is called, the variable is created; when the END (or RETURN) is encountered, the variable is destroyed. This defines the scope of i. All Foolish does is to create a variable, assign 7 to it, and then destroy it.

Note that you cannot declare two identifiers with the same name in the same scope, so, for example, the following is invalid:

```
PROCEDURE Error;
  CONST
    v = 200;
  VAR
    v: INTEGER;
  ...
```

As it happens, the scope and visibility of a local declaration extend over the same region, i.e., from BEGIN to END. In a later section we will see how to exercise more control of visibility and scope of identifiers.

Using the definition of scope above, we can declare a variable that lives throughout the whole of a program.


```
MODULE M;

  VAR i: INTEGER;     (* a global variable *)

  PROCEDURE A;
    VAR j: INTEGER;   (* a local variable *)
  BEGIN
    j := i
  END A;

BEGIN
  x := 3;
  A
END M.
```

The variable i is said to be global as it can be accessed at anytime and anywhere within the module M; it never loses its value.

7. Differences between Revision 2 and 3

This section describes the differences between the different revisions of the "Report on The Programming Language Modula-2". These reports are published in the book "Programming in Modula-2", second and third corrected edition, by Professor Niklaus Wirth.

The language changes are summarised in "Revisions and amendments to Modula-2, N. Wirth 1.2.84/14.5.84".1

7.1 Subrange Change

(Rev. 2, p. 145, Rev. 3, p. 148)

The syntax of the subrange type is changed from

```
SubrangeType =
  "[" ConstExpression ".." ConstExpression "]".
```

to

```
SubrangeType =
  [ident] "[" ConstExpression ".." ConstExpression "]".
```

The optional identifier allows the specification of the base type of the subrange, e.g., INTEGER [0..79].

7.2 Case Statement Change

(Rev. 2, p. 153, Rev. 3, p. 157)

The syntax of the case statement and variant record declaration are changed from

```
case = CaseLabelList ":" StatementSequence.
variant = CaseLabelList ":" FieldListSequence.
```

1. This paper can be found in Modula-2 News, issue 0, October 1984.

to

```
case = [CaseLabelList ":" StatementSequence].
variant = [CaseLabelList ":" FieldListSequence].
```

The inclusion of the empty case and empty variant allow the insertion of superfluous bars, similar to the empty statement allowing the insertion of superfluous semicolons, e.g.,

```
CASE file.state OF
| permanent: Close(file,reply)
| tentative: Remove(file,reply) |
ELSE
END
```

7.3 Field List Change

(Rev. 2, p. 147, Rev. 3, p. 149)

The syntax of the variant record type declaration with missing tag field is changed from

```
FieldList = ... "CASE" [ident ":"] qualident "OF"
```

to

```
FieldList = ... "CASE" [ident] ":" qualidenty "OF"
```

This means that declarations like

```
CASE BOOLEAN OF ...
```

must be rewritten as

```
CASE : BOOLEAN OF ...
```

The fact that the colon is always present makes it evident which part, if any, was omitted.

8. Internal Data Formats

This section describes the way that data is represented for each data type. This information is specific to the Modula-2 compiler provided, and should not be taken as a guide for all implementations.

For the 68000 processor, a byte is eight bits, a word is 16 bits, and a longword is 32 bits. Quadwords are 64 bits wide, but are not directly supported by the 68000; the quadword is only used for representing longreal values and is handled by special code sequences.

8.1 Character Representation

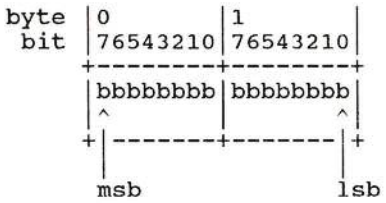
A character requires one byte of storage; the interpretation of the bits is character set dependent. The range offered by the character type is 0C..377C inclusive, i.e., CHR(0)..CHR(255). Note that if a 7-bit code is used, then the effective range is 0C..177C, the eighth bit is ignored.

8.2 Boolean Representation

Boolean values require one byte of storage. The value one represents TRUE, and zero represents FALSE.

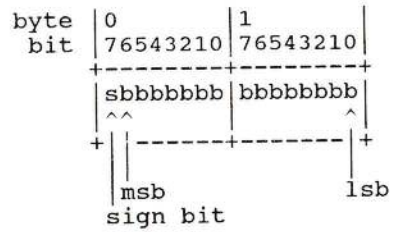
8.3 Cardinal Representation

Cardinal values are represented as an unsigned 16-bit binary number and require two bytes of storage. The range offered by cardinals is 0 to 65535 inclusive.



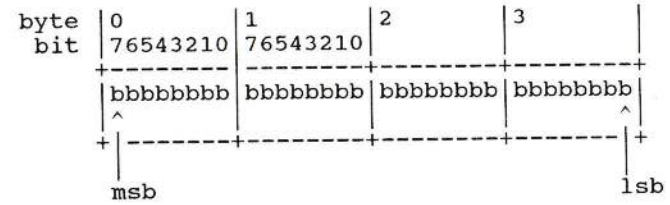
8.4 Integer Representation

Integer values are represented as a two's compliment 16-bit binary number and require two bytes of storage. The range offered by integers is -32768 to +32767 inclusive.



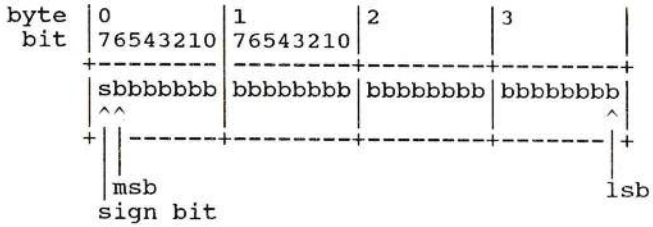
8.5 Long Cardinal Representation

Cardinal values are represented as an unsigned 16-bit binary number and require two bytes of storage. The range offered by cardinals is 0 to 4294967295 inclusive.



8.6 Long Integer Representation

Integer values are represented as a two's compliment 32-bit binary number and require two bytes of storage. The range offered by integers is -2147483647 to +2147483647 inclusive.



8.7 User Defined Scalar Types

Depending on the number of items in the type, user defined scalars will occupy either one or two bytes. If the number of enumerated items for the type is 256 or less, one byte of storage will be used; two bytes of storage will be used if there are over 256 items. The first enumerated item is allocated the value zero, and subsequent items are allocated ascending values.

8.8 Subrange Representation

Subrange types occupy the same number of bytes as their base type. For example, the subrange [0..9] will occupy two bytes as it has a cardinal base type (even though it could be represented in one byte).

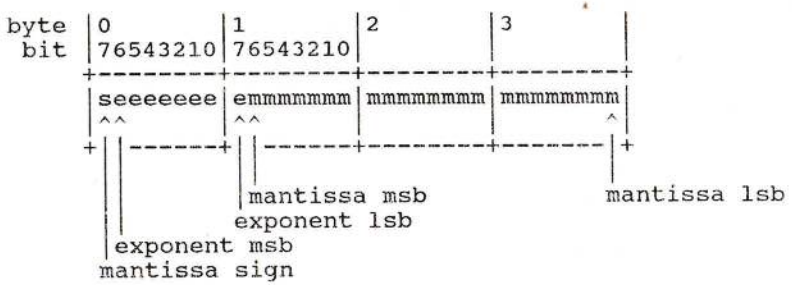
8.9 Pointer Representation

Pointers are represented as physical 68000 addresses. They require four bytes of storage. The 68000 has a 24-bit address bus and uses only the low three bytes of the address; the high order byte is ignored. The 68020 has a 32-bit address bus and uses all 32 bits for the address. If code is to be compatible between 68000 and 68020 processors do not use the high order byte to store extra information.

8.10 Real Representation

Real numbers conform to that recommended by the IEEE-CS Floating Point Arithmetic working group. A real number is represented as a 32-bit vector and requires four bytes of

storage. Real numbers are always normalised.



bit: 31 = sign of mantissa, 0 = +ve, 1 = -ve
30-23 = exponent, excess 127
22-00 = mantissa, binary point assumed between bits 23 and 21.

The number zero is represented by an exponent of zero, the mantissa and mantissa sign bit are ignored. Note that the number -0 may arise with an exponent of zero and a mantissa sign bit of one. This case is explicitly checked for by the real math subroutines. Real operations are always rounded. Overflow is always detected and results in a run-time error; underflow produces zero and computation continues with no error.

The range of numbers that can be represented is approximately:

$$1.7 * 10^{38} \text{ to } 5.8 * 10^{-39}$$

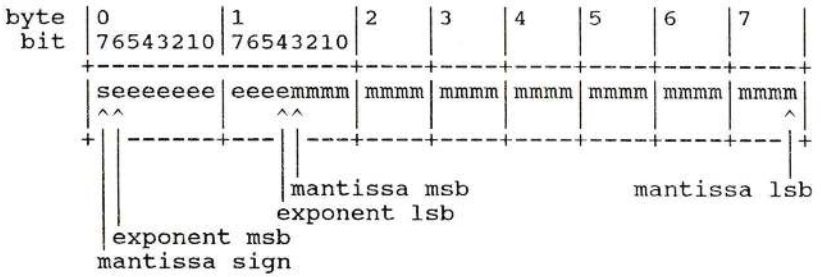
The precision around unity is

$$1 +/- 2^{-23} \text{ i.e., } 1 +/- 1.2 * 10^{-7}$$

giving seven significant digits.

8.11 Long Real Representation

Longreal numbers conform to that recommended by the IEEE-CS Floating Point Arithmetic working group. A longreal number is represented as a 64-bit vector and requires four bytes of storage. Longreal numbers are always normalised.




```
bit:  63    = sign of mantissa, 0 = +ve, 1 = -ve
      62-52 = exponent, excess 1023
      51-00 = mantissa, binary point
              assumed between bits 52 and 51.
```

The number zero is represented by an exponent of zero, the mantissa and mantissa sign bit are ignored. Note that the number -0 may arise with an exponent of zero and a mantissa sign bit of one. This case is explicitly checked for by the longreal math subroutines. Real operations are always rounded. Overflow is always detected and results in a run-time error; underflow produces zero and computation continues with no error.

The range of numbers that can be represented is approximately:

$$8.9 * 10^{307} \text{ to } 1.1 * 10^{-308}$$

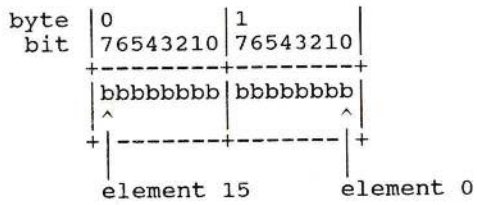
The precision around unity is

$$1 +/- 2^{-52} \text{ i.e., } 1 +/- 2.2 * 10^{-16}$$

giving sixteen significant digits.

8.12 Set Representation

Sets are represented in one word as:



8.13 Array Representation

According to the storage size of the element, arrays can be represented in two formats. If each element occupies one byte of storage, successive byte addresses will be used to hold the elements. So, for example, arrays of characters will use adjacent memory locations. All other arrays will be stored consecutively and word aligned.

9. The Syntax of Modula-2

```
ident =
    letter | {letter | digit}.
number =
    integer | real.
integer =
    digit {digit} |
    octalDigit {octalDigit} ("B" | "C") |
    digit {hexDigit} "H".
real =
    digit {digit} "." {digit} [ScaleFactor].
ScaleFactor =
    "E" ["+" | "-"] digit {digit}.
hexDigit =
    digit | "A" | "B" | "C" | "D" | "E" | "F".
digit =
    octalDigit | "8" | "9".
octalDigit =
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
string =
    '"' {character} '"' | "'" {character} "'".
qualident =
    ident {"." ident}.
ConstDeclaration =
    ident "=" ConstExpression.
ConstExpression =
    expression.
TypeDeclaration =
    ident "=" type.
type =
    SimpleType | ArrayType | RecordType | SetType |
    PointerType | ProcedureType.
SimpleType =
    qualident | enumeration | SubrangeType.
enumeration =
    "(" IdentList ")".
IdentList =
    ident {"," ident}.
SubrangeType =
    [qualident] "[" ConstExpression ".."
                    ConstExpression "]".
ArrayType =
    "ARRAY" SimpleType {"," SimpleType} "OF" type.
RecordType =
    "RECORD" FieldListSequence "END".
```



```
FieldListSequence =
    FieldList {";" FieldList}.
FieldList =
    [IdentList ":" type |
    "CASE" [ident] ":" qualident "OF"
      variant {"|" variant}
    ["ELSE" FieldListSequence] "END"].
variant =
    [CaseLabelList ":" FieldListSequence].
CaseLabelList =
    CaseLabels {"," CaseLabels}.
CaseLabels =
    ConstExpression [".." ConstExpression].
SetType =
    "SET" "OF" SimpleType.
PointerType =
    "POINTER" "TO" type.
ProcedureType =
    "PROCEDURE" [FormalTypeList].
FormalTypeList =
    "(" [[VAR] FormalType
         {"," [VAR] FormalType}]
    ")" [":" qualident].
VariableDeclaration =
    IdentList ":" type.
designator =
    qualident {"." ident | "[" ExpList "]" | "^"}.
ExpList =
    expression {"," expression}.
expression =
    SimpleExpression [relation SimpleExpression].
relation =
    "=" | "#" | "<" | "<=" | ">" | ">=" | "<>" | "IN".
SimpleExpression =
    ["+" | "-"] term {AddOperator term}.
AddOperator =
    "+" | "-" | "OR".
term =
    factor {MulOperator factor}.
MulOperator =
    "*" | "/" | "DIV" | "MOD" | "AND".
factor =
    number | string | set |
    designator [ActualParameters] |
    "(" expression ")" | "NOT" factor.
set =
```

```
    [qualident] "{" [element {"," element}] "}".
element =
    expression [".." expression].
ActualParameters =
    "(" [ExpList] ")".
statement =
    [assignment | ProcedureCall |
     IfStatement | CaseStatement |
     WhileStatement | RepeatStatement |
     LoopStatement | ForStatement |
     WithStatement | "EXIT" |
     "RETURN" [expression]].
assignment =
    designator "=" expression.
ProcedureCall =
    designator [ActualParameters].
StatementSequence =
    statement {";" statement}.
IfStatement =
    "IF" expression "THEN" StatementSequence
    {"ELSIF" expression "THEN" StatementSequence}
    ["ELSE" StatementSequence] "END".
CaseStatement =
    "CASE" expression "OF" case {"|" case}
    ["ELSE" StatementSequence] "END".
case =
     [CaseLabelList ":" StatementSequence].
WhileStatement =
    "WHILE" expression "DO" StatementSequence "END".
RepeatStatement =
    "REPEAT" StatementSequence "UNTIL" expression.
ForStatement =
    "FOR" ident ":=" expression "TO" expression
    ["BY" ConstExpression] "DO"
    StatementSequence "END".
LoopStatement =
    "LOOP" StatementSequence "END".
WithStatement =
    "WITH" designator "DO" StatementSequence "END".
ProcedureDeclaration =
    ProcedureHeading ";" block ident.
ProcedureHeading =
    "PROCEDURE" ident [FormalParameters].
block =
    {declaration} ["BEGIN" StatementSequence] "END".
declaration =
```



```
    "CONST" {ConstDeclaration ";"} |
    "TYPE" {TypeDeclaration ";"} |
    "VAR" {VariableDeclaration ";"} |
    ProcedureDeclaration ";" |
    ModuleDeclaration ";".
FormalParameters =
    "(" [FPSection {";" FPSection}]
    ")" [":" qualident].
FPSection =
    ["VAR"] IdentList ":" FormalType.
FormalType =
    ["ARRAY" "OF"] qualident.
ModuleDeclaration =
    "MODULE" ident [priority] ";"
    {import} {export} block ident.
priority =
    "[" ConstExpression "]".
export =
    "EXPORT" ["QUALIFIED"] IdentList ";".
import =
    ["FROM" ident] "IMPORT" IdentList ";".
DefinitionModule =
    "DEFINITION" "MODULE" ident ";"
    {import} {definition} "END" ident ".".
definition =
    "CONST" {ConstDeclaration ";"} |
    "TYPE" {ident ["=" type] ";"} |
    "VAR" {VariableDeclaration ";"} |
    ProcedureHeading ";".
ProgramModule =
    "MODULE" ident [priority] ";"
    {import} block ident ".".
CompilationUnit =
    DefinitionModule |
    ["IMPLEMENTATION"] ProgramModule.
```

10. Modula-2 Compiler Error Codes and Restrictions

10.1 Syntax Errors

10 identifier expected
11 , comma expected
12 ; semicolon expected
13 : colon expected
14
15) right parenthesis expected
16] right bracket expected
17 } right brace expected
18 = equal sign expected
19 := assignment expected
20 END expected
21
22 (left parenthesis expected
23 OF expected
24 TO expected
25 DO expected
26 UNTIL expected
27 THEN expected
28 MODULE expected
29 illegal digit or number too large
30 IMPORT expected
31 factor starts with illegal symbol
32 identifier, (, or [expected
33 identifier, ARRAY, RECORD, SET, POINTER, PROCEDURE, (, or [expected
34 Type followed by illegal symbol
35 statement starts with illegal symbol
36 declaration followed by illegal symbol
37 statement part is not allowed in definition module
38 export list not allowed in program module
39 EXIT not inside a LOOP statement
40 illegal character in number
41 number too large
42 comment without closing *)
44 expression must contain constant operands only
45 control character within string

10.2 Undefined

50 identifier not declared or not visible

10.3 Class and Type Errors

51 object should be a constant
52 object should be a type
53 object should be a variable
54 object should be a procedure
55 object should be a module
56 type should be a subrange
57 type should be a record
58 type should be an array
59 type should be a set
60 illegal base type of set
61 incompatible type of label or of subrange bound
62 multiply defined case (label)
63 low bound > high bound
64 more actual than formal parameters
65 fewer actual than formal parameters

10.4 Mismatch between Parameter Lists in Definition and in Implementation Modules

66 more parameters in implementation than in definition
67 parameters with equal types in implementation have different types in definition
68 mismatch between VAR specifications
69 mismatch between type specifications
70 more parameters in definition than in implementation
71 mismatch between result type specifications
72 function in definition, pure procedure in implementation
73 procedure in definition has parameters, but not in implementation
74 code procedure cannot be declared in definition module
75 illegal type of control variable in FOR statement
76 procedure call of a function
77 identifiers in heading and at end do not match
78 redefinition of a type that is declared in definition part
79 imported module not found
80 unsatisfied export list entry
81 illegal type of procedure result
82 illegal base type of subrange

83 illegal type of case expression
84 writing of symbol file failed
85 keys of imported symbol files do not match
86 error in format of symbol file
88 symbol file not successfully opened
89 procedure declared in definition module, but not in implementation

10.5 Implementation Restrictions of Compiler

90 in {a..b}, if a is a constant, b must also be a constant
91 code procedure can have at most 8 bytes of code
92 too many cases
93 too many exit statements
94 index type of array must be a subrange
95 subrange bound must be less than 2^15
96 too many global modules
97 too many procedures in definition module
98 too many structure elements in definition module
99 too many variables or record too large

10.6 Multiple Definition

100 multiple definition within the same scope

10.7 Class and Type Incompatibilities

101 illegal use of type
102 illegal use of procedure
103 illegal use of constant
104 illegal use of type
105 illegal use of procedure
106 illegal use of expression
107 illegal use of module
108 constant index out of range
109 indexed variable is not an array, or the index has the wrong type
110 record selector is not a field identifier
111 dereferenced variable is not a pointer
112 operand type incompatible with sign inversion
113 operand type incompatible with NOT
114 x IN y: type(x) # basetype(y)
115 type of x cannot be the basetype of a set, or y is not a set
116 {a..b}: type of either a or b is not equal to the base

type of the set
117 incompatible operand types
118 operand type incompatible with *
119 operand type incompatible with /
120 operand type incompatible with DIV
121 operand type incompatible with MOD
122 operand type incompatible with AND
123 operand type incompatible with +
124 operand type incompatible with -
125 operand type incompatible with OR
126 operand type incompatible with relation
127 procedure must have level 0
128 result type of P does not match that of T
129 mismatch of a parameter of P with the formal type list of T
130 procedure has fewer parameters than the formal type list
131 procedure has more parameters than the formal type list
132 assignment of a negative integer to a cardinal variable
133 incompatible assignment
134 assignment to non-variable
135 type of expression in IF, WHILE, UNTIL clause must be BOOLEAN
136 call of an object which is not a procedure
137 type of VAR parameter is not identical to that of actual parameter
139 type of RETURN expression differs from procedure type
140 illegal type of CASE expression
141 step in FOR clause cannot be 0
142 illegal type of control variable
144 incorrect type of parameter of standard procedure
145 this parameter should be a type identifier
146 string is too long
147 incorrect priority specification

10.8 Name Collision

150 exported identifier collides with declared identifier

10.9 Implementation Restrictions of System

200 (not yet implemented)
201 integer too small for sign inversion
202 set element outside word range
203 overflow in multiplication
204 overflow in division

205 division by zero, or modulus with negative value
206 overflow in addition
207 overflow in subtraction
208 cardinal value assigned to integer variable too large
209 set size too large
210 array size too large
211 address too large (compiler error?)
212 character array component cannot correspond to VAR parameter
213 illegal store operation (compiler error?)
214 set elements must be constants
215 expression too complex (stack overflow)
216 double precision multiply and divide are not implemented
222 output file not opened (directory full?)
223 output incomplete (disk full?)
224 too many external references
225 too many strings
226 program too long
230 expression not loadable (implementation restriction)
231 expression not addressable (implementation restriction)
232 expression not allowed (implementation restriction)
234 register reservation error
235 illegal selector for constant index / field
236 too many nested WITH (> 4)
237 illegal operand (implementation restriction)
238 illegal size of operand (implementation restriction)
239 type should be LONGREAL
240 parameter should be dynamic array parameter
241 illegal type for floating point operation
244 implementation restriction for floating point comparison