

TopSpeed® Modula-2

For IBM® Personal Computers and Compatibles

Language and Library Reference

TopSpeed Corporation

Copyright© 1990-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

10 9 8 7 6 5 4 3 2 1

Contents

CHAPTER 1

INTRODUCTION	20
Manual Structure	21
Typographic Conventions	21
Modula-2 Syntax and Semantics	23
Syntax	23
Static Semantics	23
Dynamic Semantics	23
The TopSpeed Implementation of Modula-2	24
Textual Topics	25
Tokens	25
Keywords	25
Delimiters	26
Generic tokens	26
Separators	27
Syntax Descriptions	28
Declarations and Visibility	30
Alias Declarations	31
Predefined Identifiers	31
Types	32
Numeric Types	32
Ordinal Types	33
Subrange Types	33
Set Types	34
Array Types	34
Record Types	35
Pointer Types	36
Type Compatibility	37
Objects and Values	38

Constants	38
Set Values	39
Designators	40
Expressions	41
Statements	43
Assignment Statement	43
IF Statement	43
CASE Statement	44
WHILE Statement	44
REPEAT Statement	44
LOOP Statement	45
FOR Statement	45
WITH Statement	46
GOTO Statement	46
Procedures	47
Bodies	47
External Procedures	49
Calling Procedures	49
Procedure Types	50
Predefined Procedures	51
Modules	54
Server Module	54
Importing	56
Local Module	57

CHAPTER 3

OBJECT-ORIENTED EXTENSIONS 58

What is Object Oriented Programming?	59
Encapsulation	59
Inheritance	60
Polymorphism	61
Class Declarations	63
Class Interface Declarations	63
Class Implementation Declarations	64

Declaring Class Variables	66
Invoking Methods	67
Inheritance	68
Over-riding Methods	69
Invoking a Base Class Method	70
Multiple Inheritance	70
Aliasing	72
Compatibility Rules	73
The Checked Guard Operator	74
The IS Operator	76
Virtual and Static Methods	78
Virtual Methods	78
Static Methods	79
Case Study	83

CHAPTER 4

LIBRARY SUMMARY 91

Overview	92
Library Summary	93
BiosIO — (DOS Only)	93
Dev — (OS/2 Only)	93
Dos — (OS/2 Only)	93
Err — (OS/2 Only)	93
FIO —	94
FIOR —	94
FloatExc —	94
FormIO	94
Isp — (OS/2 Only)	94
Gpi — (OS/2 Only)	95
Graph	95
IO	95
Kbd — (OS/2 Only)	96
Lib —	96
LIM	96

MATHLIB —	96
Mou — (OS/2 Only)	97
MsMouse	97
OS2DEF — (OS/2 Only)	97
Pic — (OS/2 Only)	97
PMErr — (OS/2 Only)	97
Process	97
ShtHeap	98
Spl — (OS/2 Only)	98
Storage	98
Str	98
SYSTEM	99
Vio — (OS/2 Only)	99
Win — (OS/2 Only)	99
Window —	100
OS/2 Library Interface	101
Modula-2 Core Library	102

CHAPTER 5

LIBRARY REFERENCE 103

MODULE BiosIO (DOS Only)	103
BiosIO Reference	104
KeyPressed — Has a key been pressed?	104
RdKey — Read character without screen echo	104
RdChar — Read character with screen echo	104
KBFlags — Determine state of 'modifier' keys	105
MODULE FIO	106
Introduction	106
File Handles	106
File Positioning	106
Buffers	107
Definitions in FIO.DEF	107
FIO Reference	116
Append — Open existing file at end	116

AppendHandle	— Add a handle to the file system	117
AppendStream	— Get file handle from stream	117
AssignBuffer	— Assign a buffer to an open file	117
ChDir	— Change Directory	118
Close	— Close an open file	118
Create	— Create and open new file	119
Erase	— Delete a file	119
Exists	— Check for file existence	120
Flush	— Flush buffer	120
GetCurrentDate	— Get today's date	120
GetDir	— Get current working directory	121
GetDrive	— Get current drive	121
GetFileDate	— Finds the file date	121
GetFileStamp	— Get file stamp	122
GetPos	— Get current file position	122
GetStreamPointer	— Get stream buffer handle	122
IOresult	— Determine result of last operation	123
MkDir	— Make a new directory	123
Open	— Open an existing file	124
OpenRead	— Open an existing file for Reading	125
RdBin	— Read binary data	126
RdItem	— Read separated string from file	126
Rdsimpletype	— Formatted input	127
RdStr	— Read a string from a file	128
ReadFirstEntry	— Begin directory scan	129
ReadNextEntry	— Find next matching file	131
Rename	— Rename a file	131
RmDir	— Remove a directory	131
Seek	— Set new file position	132
SetDrive	— Set default drive	132
SetFileDate	— Change a file's date	132
Size	— Determine the size of a file	132
ThreadEOF	— Get thread's EOF status	132

ThreadOK — Get thread's OK status	133
Truncate — Truncate a file	133
WrBin — Write binary data	134
WrCharRep — Write character repeatedly	134
WrLn — Start a new line	134
Wrsimpletype — Formatted output	135
WrStr — Write String to file	138
WrStrAdj — Write formatted string	138
MODULE FIOR	139
Introduction	139
FIOR Reference	139
AddExtension — Manipulate extensions	140
ChangeExtension	140
RemoveExtension	140
Create — Create file using redirection	141
Erase — Delete file using redirection	141
ExpandPath — Expand path using redirection file	141
FindNewPath — Find path for file using redirection	142
FindPath	142
IOresult — Determine error code	142
IsExtension — Check file extension	143
MakePath — Path and file manipulation	143
SplitPath	143
Open — Open file using redirection	143
ReadRedirectionFile — Establish new redirection file	144
MODULE FloatExc	145
Introduction	145
FloatExc Reference	145
DisableExceptionHandling — Disable Exception Handling	145
EnableExceptionHandling — Enable Exception Handling	145
InstallHandler — Installs exception handler	146
MODULE FormIO	147
Introduction	147

FormIO Reference	148
WrF — Output format string	148
WrF1 — Output format string and one argument	148
WrF2 — Output format string and two arguments	148
WrF3 — Output format string and three arguments	149
WrF4 — Output format string and four arguments	149
WrF5 — Output format string and five arguments	149
Introduction	150
Variables	150
Supported Video Adapters and Screen Modes	152
Colors	154
Graph Reference	156
Arc — Draw an arc	156
Circle — Draw a circle	157
Disc	157
TrueCircle	157
TrueDisc	157
ClearScreen — Clear the screen	158
Cube — Draw a cube	158
DisplayCursor — Show/Hide cursor	159
Ellipse — Draw an ellipse	159
FloodFill — Fill region	160
StackFill	160
GetBkColor — Get/Set current background color	161
SetBkColor	161
GetFillMask — Get current fill mask	161
GetImage — Get pixel area from screen	162
GetLineStyle — Get current line style	162
GetTextColor — Get current text color	162
GetTextPosition — Get current text output position	163
GetVideoConfig — Get video configuration	163
GraphMode — Switch to graphics mode	163
HLine — Draw a horizontal line	164

ImageSize — Determine memory size of pixel area	164
InitScreentype — Initialization routines	164
Line — Draw a line	165
OutText — Output text to screen	165
Pie — Draw a pie chart slice	166
Plot — Set a single pixel	166
Point — Get the value of a single pixel	167
Polygon — Draw a polygon	167
PutImage — Put a pixel area on to the screen	168
Rectangle — Draw rectangle	169
RemapPalette — Specify new palette element for adapter	170
RemapAllPalette	170
SelectPalette — Choose a new palette for adapter	171
SetActivePage — Switch video pages	172
SetVisualPage	172
SetClipRgn — Set clipping region	173
SetFillMask — Set new fill mask	174
SetLineStyle — Set new line style	175
SetTextColor — Set new text color	175
SetTextPosition — Set new text output position	176
SetTextWindow — Set text window boundaries	176
SetVideoMode — Set new video mode	177
TextMode — Switch to text mode	177
Wrapon — Toggle text wrapping	178
MODULE IO	179
Introduction	179
The IO Definition File	179
Redirecting Input and Output	179
IO Reference	182
EndOfRd — Skip delimiters	182
KeyPressed — Determine if a key has been pressed	182
RdItem — Read delimited input string	183
RdKey — Read a single character	184

RdLn — Skip rest of input line	184
Rdsimpletype — Formatted input of simple types	185
RdStr — Input string	185
RedirectInput — I/O Redirection to file	186
RedirectOutput	186
ThreadOK — Get OK variable	186
WrCharRep — Write repeated character	186
WrLn — Start new line	187
Wrsimpletype — Formatted output of simple types	187
WrStr — Output string	189
WrStrAdj — Adjust length of string before output	189
MODULE Lib	190
Introduction	190
Lib Reference	191
AddAddr — Address Arithmetic Procedures	191
SubAddr	191
IncAddr	191
DecAddr	191
Compare — Compare memory blocks	192
Cpuld — Determine CPU type	193
Delay — Timed delay	193
DisableBreakCheck — Switch off CtrlBrk	194
Dos — Call DOS	194
EnableBreakCheck — Switch on CtrlBrk checking	194
Environment — Get Environment Variable by Number	195
EnvironmentFind — Find environment variable by name	195
Exec — Execute another program	196
ExecCmd — Execute a DOS or OS/2 command	197
Execute — Execute a program	197
FatalError — Terminate process with error message	197
Fill — Fill memory with byte	198
GetDate — System date & time procedures	198
GetTime	198

SetDate	198
SetTime	198
HashString — Compute hash value from string	199
Intr — Software Interrupt	199
MathError — MATHLIB Error Handling	199
Move — move memory bytes	200
NoSound — Turn off speaker	200
ParamCount — Command line parameter count	200
ParamStr — Get a command line argument	201
QSort — General purpose sorting routines	201
RAND — Generate random Real Number	203
RANDOM — Generate random Cardinal	203
RANDOMIZE — Initialize Random number generator	203
ScanL — Search Memory for Byte Left	204
ScanNeL — Search Memory for unequal byte Left	204
ScanNeR — Search Memory for unequal Byte Right	205
ScanR — Search Memory for Byte Right	205
SetJmp — Long Jumps	206
LongJmp	206
SetReturnCode — Specify return code	207
Sound — Sound speaker	208
SysErrno — Get last error code	208
Terminate — Link termination procedures	208
UserBreak — Abort Program	209
WordFill — Fill memory with word	209
WordMove — Move memory words	210
WrDosError — Output error message	210
MODULE LIM	211
Introduction	211
Variables	211
LIM Reference	211
AllocatePages — Allocate expanded pages	211
DeAllocatePages — Deallocate expanded pages	211

FreePages — Get memory available	212
GetPageFrame — Get page frame	212
GetStatus — Return error status of the EMS driver	212
MapPage — Map physical and logical pages	213
MODULE MATHLIB	214
Introduction	214
Error Handling	214
MATHLIB Reference	215
Exp — Exponential function	215
Log — Logarithmic Functions	215
Log10	215
LongToBcd — Convert real number to BCD format	216
BcdToLong	216
MathError — MATHLIB Error Handling	216
Mod — Real Modulus	216
Pow — Raise to the power	217
Rexp — Split Real Number into Mantissa and Exponent	217
Sin — Trigonometric Functions	218
Cos	218
Tan	218
ASin	218
ACos	218
ATan	218
ATan2	218
SinH — Hyperbolic Function	219
Cosh	219
TanH	219
Sqrt — Square Root	219
MODULE MsMouse	220
Introduction	220
Data Types and Constants	220
MsMouse Reference	223
Cursor — Set the cursor mode	223

DriverSize — Bytes needed to store driver state	223
GetMotion — Distance moved since last call	223
GetPage — Display Page	224
SetPage	224
GetPress — Get button press information	224
GetRelease — Get button release information	224
GetSensitivity — Get/Set mouse sensitivity settings	225
SetSensitivity	225
GetStatus — Get mouse status	225
LightPen — Set light pen emulation	225
Reset — Reset mouse driver	225
SaveDriver — Save/Restore Driver state	226
RestoreDriver	226
SetDouble — Set double speed threshold	226
SetGraphCursor — Set graphics cursor shape	226
SetInterrupt — Set interrupt call mask	226
SetMickeys — Define mickeys per click ratio	227
SetPosition — Set cursor position	227
SetRange — Set area for mouse cursor	227
SetTextCursor — Set text cursor shape	227
UpdateScreen — Specify update area	227
MODULE Process	228
Introduction	228
Skeleton Multi-Thread Programs	229
Processes	229
Example of Use	229
Process Reference	233
Awaited — Is there anyone waiting?	233
Delay — Wait for a time	233
Init — Initialize a signal	233
Lock — Lock current process	233
Notify — Reschedule when possible	234
SEND — Raise a signal	234

StartProcess — Start a new process	234
StartScheduler — Start the Scheduler	234
StopProcess — Stop a process	235
StopScheduler — Stops the Scheduler	235
Unlock — Unlock current process	235
WAIT — Wait for a signal	235
MODULE ShtHeap	236
Introduction	236
Constants	236
ShtHeap Reference	236
Allocate — Allocate block from short heap	237
Free — Free a block back to the short heap	237
Increase — Increase size of short heap	237
Initialize — Initialize a short heap	237
Largest — Report largest available block	238
Test — Test the integrity of a short heap	238
Total — Total free space	238
MODULE Storage	239
Introduction	239
Variables	239
Near and Far Variants	239
Storage Reference	241
ALLOCATE — Allocate memory from the heap	241
NearAllocate — Allocate memory from the near heap	241
FarAllocate — Allocate memory from the far heap	241
Available — Is there enough memory?	242
NearAvailable — Is there enough memory?	242
FarAvailable — Is there enough memory?	242
DEALLOCATE — Return memory to the heap	242
NearDeallocate — Return memory to the near heap	243
FarDeallocate — Return memory to the far heap	243
HeapAllocate — Allocate memory from a heap	243
NearHeapAllocate — Allocate memory from near heap	244

FarHeapAllocate — Allocate memory from far heap	244
HeapAvail — Largest available block	245
NearHeapAvail — Largest available block	245
FarHeapAvail — Largest available block	245
HeapChangeAlloc — Change size of allocation without relocation	246
NearHeapChangeAlloc — Change size of allocation without relocation	246
FarHeapChangeAlloc — Change size of allocation without relocation	247
HeapChangeSize — Change size of allocation	247
NearHeapChangeSize — Change size of allocation	248
FarHeapChangeSize — Change size of allocation	248
HeapDeallocate — Free memory back to a heap	249
NearHeapDeallocate — Free memory back to a heap	249
FarHeapDeallocate — Free memory back to a heap	250
HeapTotalAvail — Total available memory on heap	250
NearHeapTotalAvail — Total available memory on heap	250
FarHeapTotalAvail — Total available memory on heap	251
MakeHeap — Create a sub-heap	251
NearMakeHeap — Create a sub-heap	251
FarMakeHeap — Create a sub-heap	252
SegAllocate — Allocate a segment	252
SegDeallocate — Deallocate a segment	252
MODULE Str	253
Introduction	253
Strings in Modula-2	253
Definition File	253
Conversion Procedures	254
Str Reference	255
Append — Append a string	255
Caps — Convert to upper case	255
CardToStr — Convert Cardinal to String	256
CharPos — Find character in string	256
Compare — Compare two strings	257
Concat — Join two strings	257

Copy — Copy a string	258
Delete — Delete character from a string	258
FindSubStr — Find all matches	259
FixRealToStr — Convert Real to String without exponent	259
Insert — Insert characters into string	260
IntToStr — Convert Integer to String	260
Item — Extract sub-field	261
ItemS — Extract subfield	261
Length — Length of string	262
Match — Wildcard string match	262
NextPos — Locate next substring	263
Pos — Locate substring	263
Prepend — Prepend string	263
RCharPos — Find character in string from right	264
RealToStr — Convert Real to String	264
Slice — Extract substring	265
StrToC — Convert string to C	265
StrToCard — Convert String to Cardinal	266
StrToInt — Convert String to Integer	266
StrToPas — Convert string to Pascal	267
StrToReal — Convert String to Real	267
Subst — Replace substring	267
MODULE SYSTEM	268
Introduction	268
The Registers Record	268
Coroutines	268
Low-level Procedures	269
SYSTEM Reference	270
DI — Disable interrupts	270
EI — Enable interrupts	270
GetFlags — Read processor flags	270
In — Read byte value from port	270
InW — Read word value from port	271

IOTRANSFER — Transfer to I/O co-process	271
NEWPROCESS — Start new process	272
Out — Write byte to value port	272
OutW — Write word to value port	272
SetFlags — Set processor flags	272
TRANSFER — Transfer to co-process	273
MODULE Window	274
Introduction	274
Terminology	274
Constants and Types	275
Types of Window Procedures	275
Window Manipulation Procedures	276
Positioning Procedures	278
Contents Manipulation Procedures	278
Multi-thread Procedure	279
Window Reference	282
At — Get window at position	282
Change — Alter window's size and position	282
Clear — Clear window	282
ClrEol — Clear to end of line	282
Close — Close window	283
ConvertCoords — Convert window coordinates to screen coordinates	283
CursorOn — Turn cursor on/off	283
CursorOff	283
DelLine — Delete line	283
DirectWrite — Write directly to screen	284
GotoXY — Set new position in window	284
Hide — Hide a window	284
Info — Get information on window	284
InsLine — Insert blank line	285
ObscuredAt — Is window obscured at this position?	285
Open — Open a new window	285
PaletteOpen	285

PaletteColor — Get current palette color	286
PaletteColorUsed — Is palette color used in window?	286
PutBeneath — Put one window beneath another	286
PutOnTop — Display window on top of others	287
RdBufferLn — Read line directly from window buffer	287
SetFrame — Set new frame for window	288
SetPalette — Set palette for palette window	288
SetPaletteColor — Set new palette color	288
SetProcessLocks — Multi-process support	289
SetTitle — Set title for window	289
SetWrap — Toggle end of line wrap	290
Snapshot — Capture data from screen	290
TextBackground — Set text background color	290
TextColor — Set text color	290
Top — Get top windows	290
Use — Switch to another window	291
Used — Get current window	291
WhereX — Determine current position in window	291
WhereY	291
WrBufferLn — Write line directly to window buffer	292
File Format	293
General Errors	293
Window Module	294
DOS Process Module	294
OS/2 Process Module	294
Memory Management	295
File I/O	295
Process Module	296

CHAPTER 1

INTRODUCTION

This manual is intended to perform two functions:

- To act as a reference to the TopSpeed Modula-2 language.
- To act as a reference for the modules and procedures making up the TopSpeed Modula-2 Library.

If you have little, or no, programming experience, please read *The TopSpeed Modula-2 Language Tutorial* before attempting to use this manual.

Manual Structure

This manual is divided into the following chapters:

- Introduction** This chapter, the Introduction, serves to acquaint you with this manual and to describe the conventions and notation you will experience in the remainder.
- Language Reference** This chapter describes the TopSpeed implementation of Modula-2. This description is aimed specifically at those persons already familiar with programming practises and terminology; it is not intended for 'novice' users.
- Object Oriented Extensions** This chapter describes the TopSpeed Modula-2 object oriented extensions, and provides a short tutorial covering their usage.
- Library Summary** This chapter summarizes the modules supplied with the TopSpeed Modula-2 library. details are given as to the overall functionality of each module. This chapter is intended as a 'reminder' for those persons experienced in TopSpeed Modula-2.
- Library Reference** This chapter provides an indepth analysis and explanation of the individual procedures making up the modules of the TopSpeed Modula-2 library. This chapter should be used as a reference for those persons wishing to use specific procedures.

Typographic Conventions

The following typographic conventions are used throughout this manual:

- Italics* are used for emphasis and to introduce new concepts.
- Typewriter is used for program examples and other items which would appear on the screen.
- Keyboard* is used to show keys (such as *F1* or *RETURN*) which you press at the keyboard.
- Bold Italics*** are used in syntax descriptions.

When it is necessary to show a combination of keys, the following convention is used:

Alt-C

This means you should hold down the *Alt* key and press *C*.

Where example screen outputs are shown, the illustration is boxed.

CHAPTER 2

LANGUAGE REFERENCE

This chapter introduces and describes the language used by TopSpeed Modula-2. The information given in this chapter assumes that you have a good knowledge of programming principles and practices. This information is intentionally concise, and relies upon your programming experience to furnish any required background information.

If at any time, you feel that you lack sufficient knowledge, please refer to *The TopSpeed Modula-2 Tutorial*.

Modula-2 Syntax and Semantics

In order to compile and run correctly, all programs written with TopSpeed Modula-2 must obey three sets of rules:

- The *syntax rules* of the language - the textual format of source programs.
- The *static semantic rules* of the language - the interpretation of specific program elements by the compiler.
- The *dynamic semantic rules* of the language - describing the behavior of running programs.

Syntax

All TopSpeed Modula-2 programs must conform to the language syntax described in this chapter. The syntax of the language specifies the textual structure of your program.

Static Semantics

As well as textual structure, programs must conform to the static, compile-time semantics of TopSpeed Modula-2. These semantics define the meaning of program identifiers and the way in which they are used. The compiler checks programs to ensure that they conform to these rules.

Dynamic Semantics

All programs must conform to TopSpeed Modula-2's dynamic, run-time semantics (for example, array indexes must stay within their defined limits). The compiler cannot check such aberrations, and it is your decision (through the use of pragmas) whether these are checked while your program is running.

If your program does not conform to the run-time semantics there is no guarantee that it will run correctly.

The TopSpeed Implementation of Modula-2

In the absence of an agreed standard for Modula-2, the TopSpeed implementation of the language is based, upon the language described in Niklaus Wirth's *Programming in Modula-2 (3rd Edition, 1985)*.

The main deviations from this language definition are:

- The 80x86 family of processors uses a *segmented address* architecture, (the operation of this architecture is fully explained in ***The TopSpeed Users Guide***). A physical address is obtained from a 16-bit *offset* in combination with a 16-bit *segment* value (in real mode), or a 16-bit *selector* value (in protected mode). This means that address arithmetic must be handled differently from that specified in *Programming in Modula-2*.

Textual Topics

This section describes the basic elements of Modula-2 and the format used for syntax descriptions.

Tokens

Tokens are the simplest textual elements of Modula-2. They are the building blocks from which all other syntax elements are built. All Modula-2 tokens are sequences of characters from the ASCII character set and are divided into four classes:

Keywords fixed sequences of letters.

Delimiters fixed sequences of non-alphanumeric characters.

Generic tokens variable-length sequences of characters and symbols that form the elements operated on by the statements and expressions of the language.

Separators sequences of characters which can occur between any of the above.

The formal syntax descriptions later in this chapter reference tokens by these names, and in certain cases, by the names of defined sub-classifications.

Keywords

The *keywords* of TopSpeed Modula-2 are:

AND	FROM	QUALIFIED
ARRAY	GOTO†	RECORD
BEGIN	IF	RETURN
BY	IMPLEMENTATION	REPEAT
CASE	IMPORT	SET
CLASS†	IN	THEN
CONST	INLINE†	TO
DEFINITION	IS	TYPE
DIV	LABEL†	UNTIL
DO	LOOP	VAR
ELSE	MOD	VIRTUAL†
ELSIF	MODULE	WHILE
END	NOT	WITH
EXIT	OF	
EXPORT	OR	
FORWARD	PROCEDURE	

Keywords marked with a † do not appear in Niklaus Wirth's book, they are particular to TopSpeed Modula-2 and are explained in detail later in this chapter.

Delimiters

The *delimiters* are:

+	-	*	/
:=	&	.	,
:	:	()
[]	{	}
^	~	=	#
<>	<	<=	>
>=	<<†	>>†	
..			

Delimiters marked with a † do not appear in Niklaus Wirth's book, they are particular to TopSpeed Modula-2 and are explained in detail later in this chapter.

Like *keywords*, delimiters must be used as individual elements, i.e., := is two delimiters together (an error) where as := is a valid single delimiter.

The following tokens are synonyms, i.e., they can be regarded as interchangeable:

AND	and	&
NOT	and	~
#	and	<>

Generic tokens

The generic tokens used by TopSpeed Modula-2 are:

Identifier a list of *letters* ('A' to 'Z', 'a' to 'z' and '_') and *digits* ('0' to '9') starting with a letter. The *keywords* are excluded from the list of possible identifiers. For example:

```
x
NameOfReciPient
_main
ThisIsALongButValidGenericToken
```

Decimal literal a list of digits. For example:

```
12345
0
255
42
```

Octal literal a list of *octal digits* ('0' to '7') followed by a 'B'. For example:

```
10B (= 8)
377B (= 255)
```

Hex literal a list of *hexadecimal digits* ('0' to '9' and 'A' to 'Z') followed by a 'H'. It must start with a digit. For example:

```
10H (= 16)
0FFH (= 255)
```

Real literal an list of digits, followed by a '.', followed optionally by a list of digits, optionally followed by an *exponent part* (an 'E', an optionally signed list of digits). For example:

```
3.14
12.3E-3 (= 0.0123)
```

String literal a list of characters enclosed in single or double quotes (' or "). The type of quote character selected cannot appear in the string, nor can the string extend over more than one line. A string of length one is called a *character literal*. A character literal can also be specified by its octal value followed by a 'C'. For example:

```
'Hello'
"How are you?"
101C (= 'A')
```

Identifiers are used to denote user-defined (and predefined) elements of a program, they are case-sensitive, i.e., Hello and HELLO are *different* identifiers.

The decimal, octal and hexadecimal literals denote whole numbers. Real literals denote real numbers. The exponent part denotes multiplication of the mantissa by the specified power of 10.

The compiler can only distinguish generic tokens by the use of separators or delimiters. Thus, 123 is a single, three-digit, decimal literal not three, one-digit, decimal literals.

Separators

The *separators* are:

White-spaces any list of blanks, tabs and line breaks.

Comments any list of characters enclosed in '(*' and '*)'. Comments can be nested and extend over line breaks.

Separators have no meaning to the compiler other than to separate tokens, except when a comment begins with '(*\$', '(*#' or '(*%'.

Syntax Descriptions

The *syntax* of Modula-2 describes how sequences of tokens can be assembled to form programs and modules.

Any syntax description simply specifies a list of *productions* which specify how *syntactical constructs* can be put together to form *sentences* of the syntax. A production tells you how to create a new syntactical construct from another.

Notice, however, that the syntax of Modula-2 does not tell you what the legal sentence *does* or *means*. That is purpose of the semantics of the language.

In this chapter, a *formal notation* is used to describe the syntax of TopSpeed Modula-2. English is used to describe the semantics. The formal notation is simply a way of expressing the syntax precisely.

The formal notation consists of productions involving syntactical constructs and tokens. Each production consists of a single syntactical construct being defined in terms of other constructs and tokens. For example:

TypeDef ::= ARRAY [*IndexList*] OF [*TypeDef*]

This production says that the syntactical element *TypeDef* is defined to be the token ARRAY, followed by an entity conforming to the rules of the syntactical construct *IndexList*, followed by the token OF, followed by an entity conforming to the rules of the syntactical construct *TypeDef*.

Each construct can have several productions associated with it, each expressing a possible expansion of the construct.

Syntax descriptions use some special characters called *meta-symbols*. The meta-symbols used in the syntax of TopSpeed Modula-2 described here, are:

<i>square brackets</i>	([and]) are used to enclose parts of a production that are optional.
<i>curly braces</i>	({ and }) are used enclose parts of production which may be repeated zero, one or more times.
<i>bar</i>	() is used to separate alternative definitions within a production.
<i>definition symbol</i>	(::=) is used to separate the syntactical construct being defined from its expansion.
<i>tokens</i>	alphabetic tokens are shown in typewriter format.
<i>single quotes</i>	are used to enclose non-alphabetic tokens. This is done to avoid confusion between the meta-symbols and the tokens of the language. For example:

IdList ::= *Identifier* { ‘,’ *Identifier* }

This production defines a construct called ***IdList*** as an ***Identifier*** (defined as a textual token on page) followed by one or more occurrences of a comma followed by an ***Identifier***. This implies that the following are three valid examples of ***IdList***:

HelloThere
a, b, c
p1, p2, p3, p4, p5

Declarations and Visibility

Every identifier in a Modula-2 program must be *predefined* or *declared*. Each declaration introduces a programmer-defined entity and establishes its properties. After an identifier has been declared in a program it can be used to name the declared entity:

Name ::= Identifier

Declarations, when they occur, come in lists of declarations:

DeclarationList ::= { Declaration }

All identifiers must be declared before they are used. The only exception to this is the case of *pointer types*. In this case, the type to which the pointer type is bound must be declared *before* the end of the current declaration list. Until a such an entity is declared, any operation requiring knowledge of the structure of the type is illegal.

All identifiers declared in a declaration list must be distinct; i.e. they must have different names.

Every declaration remains in effect throughout its *scope*. The scope of a declaration is effective throughout the program block in which the declaration is made.

Declaration lists may appear in nested procedures and modules, and it is legal to redeclare an identifier within such inner program blocks. The WITH statement is another way of making a nested scope for identifiers.

A particular occurrence of an identifier denotes the entity defined in the innermost enclosing scope that contains a declaration of the same identifier. If an identifier is redeclared in an inner scope, it *hides* any occurrences of the same identifier in any outer scope.

For example:

```

MODULE M;
  VAR
    i, j : CARDINAL;
  (* Two variables of the module M *)
  PROCEDURE P;
    VAR
      i, k : INTEGER;
    (* Two variables of the procedure P *)
    BEGIN
      i := 7; (* P's i *)
      j := 8; (* M's j *)
      k := 9; (* P's k *)
    END P;
  BEGIN
    i := 12; (* M's i *)
    j := 14; (* M's j *)
    (* there is no k visible at this point *)
  END M.
```

Alias Declarations

An *alias* declaration defines a new name for an existing entity. It defines nothing new, but simply gives an alternative name for something that already exists:

Declaration ::= CONST { *Identifier* '::' '=' *Name* ';' }

The name can denote any entity.

Predefined Identifiers

The following identifiers are considered to be declared in a outermost scope which embraces the entire program.

ABS	ADDRESS
ADR	BITSET
BOOLEAN	BYTE
CAP	CARDINAL
CHAR	CHR
DEC	DISPOSE
EXCL	FALSE
FarADDRESS	FarADR
FarNIL	FieldOfs
FIXREAL	FLOAT
HALT	HIGH
INC	INCL
INTEGER	LONGCARD
LONGINT	LONGREAL
LONGWORD	MAX
MIN	NearADDRESS
NearADR	NearNIL
NEW	NIL
NULLPROC	ODD
Ofs	ORD
PROC	REAL
Seg	SHORTADDR
SHORTCARD	SHORTINT
SIZE	TEMPREAL
TRUE	TRUNC
VAL	VSIZE
WORD	

Types

A type defines a set of values. Modula-2 contains a number of predefined types and constructs for building new types. New types are given names in type declarations:

```
Declaration ::= TYPE { Identifier '=' TypeDef ';' }
```

Some types are called *simple types*:

```
TypeDef ::= SimpleType
SimpleType ::= Name
```

If the name specified in a type declaration is just the name of another type, then the newly defined type is *identical* to the old type. For example:

```
TYPE JustACardinal = CARDINAL;
```

Numeric Types

There are three groups of predefined numeric types in TopSpeed Modula-2: CARDINAL types, INTEGER types and REAL types.

CARDINAL Types

The three predefined *cardinal* (positive whole number) types and their numeric ranges are:

```
CARDINAL    0 to 65535      (0 to 216 - 1)
SHORTCARD   0 to 255       (0 to 28 - 1)
LONGCARD    0 to 4294967295 (0 to 232 - 1)
```

INTEGER Types

The three predefined *integer* (signed whole number) types and their numeric ranges are:

```
INTEGER     -32768 to +32767 (-215 to +215 - 1)
SHORTINT    -128 to +127    (-27 to +27 - 1)
LONGINT     -2147483648 to +2147483647 (-231 to +231 - 1)
```

REAL Types

The four predefined *real* (floating point) types and their numeric ranges are:

```
REAL        ± 1.2E-38 to 3.4E+38  6 digits precision
LONGREAL    ± 2.3E-308 to 1.7E+308 15 digits precision
TEMPREAL    ± 3.36E-4931 to 1.19E+4932 19 digits precision
FIXREAL     ± 1.84e19 to 1.84e19-1 19 digits precision
```

Ordinal Types

The *ordinal* types in TopSpeed Modula-2 are collectively the integer, cardinal, *character* and *enumeration* types. The first two of these have been defined above. An ordinal type consists of whole number values. The enumeration and character types are numbered consecutively from zero. We also define the *short ordinal* types consisting of all ordinal types except LONGINT and LONGCARD.

CHAR (Character) Type

The predefined type, CHAR, contains 256 values. The first 128 represent the characters in the ASCII set, the remaining 128 are special graphic characters.

Enumeration Types

An enumeration type is simply a list of all the possible values for the type. The values, called *enumeration literals*, are simply identifiers:

```
SimpleType ::= '(' IdentifierList ')'
```

For example:

```
TYPE
  Day    = ( Monday, Tuesday,
            Wednesday, Thursday,
            Friday, Saturday,
            Sunday );
```

Modula-2 has one predefined enumeration type defining truth values:

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

Subrange Types

Given an ordinal type, it is possible to define a *subrange* type of that base type consisting of a range of values selected from the base type:

```
SimpleType ::= [ Name ] '[' Expr '..' Expr ']'
```

The two expressions must be constants and of the same type. They restrict the range of values allowed in the type by specifying an lower and upper bound. The lower bound must be less than or equal to the upper bound. If the *Name* is present then it specifies the base type for the subrange. If *Name* is unspecified then the base type is determined by the expressions. If the expressions are enumeration or ordinal literals, then the base type is the enumeration (or subrange) type to which both expressions belong. If the expressions are numeric literals then the base type is INTEGER if the first expression is negative, otherwise the base type is CARDINAL.

For example:

```

TYPE
  Year = [1900..2001];
  (* Base is CARDINAL *)
  Temp = [-40..40];
  (* Base is INTEGER *)
  Digits = ['0'..'9'];
  (* Base is CHAR *)
  IntYear = INTEGER [1900..2001]
  (* Base is INTEGER *)
  WeekDay = [Monday..Friday];
  (* Base is Day *)

```

All subrange types are themselves ordinal types.

Set Types

Given a short ordinal type, it is possible to define a *set* type consisting of unordered sets of values of that short ordinal type:

```

TypeDef ::= SET OF SimpleType

```

The minimum ordinal value of the set element type must be greater than or equal to 0.

A set can contain any subset of the values that make up the elements of the set's type. For example:

```

TYPE Chars = SET OF CHAR;

```

TopSpeed Modula-2 contains one predefined set type representing the bits of a single machine word:

```

TYPE BITSET = SET OF [0..15];

```

Array Types

An *array* type defines an collection of elements of identical types which can be individually referenced by a short ordinal *index*. Each member of the collection has an *array element* type:

```

TypeDef ::= ARRAY IndexList OF TypeDef
IndexList ::= SimpleType { ' ', SimpleType }

```

When more than one index type occurs in the index list, for example:

```

TYPE TwoDArray = ARRAY [1..10,1..5] OF INTEGER;

```

it is equivalent to an expanded definition, like this:

```

TYPE TwoDArray = ARRAY [1..10] OF
  ARRAY [1..5] OF INTEGER;

```

For example:

```

TYPE
  NameString = ARRAY [0..32] OF CHAR;
  IntArray = ARRAY BOOLEAN OF INTEGER;
  Week = ARRAY [Monday..Sunday] OF BOOLEAN;

```

Record Types

Record types provide a means of producing an aggregate of *fields*. Each field consists of an entity of some other type:

```

TypeDef ::= RECORD FieldDefList END
FieldDefList ::= FieldDef { ';' FieldDef }
FieldDef ::= [ IdentifierList ':' TypeDef ]

```

All the identifiers, known as *field names*, must be distinct within the record. However, they are local to the record definition and can duplicate the names of identifiers declared elsewhere.

A record value contains one value for each field defined. For example:

```

TYPE
  Person = RECORD
    First, Last : NameString;
    Age : SHORTCARD [0..125];
  END;

```

A *variant record* type allows alternative groups of fields to be present in a record value. Variant parts can be nested as required:

```

FieldDef ::= VariantPart ';'
VariantPart ::=
  CASE [ Identifier ] ':' Name OF
    { Variant } { '|' Variant }
  [ ELSE FieldDefList ]
  END
Variant ::= [ ChoiceList ':' FieldDefList ]
ChoiceList ::= Choice { ',' Choice }
Choice ::= Expr [ '..' Expr ]

```

The optional *tag identifier* that follows the word CASE is followed by the name of a short ordinal *tag type*. If the tag identifier is present it defines an actual field in the aggregate of the type stated.

Each of the variants and the optional ELSE part specify a possible field arrangement for the variant part. But, for any one occurrence of the variant record type, only one of the variants is present.

The *choice* expressions must be constant and their values must not overlap. The choice expression must also be of the same type as the tag type. A single choice with two expressions separated by *'..'* denotes all the values in the stated range.

The presence or absence of variant fields is not a physical property, the arrangement is purely logical. All variant fields are always accessible but they share storage. For example:

```

TYPE
  Shape =(Circle, Square,
           EqTriangle, Rectangle);
  Figure = RECORD
    x, y : CARDINAL;
    CASE ShapeType : Shape OF
      | Circle :
        Radius : REAL;
      | Square, EqTriangle :
        SideLength : CARDINAL;
    ELSE
      Width, Height : CARDINAL;
    END;
  END;
END;

```

Pointer Types

The *value* of a pointer type is the address of an object of a *designated* type:

TypeDef ::= POINTER [Expr] TO TypeDef

If the expression is omitted, an *absolute pointer* type is defined. By including a CARDINAL expression, a *based pointer* type is defined. Such values only contain the offset part of an address. The segment part of the address is found by evaluating the expression every time the designated object is accessed.

The only restriction in TopSpeed Modula-2 is that the expression cannot contain function calls. For example:

```

VAR
  Base: CARDINAL;
TYPE
  ListPtr = POINTER TO ListNode;
  ListNode = RECORD
    Value : CARDINAL;
    NextNode : ListPtr;
  END;
  ShortPtr = POINTER Base TO ListNode;

```

There are several predefined pointer types:

```

TYPE
  ADDRESS    = POINTER TO WORD;
  SHORTADDR  = POINTER S TO WORD;
  NearADDRESS = POINTER S TO WORD;
  FarADDRESS = POINTER TO WORD;

```

A FarADDRESS is always a 32-bit quantity, a NearADDRESS is always a 16-bit quantity, where S is the current data segment. The size of ADDRESS depends on the compilation memory model.

Virtual Pointers

TopSpeed Modula-2 now supports a special pointer type: the virtual pointer:

TypeDef ::= VIRTUAL POINTER *Identifier*

Identifier is a function taking one parameter, and returning a pointer type.

The type defined is a distinct type on which only the operations allowed are assignment, passing as a parameter and dereferencing.

The semantics for virtual pointers are that whenever they are dereferenced, the function **Identifier** is called, and its return value is dereferenced.

Array, record, pointer and class types are collectively called *compound types*; the rest, apart from set types, are called *simple types*.

Type Compatibility

Numerous situations require types to be compatible. There are three levels of compatibility, each of decreasing strength.

The strongest requires that the types are *identical*, i.e., they must denote the same type definition.

Next comes *compatible*. This includes a subrange type being compatible with their base type and other subrange types of the same base type.

Finally, *assignment compatibility* holds between the pairs CARDINAL and INTEGER, SHORTCARD and SHORTINT, LONGCARD and LONGINT.

There are three predefined types called BYTE, WORD and LONGWORD. They correspond to 1, 2 and 4 bytes of memory, respectively. They are assignment compatible with all types of equal size.

Objects and Values

Objects have a type and hold a *value* of that type. There are two types of objects: *variables* and *formal parameters*. Objects of array and record types contain several *component values*. Variables have to be declared:

```
Declaration ::= VAR { VarID { ',' VarID } ':' TypeDef ';' }
VarID ::= Identifier [ '[' Expr ':' Expr ']' ]
```

Each declaration declares all the identifiers in the list to be of the specified type. The initial value of variables is undefined. For example:

```
VAR I, J, K, L: CARDINAL;
    LongOne: LONGCARD;
    RealValue: REAL;
    PtrValue: POINTER TO INTEGER;
    CharStr: ARRAY [0..79] OF CHAR;
    Special : Location;
    People: ARRAY BOOLEAN OF POINTER TO Person;
```

A variable can be placed at a fixed physical address by specifying a **CARDINAL** segment and offset, for example:

```
VAR Screen [0B800H:0] : ARRAY [1..25] OF
    ARRAY [1..80] OF RECORD
        Chr : CHAR;
        Atr : SHORTCARD;
    END;
```

Constants

Constant literals denote the basic values:

```
Value ::= WholeNumber | RealNumber | String
```

Whole numbers are values for integer and cardinal types, real numbers are values for real types and strings are for character and character array types.

Constant record and array values are formed with *aggregates*:

```
Value ::= Name '(' Expr ',' Expr { ',' Expr } ')'
```

The expressions (there must be at least two) are constants or procedures and give the component values for the named type. For array types, one value must be given for each value in the index range. For record types, one value must be given for each field. Values must be specified for unnamed tag fields in variant records, in order to determine which variant the following values belong to.

For example:

```

TYPE
  RealProcedure = PROCEDURE ( REAL ) : REAL;

PROCEDURE Square( v : REAL ) : REAL;
BEGIN
  RETURN ( v * v );
END Square;

TYPE
  ProcRecord = RECORD
    Val1, Val2 : REAL;
    Proc : RealProcedure;
  END;

CONST
  ExampleProcRec = ProcRecord(1.0, 2.7, Square);

```

Named constants can be declared, denoting identifiers representing constant values:

Declaration ::= CONST { *Identifier* '=' *Expr* ';' }

The following are predefined constants compatible with the various pointer types:

NIL NearNIL FarNIL

NIL is equal to either NearNIL or FarNIL depending on the memory model. Neither literal nor named constants are objects.

Set Values

Set values are formed with a *set constructor*:

Value ::= [*Name*] '{' [*ChoiceList*] '}'

The expressions of the choices are values of the set type; they need not be constants. The name denotes the set type, if omitted, it implies BITSET. For example:

Chars{ 'A'..'Z', 'a'..'z', '_' }

Designators

Designators are used to denote objects. Component objects of compound objects are designated by supplying suffixes. Suffixes are also used to denote components of named aggregate constants. Using a designator as a value means the value of the designated object. This simplest form of a designator is just the name of an entity.

```
Value ::= Designator
Designator ::= Name
```

Indexing is used to designate components of array types:

```
Designator ::= Designator '[' Expr { ',' Expr } ']'
```

A list of index expressions is the same as a list a separate indexes: X[a,b,c] is identical to X[a][b][c]. The value of the expression must be assignment compatible with the index type.

Field selection is used to designate components of record types:

```
Designator ::= Designator '.' Identifier
Designator ::= Designator ActualList '.' Identifier
```

The identifier is any field of the of the record type. The resulting designator designates that field of the record type.

Dereferencing is used to designate the object pointed to by objects of pointer types:

```
Designator ::= Designator '^'
Designator ::= Designator ActualList '^'
```

Indexing, field selection and dereferencing may be mixed, and may include function calls, if they are followed by a '^' or '':

```
Persons[TRUE]^Last[0]
CurrentPerson()^.Last[0]
```

A *pointer constructor* combines CARDINAL segment and offset values to produce a physical address:

```
Designator ::= '[' Expr ':' Expr [ Name ] ']'
```

The name specifies the resulting absolute far pointer type, if omitted, it implies FarADDRESS.

Expressions

Expressions specify the computation of values. Within expressions, *operators* are used to combine *operands*.

Expressions, like values, have types unless all their operands are numeric literals. In this case, the determination of the type of the expression is deferred until the type is required to be known. Unless a context is introduced, the only distinction is between whole numbers and real numbers. Generally, when the operands of an operator or predefined function are all constant, the result is also constant and is calculated at compile-time, but note that:

ADR(“some string or other”)

is *not* a constant. You cannot use the address of string literals in constant expressions in this way.

The precedence of operators is defined in the syntax below:

```
Expr ::= SimpleExpr [ RelOp SimpleExpr ]
SimpleExpr ::= [ SignOp ] Term { AddOp Term }
Term ::= Factor { MulOp Factor }
```

Factor ::= ‘(*Expr* ’) | NOT **Factor** | *Value*

RelOp ::= ‘=’ | ‘#’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | IN | <>

SignOp ::= ‘+’ | ‘-’

AddOp ::= ‘+’ | ‘-’ | OR

MulOp ::= ‘*’ | ‘/’ | DIV | MOD | AND | ‘<<’ | ‘>>’

The operation indicated by the operator depends on both the operator and the operand types. All operators, except IN, require operands of compatible types. All relational operators and IN, produces a result of type BOOLEAN; the rest produce a result of the same type as their operand(s).

The operators ‘=’ and ‘#’ are defined for all types, and compare for equality and inequality, respectively. The operators ‘#’ and ‘<>’ are identical.

The operators ‘<’, ‘<=’, ‘>’ and ‘>=’ are defined for ordinal and real types and compare for relative ordering. ‘>=’ and ‘<=’ are defined for set types and test for subset and superset.

The sign operator ‘+’, is defined for all numeric types and does nothing. The sign operator, ‘-’ is defined for real and integer types and negates the operand.

The operators '+' and '-' are defined for numeric types and indicate addition and subtraction. They are also defined for set types, and indicate set union and set difference. '+' is also defined for any combination of character constants and string literals, and indicates concatenation to form a string literal. The OR operator takes BOOLEAN operands and computes the logical-or operation; the right operand is only evaluated if the left operand is FALSE.

The operators '*', DIV and MOD are defined for integer and cardinal types and compute the product, quotient (truncated towards zero) and remainder. '<<' and '>>' are defined for cardinal types, computing logical left and right shifts of the left operand by the right. '*' and '/' are defined for real types and compute the product and quotient. '*' and '/' are also defined for set types and compute set intersection and symmetric set difference. The AND operator is defined for BOOLEAN operands, computing the logical-and operation; the right operand is only evaluated if the left is TRUE.

The NOT operator takes a BOOLEAN operand and complements it.

The IN operator takes values of set types as a right operand and values of the set element type as the left operand. It tests the elements for membership of the set.

Note: Because sets are represented as bitmaps, the operators '+', '*', '/' and '-' corresponding to the bitwise logical operation OR, AND, XOR and AND NOT, respectively.

For example:

```
(j = 0) OR ((i MOD j) = (i - ((i DIV j) * j)))
(x * y) / Ratio
r.x + 1
s[1] IN (z * (Chars['A'..'F'] + Chars{'0'..'9'}))
'Line terminated with CR/LF' + CHR(13) + CHR(10)
```

The operators 'NOT', 'AND' and 'OR' can be used to perform bitwise operations on whole number expressions.

Statements

Programs achieve their effect by executing *statements*. Statements come in lists and are executed one at a time:

```
StmtList ::= [ Stmt ] { ';' Stmt }
```

Assignment Statement

The assignment statement is used to change the value of an object:

```
Stmt ::= Designator ':=' Expr
```

The expression is evaluated and its value replaces the old value of the designated object. The value must be assignment compatible with the designated object's type.

String literals are a special case. They can be assigned to any object whose type is ARRAY ... OF CHAR and is long enough to contain the string. If the object is longer than the string literal, a null character (0C) is placed at the end of the string. For example:

```
x : 0.010356;
i := j + 1;
z := z - Chars{'a'..'f'};
Persons[NOT Bad]^First := 'Kurt';
```

IF Statement

The IF statement is used to select a statement list conditionally based on the result of a BOOLEAN expression called the *condition*:

```
Stmt ::= IF Expr THEN StmtList
        { ELSIF Expr THEN StmtList } [ ELSE StmtList ]
        END
```

At most one of the statement lists is executed, namely the one following the expression that is TRUE. The ELSE part is treated as ELSIF TRUE THEN. For example:

```
IF (i > j) THEN
  m := i;
ELSIF (i < j) THEN
  m := j;
ELSE
  b := TRUE;
  m := i;
END;
```

CASE Statement

The CASE statement selects between alternative statement lists depending the value of an ordinal expression:

```

Stmt ::= CASE Expr OF [ ‘|’ ] Case { ‘|’ Case }
        [ ELSE StmtList ]
        END
Case ::= [ ChoiceList ‘:’ StmtList ]

```

The types of the choice expressions must be compatible with the case expression’s type, they must be constants and they must not overlap.

The statement list executed will be the one whose choice list includes the value of the case expression. If no choice list matches, then the statement list following ELSE will be executed, if present (if no ELSE is present no statement list is executed). For example:

```

CASE s[i] OF
| ‘.’ : i := 999
| ‘A’..‘Z’ : l := l + 1;
              u := u + 1;
| ‘a’..‘z’ : l := l + 1;
| ‘[’, ‘]’ : x := x + 1;
ELSE
  TheEnd := TRUE;
END;

```

WHILE Statement

The WHILE statement is used to execute a statement list zero or more times based on the value of a condition:

```

Stmt ::= WHILE Expr DO StmtList END

```

The expression is evaluated *before* each execution of the statement list. Execution continues until the expression is FALSE. For example:

```

WHILE (i > 0) AND ((i MOD 2) = 0) DO
  i := i DIV 2;
END;

```

REPEAT Statement

The REPEAT statement is used to execute a statement list one or more times based on the value of a condition:

```

Stmt ::= REPEAT StmtList UNTIL Expr

```

The expression is evaluated *after* the statement list has been executed. Execution continues until the expression is TRUE. For example:

```
REPEAT
  i := (i MOD n) + 1;
UNTIL (s[i] = ' ');
```

LOOP Statement

The LOOP statement executes a statement list repeatedly. Several exit points are possible from the loop.

```
Stmt ::= LOOP StmtList END
       EXIT
```

An EXIT statement is only legal inside a loop; it terminates the innermost LOOP statement. It has no effect on WHILE or REPEAT statements only the enclosing LOOP statement. For example:

```
LOOP
  IF (i = j) THEN
    EXIT
  END;
  WHILE (i < j) DO
    i := (i * j) MOD n;
    IF (i = 0) THEN
      i := j;
      EXIT; (* Exits WHILE as well as LOOP *)
    END;
  END;
  i := i DIV 2;
END;
```

FOR Statement

The FOR statement is used to execute a statement list a particular number of times with a *control variable* taking a series of values.

```
Stmt ::=
  FOR Identifier ':' Expr TO Expr [ BY Expr ] DO
    StmtList
  END
```

The first two expressions are evaluated to calculate the *start* and *stop* values for the control variable. They must be assignment compatible with the type of identifier which must be declared as an ordinal type. The expression after BY is called the *step* value and must be a constant whole number. If omitted the step value defaults to +1.

The control variable is initially assigned the start value and takes on successive values by adding the step value. The FOR statement terminates

when the control variable exceeds the stop value; if the start value is initially greater than the stop value, the statement is not executed at all.

The value of the control variable should not be changed within the statement list and its value is undefined when the statement terminates. For example:

```
FOR Ball := Black TO Yellow BY -2 DO
  LastBall := Ball;
END;
```

WITH Statement

The WITH statement is used to create a local scope where the field names of a record type can be used without an initial designator.

```
Stmt ::= WITH Designator DO StmtList END
```

For example:

```
WITH Persons[i] DO
  IF (Age < 4) THEN
    First := "baby";
  ELSIF (Age < 15) THEN
    First := "junior";
  END;
END;
```

GOTO Statement

The GOTO statement is used to unconditionally alter the flow of execution:

```
Stmt ::= GOTO Identifier
```

The target statement is indicated by the label identifier and must be located somewhere within the same *body*:

```
Stmt ::= Identifier ':' [ Stmt ]
```

Labels must be declared before they are used:

```
Declaration ::= LABEL IdentifierList ';' ;
```

Procedures

Procedures are used to group commonly performed operations into isolated blocks. *Proper procedures* are used like statements; *function procedures* compute values and are used in expressions.

Procedures must be declared like other entities, they are invoked by calls to obtain their effect, they can have *parameters* that are objects or values supplied at the call and they finish executing by *returning*.

Procedures can also be handled without being called, they can be valid values of a *procedure type* and can be manipulated as such.

```
ProcHead ::= PROCEDURE Identifier [ FormalList [ ':' Name ] ] ';'
FormalList ::= '(' [ FormalSection { ';' FormalSection } ] ')'
FormalSection ::= [ VAR ] IdentifierList ':' FormalType
FormalType ::= [ ARRAY OF ] Name
```

A procedure heading declares the identifier denoting the procedure. It possibly has a list of formal parameters, declared similarly to variables. The absence of the *:* **Name** in the procedure head indicates a proper procedure, its presence indicates a function and the name denotes the returned value's type.

Each formal section declares a list of formal parameters, states their *formal type* and indicates whether they are *variable* (VAR) parameters or *value* parameters. All the parameter identifiers must be distinct. For example:

```
PROCEDURE NewLine;
PROCEDURE RandomNo() : CARDINAL;
PROCEDURE PrintNumber(n: INTEGER;
                      Width : CARDINAL );
PROCEDURE Sum(VAR Result: LONGREAL;
              List : ARRAY OF Real );
```

Bodies

The procedure heading is combined with a *body* specifying the internal workings of the procedure. A FORWARD declaration can be used to delay specifying the body:

```
Declaration ::= ProcHead Body Identifier ';'
Declaration ::= ProcHead FORWARD ';'
Body ::= DclList [ BEGIN StmtList ] END
```

The identifier after the body repeats the procedure's name. A FORWARD declared procedure must be completed later in the same declaration list with a full declaration repeating the procedure heading and supplying a body.

The declaration list of a procedure declares entities that are local to the procedure; they must have names distinct from the formal parameters.

Formal parameters of type ARRAY OF type, *open array parameters*, are considered to be local arrays, indexed with a CARDINAL starting from 0. The upper index can be obtained with HIGH function.

Local variables come into existence when procedures are called and vanish when they return. Procedures can call themselves directly or indirectly, causing several invocations of the local variables to come into existence simultaneously. Designating a local variable refers to the instance for the most recent invocation of the procedure.

A formal value parameter is considered a local variable whose value is initialized when the procedure is called. Formal VAR parameters denote actual objects that are identified by the caller. Open arrays can be used as formal parameters to other procedures having formal open array parameters, otherwise they can only be manipulated element-wise.

The statement list of a body specifies the actions to be performed by the procedure.

Executing a RETURN statement is the only legal way of exiting a function; proper procedures can return by simply reaching the end of the statement list.

Stmt ::= RETURN [Expr]

The expression *must* be present for functions exclusively and the expression's type must be assignment compatible with the function's return type.

A procedure can also be defined as a *binary procedure*:

Declaration ::= ProcHead '=' Expr ';' ;

In such declarations, the expression is a constant expression, possibly an aggregate (a record or array). This expression is a series of bytes that are machine code instructions. For example:

```
PROCEDURE Max( x, y : LONGREAL ) : LONGREAL;
BEGIN
  IF (x > y) THEN
    RETURN x;
  ELSE
    RETURN y;
  END;
END Max;
```

```

PROCEDURE HypSquare( a, b : LONGREAL ) : LONGREAL; FORWARD;

PROCEDURE Accumulate( VAR x : LONGREAL; y : LONGREAL );
  VAR
    z : LONGREAL;
  BEGIN
    a := HypSquare(y,(10.0 - y));
    x := x + Max((z * z),-999.99);
  END;

PROCEDURE HypSquare( a, b : LONGREAL ) : LONGREAL;
  BEGIN
    RETURN (a * a) + (b * b);
  END;

TYPE
  A1 = ARRAY [0..0] OF SHORTCARD;
(*# save,
  call( reg_param=>(dx,ax),
    reg_saved=>(dx,ax,bx,cx,si,di,es,ds,st1,st2))
*)

INLINE PROCEDURE Out (p:CARDINAL;v:SHORTCARD)=A1(0EEH);

```

External Procedures

A procedure may be declared as external to a module.

```

ProcHead ::= PROCEDURE Identifier [ FormalList [ ‘:’ Name ] ] ‘;’ IN
Module

```

The procedure will be declared in a definition module, but not implemented in the corresponding implementation module. Typically the procedure will be implemented in another assembly language module. External declarations must precede all other procedure implementations. For example:

```

IMPLEMENTATION MODULES MYMOD;

PROCEDURE InAssembler(C:CARDINAL); IN AsmMod

```

Calling Procedures

Proper procedures are invoked in call statements:

```

Stmt ::= Designator [ ActualList ]
ActualList ::= ‘(’ [ Expr { ‘,’ Expr } ] ‘)’

```

The *actual parameter* list must supply one value or designator for each corresponding formal parameter. For a VAR parameter the expression must be a designator for an object of the same type as formal parameter.

A value parameter can be any expression that is assignment compatible with the type of the formal parameter. *Strings* are valid parameters for any formal parameter of type ARRAY OF CHAR.

A formal type of ARRAY OF type is considered identical to any array with that element type. The index range of the actual parameter is mapped onto a CARDINAL starting from 0. An expression of the element type is also valid; it is considered to be an array of one element.

The formal types BYTE, WORD and LONGWORD are compatible with any type of identical size. The formal types ARRAY OF BYTE, ARRAY OF WORD and ARRAY OF LONGWORD are compatible with anything, allowing the procedure to treat the actual parameter as unstructured storage. For example:

```
NewLine;
PrintMessage("Don't Panic!");
Accumulate(x,(7.0 * y));
```

Functions are invoked in expressions:

Value ::= Designator ActualList

Procedure Types

A procedure type denotes a family of procedures with identical calling characteristics:

```
TypeDef ::=
  PROCEDURE [ '(' [ FormalTypeList ] ')' ]
    [ ':' Name ] ]

FormalTypeList ::=
  [ VAR ] FormalType { ',' [ VAR ] FormalType }
```

The procedures belonging to the type are those with corresponding procedure headings. Two procedure headings correspond if they have the same number of formal parameters and the types of each formal parameter, taken in turn, are compatible. If the procedure type specifies a function procedure, then the return types must also be compatible. Each parameter must be of identical type for VAR parameters or assignment compatible otherwise. Predefined procedures and procedures nested within other procedures are excluded. For example:

```

TYPE
  PutProc = PROCEDURE ( ARRAY OF CHAR );

VAR
  g : ARRAY BOOLEAN OF PROCEDURE ( ) : CHAR;
  p : PutProc;

PROCEDURE MyProc( ARRAY OF CHAR );

BEGIN
  p := MyProc;
  (*
    this is valid because MyProc has a
    the corresponding number and type of
    parameters
  *)

```

There is one predefined procedure type that is defined for all *far procedures*:

```
TYPE PROC = PROCEDURE;
```

There is one predefined procedure value, `NULLPROC` that is compatible with all far procedure types. It causes a run-time error if called. Procedure values are denoted by designators without parameter lists.

Predefined Procedures

Modula-2 contains a number of predefined procedures. Some are generic in the sense that they can take several different parameter types and take one or two parameters.

Predefined Function Procedures

<code>ABS(X)</code>	Absolute value of numeric operands.
<code>ADR(X)</code>	The physical address of object X.
<code>CAP(C)</code>	Character C converted to upper case.
<code>CHR(X)</code>	The CHAR with ordinal value X.
<code>FarADR(X)</code>	Like <code>ADR(X)</code> but always a far address.
<code>FieldOfs(R.F)</code>	The relative offset of field F in record R.
<code>FLOAT(C)</code>	The REAL value of CARDINAL C.
<code>HIGH(A)</code>	The index upper bound for open array A.
<code>MAX(T)</code>	The maximum value for ordinal/real type T.

MIN(T)	The minimum value for ordinal/real type T.
NearADR	Like ADR(X) but always a near address.
ODD(X)	TRUE if the ordinal X is an odd number.
Ofs(X)	The offset part of the address of X.
ORD(X)	The CARDINAL value of X.
Seg(X)	The segment part of the address of the object X.
SIZE(T)	The size in bytes of the object or type T.
TRUNC(R)	The truncated CARDINAL value of REAL R.
VAL(T,X)	The value X converted to type T.
VSIZE(R.F)	Size of the record type R as if it contained only the fields up to, and including, field F.

VAL can convert values between any two numeric or ordinal types. Also, any type name can be used as a *type transfer* function, taking one value parameter of any type. The value is interpreted as a value of the named type. The actual bit pattern of the value remains unchanged unless the value type and the transfer type are numeric or ordinal, in which case a proper VAL conversion is performed.

A type transfer function is considered well-behaved if the sizes of the transfer type and the value type are the same, or if a proper VAL conversion is performed. If the value type size is more than the transfer type size, then the remaining last bytes of the value are ignored. If it is shorter, the last bytes of the resulting value are undefined. For example:

```
i := CARDINAL(BITSET(i) * BITSET(j))
(* bitwise AND *)
l := ADDRESS(p);
(* Not the address of p!!!!!!);
```

Predefined Proper Procedures

DEC(X) Decrement ordinal object X by one.

DEC(X,N) Decrement ordinal object X by N.

DISPOSE(X) When the compiler encounters the DISPOSE procedure it is replaced by a call to DEALLOCATE(X,SIZE(X^)). When used with pointers to objects, the SIZE of the object is determined dynamically, at run-time.

EXCL(S,E) Exclude element E from set object S.

HALT Terminate program execution normally.

INC(X) Increment ordinal object X by one.

INC(X,N) Increment ordinal object X by N.

INCL(S,E) Include element E in set object S.

NEW(X) This is replaced by a call to ALLOCATE(X,SIZE(X^)).

Modules

Modules encapsulate related declarations. An executing program consists of a *main module* and a number of server modules. The *server modules* consist of a *definition part* which is visible to clients, and an *implementation part* that hides the internal workings from the client.

Modules provide a mechanism for implementing features not supported explicitly in the language. Such features include: input and output, string handling, storage management, concurrency, operating system access, etc. (see Chapter 5).

Modules are the basic unit of compilation:

Compilation ::= *DefModule* ‘.’ [IMPLEMENTATION] *Module* ‘.’

Module ::= MODULE *Identifier* [*Priority*] ‘;’
 { *Import* } [*Export*] *Body Identifier*

DefModule ::= DEFINITION MODULE *Identifier* ‘;’
 { *Import* } *DclList* END *Identifier*

Priority ::= ‘[‘ *Expr* ‘]’

Each identifier names the module defined. Modules optionally have a constant CARDINAL priority used with the SYSTEM module (see Chapter 5). A compilation module without IMPLEMENTATION is a main module.

Server Module

The definition part of a module declares the entities available to clients. Only CONST, VAR and TYPE declarations are legal, in addition to the following which are legal only in definition parts:

Declaration ::= *ProcHead*

Declaration ::= TYPE [*Identifier* [‘=’ *TypeDef*] ‘;’]

The first form declares the existence of a procedure in the implementation part. The second form, when the type definition is omitted, declares the existence of an opaque pointer type with a unknown designated type. The operation allowed on objects of opaque type is assignment and tests for equality. Full definitions for the objects of the opaque type must be given in the implementation part of the module.

Binary inline procedures may be defined in both definition and implementation parts. Inline procedures may also be declared in both definition and implementation parts.

Declaration ::= INLINE ProcHead Body Identifier

Declaration ::= INLINE ProcHead ‘=’ Expr

The first form declares an inline procedure where the compiled body is placed in the resulting object code every time it occurs, i.e., it is not called.

The second form declares a *binary* inline procedure, as described in “Procedure Types”, above.

Objects declared in compilation modules are called *global* and exist throughout the execution of a program. For example:

```

DEFINITION MODULE Str;
(* Some string handling *)
CONST
  MaxWidth = 20;
TYPE
  Buffer = ARRAY [1..MaxWidth] OF CHAR;
VAR
  Width : [1..MaxWidth];
PROCEDURE Put( C : CARDINAL ) : Buffer;
PROCEDURE SetFill( C : CHAR );
END Str.

```

The implementation part contains declarations private to the module. These declarations are not accessible to client modules. The optional list of statements, the *initialization code*, is executed before any statements of clients are executed. If this is impossible to satisfy because of the circularity of dependencies, the initialization order is undefined within the circle.

The declarations in the implementation part form a single scope together with those of the definition part. Consequently every name from the definition part is visible and additional names in this scope must be distinct from them. For example:

```

IMPLEMENTATION MODULE Str;
VAR
  FillChar : CHAR;
PROCEDURE Put( C : CARDINAL ) : Buffer;
VAR
  P : [0..MaxWidth];
  S : Buffer;
BEGIN
  P := Width;
  REPEAT
    S[P] := CHR(ORD('0') + (C MOD 10));
    C := C DIV 10;
    DEC(P);
  UNTIL (C = 0) OR (P = 0);
  WHILE (P > 0) DO
    S[P] := FillChar;
  END
  RETURN S;
END Put;
PROCEDURE SetFill( C : CHAR );
BEGIN
  FillChar := C;
END;
BEGIN
  FillChar := ' ';
END Str.

```

Importing

Clients gain access to a server module by *importing* from it:

```
Import ::= [ FROM Identifier ] IMPORT IdentifierList ';' ;
```

The FROM form takes one module and a list of identifiers naming entities in it. Those names are considered declarations of those identifiers and thus achieve direct visibility of the named entities. Importing an enumeration type also imports the enumeration literals.

The FROM-less form imports each of the modules named. *Qualified names* are used to denote the entities from those modules:

```
Name ::= Name '.' Identifier
```

The name denotes a module, the identifier denotes an entity defined in the modules definition part. The same qualified notation can be used for the corresponding definition part in the module's implementation part. For example:

```

FROM Str IMPORT Put, Width;
IMPORT Str;
...
VAR
  S : Str.Buffer;
...
Str.SetFill('.');
Width := Str.MaxWidth DIV 2;
S := Put(i + 7);

```

Local Module

In addition to their use as compilation units, *local modules* can be declared:

Declaration ::= Module ‘;’

Local modules obey special scope rules. Any required entity (except the predefined ones) from outside the local module must be imported, and must be visible outside the local module. Thus the FROM-less import can name any entity (not only server modules). Using the FROM form requires the named server module to be visible (and, thus, already imported) immediately outside the local module.

Exportation is valid only in local modules:

Export ::= EXPORT [QUALIFIED] IdentifierList ‘;’

The identifier list contains entities declared in the local scope that are required in the outer scope.

Qualification can be used to access any entity.

The QUALIFIED keyword has no effect.

The optional statement list of the local module is executed before the statement list of the enclosing body.

CHAPTER 3

OBJECT-ORIENTED EXTENSIONS

This chapter is split into two sections. The first section is intended to give the reader some insight into object oriented programming. The second section is a reference guide to the TopSpeed Modula-2 extensions that implement the object oriented programming style.

What is Object Oriented Programming?

Programming in an object oriented manner, using TopSpeed Modula-2's object oriented extensions, requires you to forget about the traditional top down approach to program design. Instead you will identify those entities (objects) within the problem, that have some physical or conceptual boundary that divides them from the rest of the problem. At first you may find it difficult to break a programming problem into objects. However, one ad-hoc method that *may* help you is to write down what the program should do (or read your statement of requirements) and pick out all the important nouns. These nouns will become your objects.

Object oriented programming gives the programmer a triumvirate of very powerful programming aids:

- Encapsulation.
- Inheritance.
- Polymorphism.

These terms can best be defined in relation to traditional programming methods and structures.

Encapsulation

When designing a traditional monolithic Modula-2 program the programmer will usually solve it in a top down manner. An overall algorithm will be specified which will be successively refined into smaller steps. This design methodology typically yields the structure of a main program, consisting of a number of procedure or function calls which will themselves call further functions or procedures.

In such a tree-like structure, data often takes secondary importance. It is transformed as it is passed along the hierarchy. The ramification of data being passed up and down this hierarchy is that if a change is made to the format of any data structure, all the functions and procedures that operate on that data will also have to be modified to reflect the new data type. Thus one small change has a knock on effect throughout the program, involving changes to numerous widely scattered routines.

This knock on effect can be reduced by using the **module**. This allows the programmer to encapsulate data in separate compilation units. Encapsulating data (sometimes called information or data-hiding), means hiding or surrounding a data structure with a limited set of routines that can access it. Used properly, the use of modules restricts access to the data to a set of interface procedures (and functions).

Any changes to the data format can now be limited to that **module** without any changes to programs that import that **module**, so long as the headings of the interface procedures remain unchanged.

Object oriented programming allows you to encapsulate data by incorporating both the data and the very functions and procedures (called *methods* in object oriented terminology) that operate on that data into a data structure (the object). Access to data contained in the object is then limited to those methods.

So, an object consists of some data and a set of associated actions that operate on that data. To get an object to perform one of its actions/methods, you send a *message* to the object. For example, an object that represents an abstract data type would have data (the elements of the stack) and methods to perform actions applicable to that data, such as push and pop.

All objects must belong to a *class*. A class defines the implementation of a particular kind of object and can be thought of as a blueprint for making objects of a certain type. A class declaration in TopSpeed Modula-2 is similar to a record declaration, in so far as it is a structured data type consisting of a number of fields of differing types. The most visible difference you will notice is that the fields of a class, not only contain data variables (called *instance variables*), but also the methods that manipulate those data variables (in the form of function and procedure headings). Variables of a class type are referred to as *class instances* or *class members*, or, as we have been doing so far, as *objects*.

If you need to change the implementation of an object, the changes will not propagate to other parts of your program, as the data and the functions that operate on the data are all defined in the same place, namely the class declaration.

Inheritance

Sometimes when writing a module to implement an abstract data type, you will realize that you (or a colleague) have already done something similar, either for the same program, or for some previous program. In such a case you would probably copy the whole module, rename it, and make what changes where necessary to mould this old module into some thing new. In effect you would have re-used part of the old module. However, attempts at re-using code are full of pitfalls, as it requires a thorough knowledge of the implementation and usually means getting deep into the code, quite a problem if you did not write the code yourself.

The object oriented idea of *inheritance* is also a method of reusing or sharing code, but in a structured, controlled way, by defining a class and then using it to build a new class in terms of the original class. The new class is called the

derived class of the original class, and the original class is called the *base class*.

Instances of the derived class inherit all the methods and instance variables of the base class. In addition, the derived class can define additional instance variables and methods, as well as redefining old methods by redeclaring them in the context of the new class.

If some tried and tested module is altered to reflect some new data type, debugging the code can be a nightmare. Errors creep into previously functioning code. It is all too easy to forget what has changed, and what has remained unaltered (even if explanatory comments are used).

In contrast, by using classes and the inheritance mechanism, the ways in which the new class differs from the old are made explicit in the class declaration. They are listed with the additional instance variables and methods. Furthermore, errors cannot creep into the previously tried and tested code of the base class; as a consequence, debugging is isolated to any new or over-ridden methods.

By using inheritance a hierarchy of derived classes can be built. This hierarchy typically has a general class at the top, with the classes becoming more specific as they cascade towards the bottom; each derived class inheriting access to all its base classes' code and data.

Consider the following scenario: you have defined a class called Person, with instance variables of name, age and address, and with methods to access that data.

Next you find you need to define another class called Employee. This new class Employee needs all the data associated with Person, plus extra data such as salary and employee number.

With inheritance you do not have to create the new class from scratch. When defining the new class, Employee, specify that it inherits from Person, i.e. this new class is to have all the instance variables and methods of Person. To complete the definition of Employee, you only need to declare those instance variables such as salary, employee etc. and the associated methods, that are not included in Person. If one or more of the inherited methods is unsuitable for the new class, you can over-ride the action of that method by redefining the method in the new class.

Polymorphism

Polymorphism is the ability to issue the same command to different objects. The object then decides the appropriate action for that command. In object oriented terminology it is said that a *message* is sent to the object, which then invokes a

method of the same name (if one is present). Procedural languages generally have no notion of polymorphism.

Consider for example that you had written two modules to implement the abstract data types Person and Employee. If both these modules were used in the same program, you would have to make sure that procedures declared in the interface parts of these modules did not have the same names, otherwise name clashes would occur. Method names, however, are only in scope within their own objects, therefore such a name conflict never occurs. For example, you might have many objects of different class types, all having a method called print.

Class Declarations

The class in TopSpeed Modula-2 is implemented as an extension of the record type. A class declaration consists of two parts:

- An interface part.
- An implementation part.

Class Interface Declarations

A class interface can be declared anywhere in a Modula-2 program where it would be valid to make a type declaration.

A class interface declaration consists of the following:

- A field list and class-alias sequence. This declares the instance variables (the data that is associated with the class) together with any aliases.
- A method declaration sequence. This consists of a series of procedure headings that specify the messages that objects of that class will respond to.

Implementation of the methods are declared later in the implementation part. The syntax for a class interface declaration is as follows:

Syntax

```

ClassDecl ::=
  CLASS ClassIdentifier [ '(' Name { ',' Name } ')' ] ';'
    [ ClassFieldDefList ]
    [ MethodDeclList ]
  END ClassIdentifier ';'
ClassFieldDefList ::=
  ClassFieldDef { ';' ClassFieldDef }
ClassFieldDef ::=
  IdentifierList ':' TypeDef |
  Identifier '=' Name
MethodDeclList ::=
  MethodDecl { ';' MethodDecl }
MethodDecl ::=
  [ VIRTUAL ] PROCEDURE MethodIdentifier
    [ FormalList [ ':' Name ] ]

```

For example:

```

TYPE strType = ARRAY [0..23] OF CHAR;
CLASS Person;
  name   : strType;
  address: strType;
  age    : INTEGER;

  PROCEDURE assignName(
    theName: strType);
  PROCEDURE assignAddress(
    theAddress: strType);
  PROCEDURE assignAge(theAge: INTEGER);
  PROCEDURE getName(): strType
  PROCEDURE getAddress(): strType
  PROCEDURE getAge(): INTEGER
  PROCEDURE print;
END Person;

```

It is important to note that everything declared within a class declaration shares the same scope. A consequence of this is that a method's formal parameter list cannot duplicate identifiers already declared in the class's field list, otherwise there would be a conflict of identifiers.

For simplicity, some parts of the syntax for a class declaration have not been explained, but will be fully explored in latter sections. For an explanation of the keyword `VIRTUAL` see the section on Virtual and Static Methods in this chapter.

Class Implementation Declarations

The implementation part of a class declaration is concerned with defining the bodies of the methods:

- Classes that have been declared in the definition or implementation part of a module have their methods implemented in the implementation part of the same module.
- Classes that have been declared in a main module must have their methods implemented in that same module.

The syntax for the implementation part of a class declaration is as follows:

Syntax

```

ClassDef ::=
  CLASS IMPLEMENTATION ClassIdentifier
    [ MethodDefList ]
  BEGIN
    StmntList
  END ClassIdentifier
MethodDefList ::=
  MethodDef { ';' MethodDef }

```

```

MethodDef ::=
  [VIRTUAL ] PROCEDURE MethodIdentifier
    [ FormalList [ ':' Name ] ] ';'
    Body MethodIdentifier ';'

```

An example of the implementation part of a class declaration is as follows:

```

CLASS IMPLEMENTATION Person;

PROCEDURE assignName(theName:strType);
BEGIN
  name := theName
END;

...

(* implementation of other methods *)
PROCEDURE print;
BEGIN
  WrStr('Name    = ');
  WrStr(name);
  WrLn;
  WrStr('Age      = ');
  WrInt(age);
  WrLn;
  WrStr('Address = ');
  WrStr(address);
  WrLn;
END print;
(* object initialization block *)
BEGIN
  name := ' ';
  age := 0;
  address := ' ';
END Person;

```

In the body of a method, there is no need for an explicit reference to the particular object that the method is being invoked upon. This is because each method has an implicit value parameter called SELF, which is a reference to that particular class instance (object), and acts to bring that object's instance variables into scope. A method acts as if its entire statement part was surrounded by a WITH SELF DO statement:

```

PROCEDURE print;
BEGIN
  WITH SELF DO
    WrStr('Name    = ');
    WrStr(name);
    WrLn;
    WrStr('Age      = ');
    WrInt(age,1);
    WrLn;
    WrStr('Address = ');
    WrStr(address);
    WrLn;
  END;
END print;

```

Note: The scope of SELF and its field names is identical to that of the method parameters.

The implementation of a class terminates with a class initialization block, which can be used to initialize the data fields of a class instance. The class initialization block is activated whenever a class instance (object) is created, either in a VAR declaration, or, if the object is a dynamic variable, by a call of NEW. It is quite valid to have an empty class initialization block consisting of simply a BEGIN and an END.

Whenever an object of a derived class type is created, any initialization code inherited from its base classes (see the section on inheritance in this chapter) will be executed automatically, before the derived class's initialization code.

For example, given an inheritance hierarchy of three classes; class B inheriting from class A and class C inheriting from class B. When an object of class C is created (e.g., in a VAR declaration) Class A's initialization code will be executed first, followed by class B's and then finally, class C's initialization code will be executed.

This implies that, when a derived object is created, if any of its base classes' initialization code blocks call a virtual method (see the section on Virtual and Static Methods in this chapter), the method called will be the one defined by that base class, even if it has been over-ridden in the derived class.

Declaring Class Variables

Objects are variables of some class type, and are declared in exactly the same manner as other variables, in a var declaration. For example

```
VAR
  bill, fred : Person;
  x : INTEGER;
```

You can access both the data fields and the method fields of a class instance in the same way as you would access the fields of a record. However, in keeping with the spirit of object oriented programming, and data encapsulation in general, we recommend that you only ever access directly an object's methods, leaving access to the data fields to the methods themselves. If you need to access an object's data you should have written a method to do so!

```
x := fred.age; (* not recommended *)
x := fred.getAge; (* recommended *)
```

Invoking Methods

A method is invoked in a similar way as an ordinary Modula-2 function or procedure. The only difference is that the method name must be preceded by the object variable identifier and a period (“.”), in a manner similar to accessing the field of a record.

Syntax

Invoke ::= Designator ‘.’ MethodIdentifier [ActualList]

For example:

```
bill.getname;
```

An alternative way of calling an object’s method is to embed the call within a with statement, a construct you will be familiar with from the record type:

```
WITH bill DO  
  getname;  
END;
```

Inheritance

Most of the power of object oriented programming comes from the concept of inheritance, which is the ability to define a class as a customisation of another class type. The derived class inherits all the instance variables, methods and initialization code of its base class.

Syntax

```
ClassDecl ::=
  CLASS ClassIdentifier ['( ' Name { ' , '
    Name } ' ) ' ] ' ; '
    [ ClassFieldDefList ]
    [ MethodDeclList ]
  END ClassIdentifier ' ; ';
```

For example:

```
CLASS Employee (Person);
  employeeNo: INTEGER;
  department: strType;
  jobTitle : strType;
  salary : REAL;
  PROCEDURE getEmployeeNo():INTEGER;
  PROCEDURE getDepartment():strType;
  PROCEDURE getJobTitle():strType;
  PROCEDURE getSalary():REAL;
  PROCEDURE assignEmployeeNo(
    VAR eNo: INTEGER);
  PROCEDURE updateDepartment(
    VAR dept: strType);
  PROCEDURE updateJobTitle(
    VAR jTitle: strType);
  PROCEDURE updateSalary(
    VAR sal: REAL);
END Employee;
```

The new class Employee inherits all the instance variables and methods of its base class. In addition it declares four new data fields and eight new methods to manipulate that data.

Given the following var declarations:

```
VAR
  employeeObj:Employee;
  name:strType;
  dept:strType;
```

the following assignment statements could be made:

```
dept := employeeObj.getDept;
employeeObj.print;
```

The first statement invokes a method (getDept) declared in the new class Employee; whilst the second statement invokes the inherited method print on the class instance employeeObj.

Over-riding Methods

In the above example, the method print inherited by the Employee class will list to the screen, the values of only three of an Employee object's instance variables, namely: name, age and address. You may feel that this is inadequate, and that the print method should also list the values of employeeNo, department, jobTitle and salary. If this is the case, it will be necessary to over-ride the inherited method print.

To over-ride an inherited method, you need to re-declare the method in the class interface declaration.

```

CLASS Employee (Person);
  employeeNo: INTEGER;
  department: strType;
  jobTitle  : strType;
  salary    : REAL;
  PROCEDURE getEmployeeNo():strType;
  PROCEDURE getDepartment():strType;
  PROCEDURE getJobTitle():strType;
  PROCEDURE getSalary():REAL;
  PROCEDURE assignEmployeeNo(
    VAR eNo: INTEGER);
  PROCEDURE updateDepartment(
    VAR dept: strType);
  PROCEDURE updateJobTitle(
    VAR jTitle: strType;
  PROCEDURE updateSalary(
    VAR sal: REAL);
  PROCEDURE print;
    (* over-ride an inherited method *)
END Employee;
CLASS IMPLEMENTATION Employee;
(* implementation of Employee methods *)
....
PROCEDURE print;
BEGIN
  WrStr('Name = ');
  WrStr(name);
  WrLn;
  WrStr('Age = ');
  WrInt(age,1);
  WrLn;
  WrStr('Address = ');
  WrStr(address);
  WrLn;
  WrStr('Employee No. = ');
  WrInt(employeeNo,1);
  WrLn;
  WrStr('Department = ');
  WrStr(department);
  WrLn;
  WrStr('Job Title =');
  WrStr(jobTitle);
  WrLn;
  WrStr('Salary = ');
  WrReal(salary,10,10);
  writeLn;
END;
```

Invoking a Base Class Method

Within a method definition, a base class's method is invoked by preceding the method name with the base class's identifier and a period ("."). In the last example the print method of the class Person was over-ridden in the derived class Employee. When writing the method we not only included write statements for the instance variables that were declared for the Employee class, but also included write statements for the data variables that were inherited from the Person class.

A much better way of writing the method would have been to call the base class's method for printing within the new method.

```
PROCEDURE print;

BEGIN
  Person.print;
  WrStr('Employee No. = ');
  WrInt(employeeNo,1);
  WrLn;
  WrStr('Department = ');
  WrStr(department);
  WrLn;
  WrStr('Job Title =');
  WrStr(jobTitle);
  WrLn;
  WrStr('Salary = ');
  WrReal(salary,10,10);
  WrStr;
END print;
```

A base class's method can also be invoked directly by an object, in the body of a program block, by inserting the base class's identifier and a period between the object identifier and the method name.

For example, you may decide in your program that at a certain time you only wish to print the name age and address of an instance of Employee. This would be achieved as follows:

```
VAR
  emp: Employee;
BEGIN
  ...

  emp.Person.print;
  ...
END;
```

Multiple Inheritance

Multiple inheritance is the facility to declare a new class that inherits the features of two or more classes. For example, let us declare a class called BankAccount.

```

CLASS BankAccount;
  AccountNo: INTEGER;
  balance   : REAL;
  credit    : REAL;
  debit     : REAL;
  overdraft: REAL;
  PROCEDURE creditAccount(amount:REAL);
  PROCEDURE debitAccount(amount: REAL);
  PROCEDURE balanceOfAccount():REAL;
  PROCEDURE print;
END BankAccount;

```

The above class contains enough functionality to manipulate money going into a numbered bank account. However, in the real world, bank accounts are held by people, who are often referred to as account holders. What we really need is a class called `AccountHolder` that captures the functionality of a bank account and the functionality of a person.

```

CLASS AccountHolder ( Person ,
                      BankAccount );
END;

CLASS IMPLEMENTATION AccountHolder;
BEGIN
END AccountHolder;

```

This new class `AccountHolder` has inherited all of the instance variables and methods (bar one) of both the `Person` and `BankAccount` classes. However, inevitably with multiple inheritance with a derived class, inheriting from two or more base classes, name clashes will occur. When such conflicts occur, all clashing names are made invisible. In the above example, the class `AccountHolder` has no `print` method, because its base classes' `print` methods have been made invisible to it.

However the base classes' `print` methods can be used. A new method can be defined that calls them in a qualified form:

```

CLASS AccountHolder ( Person,
                      BankAccount );

PROCEDURE ahPrint;

END AccountHolder;

CLASS IMPLEMENTATION AccountHolder;

PROCEDURE ahPrint;
BEGIN
  Person.print;
  Account.print;
END ahPrint;

BEGIN
END AccountHolder;

```

Note: The method `ahPrint` could also have been named `print`.

Aliasing

Another way of resolving name clashes in the last example would be to make use of aliasing.

Aliasing within a class declaration is slightly different from that described in Chapter 5:

- The identifier being aliased must be qualified by a class identifier.

Syntax

Identifier '=' *QualName*

QualName ::= *ClassIdentifier* '.' *Identifier*

Aliases can only be declared within a class field list. For example:

```
CLASS AccountHolder ( Person,
                      BankAccount );
  pPrint = Person.print; (* alias *)
  aPrint = Account.print;(* alias *)
  Procedure ahPrint;
END AccountHolder;

CLASS IMPLEMENTATION AccountHolder;
PROCEDURE ahPrint;
BEGIN
  pprint;
  aprint;
END ahPrint;

BEGIN
END AccountHolder;
```

Aliasing is not restricted to resolving name clashes. It can also be used to make inherited identifiers more meaningful in the context of a derived class. In the last example we have only aliased methods, but in our final example you will see that you can also alias instance variables. For example:

```
CLASS InterestAccount ( BankAccount );
  InterestAccountNo = AccountNo;
  PROCEDURE calculateInterest(
    bal:REAL):REAL;
  PROCEDURE creditAccount(amount:REAL);
  PROCEDURE debitAccount(amount: REAL);
  PROCEDURE balanceOfAccount():REAL;
  PROCEDURE print;
END InterestAccount;
```

Compatibility Rules

There are particular compatibility rules that apply to instances of a class.

These rules can best be explained with the following example:

```

CLASS Animal;
...
END Animal;

CLASS Dog (Animal);
...
END Dog;

CLASS Alsatian (Dog);
...
END Alsatian;

TYPE
  AlsatianPtr = POINTER TO Alsatian;
  AnimalPtr   = POINTER TO Animal;

VAR
  beast1, beast2: Animal;
  pooch: Dog;
  fang : Alsatian;
  satanPtr, killerPtr : AlsatianPtr ;
  beastPtr : AnimalPtr;

PROCEDURE Action1(cur: Dog);
BEGIN
  ...
END Action1;

PROCEDURE Action2(VAR being: Animal);
BEGIN
  ...
END Action2;
```

Rule 1

Two class instances are compatible for assignment only if they are instances of the same class:

```

beast1 := beast2;
(* valid; both of class Animal *)
```

If range checks are enabled, the compiler will produce a run-time error if this rule is not obeyed.

Rule 2

An instance of a class is valid as a value parameter to a procedure, only if the actual parameter and the formal parameter are of the same class:

```

Action1(beast1);
(* invalid; beast1 of class Animal *)
Action1(pooch);
(* is valid; pooch is of class Dog *)

```

If range checks are enabled, the compiler will produce a run-time error if this rule is not obeyed.

Rule 3

An instance of a class is compatible with a formal variable parameter of a class type, if the type of the variable parameter is identical with the type of the actual parameter, or is a derived class of the formal parameter:

```

Action2(beast1);
(* is valid *)
Action2(pooch);
(* is valid as Animal is direct base class of
   Dog *)
Action2(fang);
(* is valid as Animal is base class of
   Alsatian *)

```

Rule 4

A pointer to a class instance is compatible, for assignment, to another class instance pointer in the following circumstances:

- The righthand and lefthand pointer types are identical.
- The righthand pointer is a pointer to a derived class of the lefthand pointer's type.

```

satanPtr := killerPtr; (* is valid *)
beastPtr := satanPtr;  (* is valid *)

```

The Checked Guard Operator

The checked guard operator performs a safe type conversion between objects of different types. This allows access to the fields and methods of one class, via a conversion to an object of the other type.

Syntax

```
CheckedGuardOp ::= Expr '::' QualName
```

This is possible when the actual class of the expression is a derived class of the static view of the expression, although the operator behaves differently depending on the type of the expression:

- If the type of the expression is a class T1, then the *QualName* must be a class name T2.
- If the expression is of type pointer to class T1, then the *QualName* must denote a pointer type to a class T2.

T2 can be any class relating to T1:

- T1 and T2 can be identical.
- T1 can be a base class of T2.
- T1 can be a class derived from T2.

However, the only interesting case is when T2 is a class derived from T1, either directly or indirectly.

If the conversion is not valid, and guard checks are enabled, a runtime error will occur. If T1 is a pointer type, and the expression is NIL, then the result of the conversion will still be NIL. For example:

```

CLASS T1;
  x:INTEGER;
END T1;

class T2 (T1);
  y:INTEGER;
END T2;

...

PROCEDURE p(VAR f:T1);
BEGIN
  IF f IS T2 THEN
    f.x := 200;
    (f::T2).y := 300;
  ELSE
    f.x := 100;
  END;
END p;

TYPE p1= POINTER TO T1;
      p2= POINTER TO T2;

VAR
  v1:p1;
  v2:p2;
  o1:T1;
  o2:T2;
BEGIN
  NEW(v2);
  v1 := v2;
  (v1::p2)^.y := 100;
  p(o1);
  p(o2);
END

```

The IS Operator

The IS operator is used to determine the class of the actual parameter passed to a procedure or function that has a formal variable parameter of some base class type. It is also used to determine the class of an object referenced by a pointer whose domain type is of some base class.

The operator takes two operands. The left hand operand must be an instance of a class and the right hand operand must be a class type derived from the class of the left hand operand. The operator tests to see if the left hand operand is of a derived class of the right hand operand; if this is the case the operator returns TRUE, otherwise it returns FALSE.

Syntax

Expr ::= ClassInstance IS ClassIdentifier

Example (1):

```

CLASS classOne;
...
END classOne;
CLASS classTwo (classOne);
...
END classTwo;
CLASS classThree (classOne);
...
END classThree;
...
VAR
    two   : classTwo;
    three: classThree;
    b     : BOOLEAN;

PROCEDURE test(VAR c1: classOne): BOOLEAN;
BEGIN
    RETURN := (c1 IS classTwo)
END test;

BEGIN
    b := test(two);  (* returns TRUE *)
    b := test(three); (* returns FALSE *)
END;
```

Example (2):

```
TYPE
  classOnePtr = POINTER TO classOne;
  classTwoPtr = POINTER TO classTwo;
  classThreePtr = POINTER TO
                      classThree;
VAR
  c1Ptr: classOnePtr;
  c2Ptr: classTwoPtr;
  c3Ptr: classThreePtr;
BEGIN
  ...
  c1Ptr := c2Ptr;
  b := (c1Ptr^ IS classTwo);
  (* is TRUE *)
  b := (c1Ptr^ IS classThree);
  (* is FALSE *)
END
```

Virtual and Static Methods

In TopSpeed Modula-2 methods can either be *virtual* or *static*. Unless they are explicitly flagged as being virtual in the class interface declaration, all class methods are by default static methods. Static methods are so named because they are called using early binding whilst virtual methods are called using late binding. Early binding is the name given to the mechanism where by the compiler resolves references to procedure or function calls at compile time, whereas late binding is where this resolution of procedure or function calls is deferred to run-time.

Virtual Methods

When compiling a class, the compiler inserts an extra field into the class called the ‘Virtual Method Table’ pointer. This table contains, amongst other things, pointers to virtual methods’ code. All instances of a class contain a pointer to their class’s VMT.

A method becomes virtual if the VIRTUAL keyword is used in the method declaration and definition. The virtual keyword must precede the PROCEDURE keyword in the class interface declaration. Once a method has been declared virtual in a class, any derived class that redefines that method must also declare the method as virtual.

When the compiler comes across a call to a virtual method, the source code references to that method are not replaced with the address of the method’s code. Instead, since the VMTs for each class have already been built, code is generated to dereference the pointer to that object’s VMT at run-time. This ensures that the correct code pointer will be efficiently selected, allowing the method to be called indirectly. In this manner the binding of virtual method calls can be deferred to run-time.

The advantage of this late binding becomes significant when you consider the inheritance mechanism and the compatibility rules regarding classes:

- Instances of derived classes may be assigned to instances of their base classes via pointers.
- Instances of derived classes may be passed as actual parameters to procedures or functions whose formal variable parameters are base classes of the actual parameters.

In such situations the compiler must use late binding to resolve references to method calls. The type of the calling object cannot be known until run-time.

Static Methods

The late binding associated with virtual methods has, of course, an overhead. At run-time, methods must be obtained by dereferencing an object's pointer to its VMT, and again dereferencing that pointer to call the appropriate method.

Omitting the keyword `VIRTUAL` instructs the compiler to compile the method call in the same way as it would a standard Modula-2 function or procedure call. The reference to the method call is replaced with the address of the method's code. This technique implements early binding. For example:

```

CLASS classOne;
PROCEDURE p(ch:CHAR);
END classOne ;

CLASS classTwo (classOne);
PROCEDURE p(i:INTEGER);
(* over ride method p *)

```

Notice that for static methods, a derived class does not have to match the parameter list, when over-riding that method.

Care must be taken when using static method. Problems can occur in three situations:

- When a class instance is a VAR parameter to a PROCEDURE or function.
- When a pointer to a class instance is assigned to a pointer whose domain type is a base class of the first pointer.
- when reference is made to the self parameter within a class method.

The first situation comes about because of compatibility rule 3 described earlier:

An instance of a class is compatible with a formal variable parameter of a class type, if the type of the variable parameter is identical with the type of the actual parameter, or is an base class of the actual parameter.

From this rule we can see that if a formal variable parameter is of some class type, then it is perfectly valid for an actual parameter to be of some derived class of that formal parameter.

However, if this actual parameter has methods that are static, and if those methods are called within the body of the PROCEDURE, the consequence of early binding will be that when a call is made to one of the actual parameters methods, the method that will be called will be a method of the formal parameter's class, not the method of the actual parameters class. This may cause a problem if the method called has be redefined by the derived class.

The second situation comes about because of compatibility rule 4 described in more detail in the section on Compatibility Rules earlier on in this chapter.

A pointer to a class instance is compatible, for assignment, to another class instance pointer in the following circumstances:

- *The righthand and lefthand pointer types are identical.*
- *The righthand pointer is a pointer to a derived class of the lefthand pointer's type.*

Take a look at this simple example:

```

TYPE
  onePtr = POINTER TO classOne;
  twoPtr = POINTER TO classTwo;
CLASS classOne;
  PROCEDURE p;
END classOne;

CLASS IMPLEMENTATION classOne;
PROCEDURE p;
BEGIN
  WrStr('hello');
  WrLn;
END p;
BEGIN
END classOne;

CLASS classTwo (classOne);
  PROCEDURE p;
END classTwo;

CLASS IMPLEMENTATION classTwo;
PROCEDURE p;
BEGIN
  WrStr('goodbye');
  WrLn;
END p;
BEGIN
END classTwo;
...
VAR
  ptr1 : onePtr;
  ptr2 : twoPtr;
BEGIN
  NEW(ptr2);
  ptr1 := ptr2;
  ptr1.p;
  (* 'hello' written to the screen *)
END

```

Because the method procedure p has not been declared as virtual, the call ptr1.p will result in the method declared in classOne being called, even though ptr1 has been assigned a value which is a pointer to an instance of classTwo.

The last situation whereby declaring a procedure as static can cause a problem, is where reference is made to the self parameter. This can be illustrated by the hierarchy of classes in the example on the following pages.

```

CLASS Animal;
  PROCEDURE talk;
  VIRTUAL PROCEDURE character;
END Animal;

CLASS Dog (Animal);
  PROCEDURE talk;
END Dog;

CLASS Alsatian (Dog);
  PROCEDURE talk;
END Alsatian;

CLASS IMPLEMENTATION Animal;

VIRTUAL PROCEDURE talk;
BEGIN
  IO.WrStr('I am an abstract base ',
          'class, I have no ',
          'particular characteristics');
  IO.WrLn;
END talk;

PROCEDURE character;
BEGIN
  talk;
  (* method invoked on the implicit
    self parameter *)
END character;

BEGIN
END Animal;

CLASS IMPLEMENTATION Dog;

VIRTUAL PROCEDURE talk;
BEGIN
  WrStr('I have 4 legs, a tail,'+
        'I am covered in hair'+
        'and I bark');
  WrLn;
END talk;

                                (continued over)

BEGIN
END Dog;

CLASS IMPLEMENTATION Alsatian;

VIRTUAL PROCEDURE talk;
BEGIN
  Dog.talk;
  WrStr('I am also a big dog and'+
        'can be quite aggressive'+
        'if I feel like it,'+
        'snarl gurr');
  WrLn;
END talk;

```

```
BEGIN
END Alsatian;

VAR
  satan: Alsatian;
BEGIN
  satan.character;
...

```

When the compiler encounters Animal's method for talk, it will generate code, and place it into a code segment. Next, it encounters the Animal's character method, which makes a call to Animal's talk method. Because talk is static, the compiler must make a direct call to Animal's talk, replacing the source code reference to talk with the actual address of Animal talk's code.

When the statement satan.character is executed, the call to talk will already have been bound to the character method in the base class Animal. The result of satan.character will be "I am an abstract base class, I have no particular characteristics". Clearly not what is expected from an Alsatian! The result of declaring talk as a static method will have been to nullify the over-riding of the talk method in the derived class.

Case Study

In this section shall describe, how, by using TopSpeed Modula-2's Object Oriented Extensions, we can create a linked list that can store values of any type. This will be achieved by making use of inheritance; the value we wish to store in a list will be an instance variable of a derived class of the base class Element. First of all, let's see how we write the definition.

```

DEFINITION MODULE lists;
TYPE
  elementPointer = POINTER TO Element;

CLASS Element;
  link:elementPointer;
  VIRTUAL PROCEDURE Compare(
    p: elementPointer): BOOLEAN;
  (*
    Must be implemented by the client.
    This PROCEDURE should return a
    boolean value: TRUE if 'THIS'
    equals 'p' and FALSE if 'THIS' not
    equal to 'p'
  *)
  VIRTUAL PROCEDURE print;
  (* Must be implemented by the client.
    This PROCEDURE should return the
    value of the instance variables
  *)
END Element;

```

Notice first our declaration for the class Element. Note that Element has only one instance variable (link) of type elementPointer. Note also that its two methods are deferred methods, so called because the methods will be fleshed out by derived classes of Element. This is because all other element class types will inherit from this base class.

Now lets take a look at the class interface declaration for a GenericList. It is instances of this class that will be used to store instances of derived classes of the Element class:

```

CLASS GenericList;
  ptrToLast:elementPointer;
  PROCEDURE initList;
  PROCEDURE addToList(VAR el:Element);
  PROCEDURE searchList(
    VAR el,result:Element;
    VAR found:BOOLEAN);
  PROCEDURE printList;
END GenericList;

```

GenericList has one instance variable (ptrToLast), which will be used to point to the last element entered into the list and is of type elementPointer. The methods associated with class GenericList are those that you might expect: initList, addToList, searchList and printList, which respectively,

initialize the list, add an element to the list, search the list for a particular element and print out the values of the list.

Now let's look at the implementation module lists and firstly the implementation of the class Element:

```
IMPLEMENTATION MODULE lists;
FROM Storage IMPORT ALLOCATE,
                    DEALLOCATE;
FROM Str IMPORT WrStr;

CLASS IMPLEMENTATION Element;

VIRTUAL PROCEDURE Compare(
    P: elementPointer): BOOLEAN;
(*
  It is an error not to supply an
  implementation of this method. The
  client MUST supply a method to
  compare 'THIS' with 'p'.
*)

BEGIN
    WrStr(' implemented by client ');
    HALT;
    RETURN FALSE;
END Compare;

VIRTUAL PROCEDURE print;

BEGIN
    WrStr('print method must be '+'
        'overridden by derived '+'
        'class');
    HALT;
END print;

BEGIN
    (* Element class initialization block *)
END Element;
```

Here we can see that if any of the class Element's methods are called, the program will halt. That's ok because these methods should never be called anyway! Methods of the same names *will* be called for derived classes of class Element. Remember we do not really want to store objects of class Element (because they don't contain any meaningful data), but derived classes of class Element.

More enlightening perhaps is the implementation of the class GenericList:

```

CLASS IMPLEMENTATION GenericList;
PROCEDURE initList;

BEGIN
    ptrToLast := NIL;
END initList;

PROCEDURE addToList
    (VAR e1:Element);
VAR
    newElement: elementPointer;
BEGIN
    ALLOCATE(newElement,SIZE(e1));
    Move(e1,newElement^,SIZE(e1));
    newElement^.link := ptrToLast;
    ptrToLast := newElement
END addToList;

PROCEDURE searchList
    (VAR e1,result:Element;
     VAR found:BOOLEAN);
VAR
    thePtr: elementPointer;
BEGIN
    found := FALSE;
    thePtr := ptrToLast;
    WHILE (thePtr # NIL)
        and (not found) DO
        found := e1.compare(thePtr);
        IF not found THEN
            thePtr := thePtr^.link;
        END;
    END;
    Move(thePtr^,result,SIZE(result));
END searchList;

PROCEDURE printList;
VAR
    thePtr:elementPointer;
BEGIN
    thePtr := ptrToLast;
    WHILE thePtr # NIL DO
        thePtr^.print;
        thePtr := thePtr^.link;
    END;
END printList;

BEGIN
    (*
    GenericList Class initialization block
    *)
    END GenericList;

    END lists.

```

First, notice the procedure `addToList`. To create storage for a variable of type `elementPointer`, we use the procedure `ALLOCATE` instead of the procedure `NEW` that you would normally use. This is because `ALLOCATE` allows you to specify the amount of storage that a pointer will reference. We need to do

this because ,although objects of class GenericList expect to store elements of class Element, in reality they will be storing elements which are derived classes of class Element. Inevitably, instances of these derived classes may need more storage.

Once the storage for a variable of type elementPointer has been created, that variable is dereferenced and assigned the parameter el (using the Lib.Move procedure). Although el is declared in the formal parameter list as being an object of class type Element, it will in reality be an object whose type is a derived class of class Element. Remember that it is valid to pass to the procedure addToList a variable which is a derived class of the class Element.

Next take a look at procedure searchList. Notice the following line:

```
found := el.compare(thePtr);
```

The method compare must be redefined for each derived class of class Element, because each derived class will have different data fields/instance variables.

Finally let's look at the printList procedure/method, and in particular the following statement:

```
thePtr^.print;
```

The method print must also be redefined for each derived class of class Element, since each derived class will have different data fields/instance variables.

Now we have defined our classes Element and GenericList, we can write a program called listUser that imports them. First of all we declare a derived class of Element called strElement:

```
MODULE listUser;
FROM lists IMPORT GenericList ,Element;
FROM Str    IMPORT WrStr,WrLn;

TYPE theString = ARRAY [0..23] OF CHAR;

CLASS strElement ( Element ) ;
  value:theString;
  VIRTUAL PROCEDURE compare(
    P: elementPointer): BOOLEAN;
  VIRTUAL PROCEDURE assign(str:theString);
  VIRTUAL PROCEDURE print;
END strElement;

      (continued over)
CLASS IMPLEMENTATION strElement;

VIRTUAL PROCEDURE compare
  ( p : elementPointer ) : BOOLEAN;
BEGIN
  RETURN
    Str.Compare(value ,
      (p^::strElement).value)=0;
END compare;
```

```

VIRTUAL PROCEDURE assign(
    str:theString);

BEGIN
    value := str
END assign;

VIRTUAL PROCEDURE print;

BEGIN
    WrStr(value); WrLn;
END print;

BEGIN
    (* CLASS initialization block *)
END strElement;

```

Unlike the base class `Element`, `strElement` has an instance variable called `value`, which is the data we want to store in a list. It also redefines the function/method `compare`. Notice the following statement:

```
RETURN (value =(p^::strElement).value);
```

This method will be called by an instance of class `GenericList`, to compare each element that is stored in the list. The guard operator is used to convert the object `p^` (which is of class `Element`) into an object of class `strElement`, in order that the data field `value` can be accessed and compared with some target value.

The program `listUser` also declares a derived class called `recElement`:

```

TYPE
    ageType = [0..101];
    theRecord = RECORD
        name: theString;
        age: ageType
    END;

CLASS recElement (Element);
    value:theRecord;
    VIRTUAL PROCEDURE compare(
        P: elementPointer): BOOLEAN;
    VIRTUAL PROCEDURE assign(
        rec:theRecord);
    VIRTUAL PROCEDURE print;
END recElement;

CLASS IMPLEMENTATION recElement;

VIRTUAL PROCEDURE compare
    ( p : elementPointer):BOOLEAN;
BEGIN
    RETURN Str.Compare(value.name,
        (p^::recElement).value.name));
END compare;

VIRTUAL PROCEDURE assign(rec:theRecord);

```

```

BEGIN
    value := rec;
END assign;

VIRTUAL PROCEDURE print;

BEGIN
    WrStr(value.name);
    WrLn;
    WrCard(value.age,1);
    WrLn;
    WrLn;
END print;

BEGIN
    (* CLASS initialization block *)
END recElement;

```

This derived class looks very similar to the class strElement, the main difference being that the instance variable value (the data we wish to store in a list) is a record type rather than being a string type.

Next we make the following variable declaration:

```

VAR
    aList: GenericList;

```

This same list will be used in turn to store objects of class strElement and then objects of class recElement.

Next we need to declare a non-class procedure that will create an object that will hold the string we want to put in the list.:

```

PROCEDURE addStr(str: theString);

VAR
    s: strElement;
BEGIN
    s.assign(str);
    aList.addToList(s);
    (* add the object to the list *)
END addStr;

```

Similarly we need to declare a non-class procedure that will create an object that will hold the record we want to put in the list.:

```

PROCEDURE addRec(
    str: theString;
    years: ageType);

VAR
    r: recElement;
    rec: theRecord;
BEGIN
    rec.name := str;
    rec.age := years;
    r.assign(rec);
    aList.addToList(r);
END addRec;

```

Also we need a procedure that will create an object that will hold the string we wish to search for in the list.

```
PROCEDURE getStr(str: theString);
VAR
  target, result:strElement;
  found: BOOLEAN;
BEGIN
  found := FALSE;
  target.assign(str);
  (* create a target object *)
  (* search the list for an identical *)
  (* value *)
  aList.searchList(target,result,found);
  IF found THEN
    WrStr('the value of the'+
          ' string is ');
    result.print;
  ELSE
    WrStr('value not found');
    Writeln;
  END;
END getStr;
```

And another procedure that will do the same for records:

```
PROCEDURE getRec(str: theString);
VAR
  target, result:recElement;
  rec:theRecord;
  found: BOOLEAN;
BEGIN
  rec.name := str;
  target.assign(rec);
  aList.searchList(target,result,found);
  IF found THEN
    WrStr('the value of the record'+
          ' element is: ');
    Writeln;
    result.print;
  ELSE
    WrStr('value not found');
    Writeln;
  END;
END getRec;
```

Now we have written the code we can store values into the list:

```
BEGIN (* main program *)
  aList.initList;
  (* initialize the list of strings *)
  addStr('bill');
  (* add elements to the list *)
  addStr('fred');
  addStr('joe');
  addStr('henry');
  addStr('sue');
  aList.printList;
  getStr('joe');
  (* get a value from the list *)
```

```
aList.initList;
  (* initialize the list of records *)
addRec('bill',34);
  (* add elements to the list *)
addRec('fred',2);
addRec('joe',18);
addRec('henry',23);
addRec('sue',30);
aList.printList;
getRec('henry');
  (* get a value from the list *)
END listuser.
```

The object aList first stores elements of class strElement and then elements of class recElement.

CHAPTER 4

LIBRARY SUMMARY

This chapter provides an overview of the procedures, types and constants defined by the library modules supplied with TopSpeed Modula-2. This chapter should be used in conjunction with the next chapter, *The TopSpeed Modula-2 Library Reference*.

Each of the modules described in these chapters have an associated definition (.DEF) file. However, all of the implementation modules are not written in Modula-2.

Overview

Modula-2 relies heavily on library modules to provide such services as *file I/O*, *screen and keyboard interfaces*, *mathematical function support* and *storage management*. These services are provided in two ways:

- 1 By using a set of .LIB files which contain the object code for a number of modules - these are written in Assembler and Modula-2. Most of the modules in this category are to be found in the standard libraries.
- 2 By using Modula-2 modules in a single .OBJ file.

Modula-2 requires a .DEF file to interface with these modules, but this does not mean that a separate object file exists for the module (or that a separate source file exists for the module).

Modules (such as AsmLib) which provide low-level routines in other languages, and which are referenced using the Modula-2 external procedure facility, are not described here. The library reference describes each module in terms of the .DEF files you must IMPORT, rather than details of the implementation of the modules.

Also, modules providing interfaces to OS/2 calls (such as VIO) are not documented in this manual. The calls accessed in these modules are described in detail in the appropriate OS/2 documentation.

Those modules specific to OS/2 are clearly noted in the following summary.

Library Summary

This section summarizes the contents of each library module. each of the modules are explained in more detail in the *The TopSpeed Modula-2 Library Reference*.

BiosIO — (DOS Only)

The BIOSIO module provides direct access to the BIOS console I/O routines in DOS. It allows:

- Direct keyboard input.
- Monitoring of the Shift, Ctrl and Alt keys.

Dev — (OS/2 Only)

The Dev module is an interface to OS/2 providing support for the development of OS/2 device drivers. The details of this module are not discussed here.

Dos — (OS/2 Only)

The DOS module provides an interface to the OS/2 kernel. This module is implemented through the OS/2jpi object library and *must* be imported in every OS/2 project. The details of the module are not discussed here.

Err — (OS/2 Only)

The ERR module simply defines the OS/2 kernel error numbers in symbolic fashion. It is for use with the DOS module above.

FIO —

The FIO module allows access to the filing system, files, character devices and pipes of the operating system. File access can be either to unformatted binary data or formatted text. The files are usually disk-based but may be on a media that supported by the operating system. The module covers:

- Creating, deleting, renaming, opening and closing files.
- Reading and writing data.
- Positioning the filer read/write pointer.
- Creating, deleting and changing directories.
- Directory scanning.
- Reading and writing character, booleans, integers, cardinals, reals and strings.

FIOR —

The FIOR module contains procedures for opening and closing files using the TopSpeed redirection file to redirect file references. In addition to the file-handling functions it includes procedures to manipulate and expand file names using the redirection file.

FloatExc —

The FloatExc module supports 80x87 exception handling, allowing this feature to be enabled and disabled as required.

FormIO

The FormIO module provides formatted output to the console of up to five arguments.

Isp — (OS/2 Only)

The Isp module provides definitions of constants and records for use with the OS/2 Dialog Manager.

Gpi — (OS/2 Only)

The Gpi module provides access to the graphics program interface of OS/2 Presentation Manager. The implementation of this module is in the pmjpi object library. Details of this module are not given here.

Graph

The Graph module implements a simple graphics library supporting the following graphics boards:

CGA
EGA
VGA
Hercules

The Graph module includes the following graphics functions:

- Selecting and initializing a graphics board.
- Selecting screen modes.
- Writing and reading single pixels.
- Drawing lines and circles.
- Drawing filled circles and polygons.

The module also provides functions from the TopSpeed graphics library.

IO

The IO module provides formatted input and output routines for the standard I/O devices (keyboard and screen). Facilities exist within the module definition for redirecting the I/O to other devices or device handlers. The module also contains functions dealing with direct keyboard input. The IO module includes the following facilities:

- Reading and writing characters, booleans, integers, cardinals, reals and strings.
- Redirection of input and output to any device.
- Reading characters directly from the keyboard.
- Testing the keyboard for a keypress.

Kbd — (OS/2 Only)

The Kbd module provides access to the OS/2 KBD functions for keyboard handling. The large range of functions provided make it impossible to discuss the module fully here. Please refer to the appropriate OS/2 documentation for full details of the functions available.

Lib —

The Lib module is a miscellaneous collection of useful functions and procedures. Parts of it are implemented in Modula-2 while the rest are aliases into the AsmLib module. It covers:

- Sorting.
- Random number generation.
- DOS interrupt services.
- Command line processing.
- Long jumps.
- Address arithmetic.
- *Ctrl-Brk* and error handling.
- Sound procedures.

LIM

The LIM module provides an interface to LIM expanded memory.

MATHLIB —

The MATHLIB module defines the common mathematical functions. It is implemented in assembly language. Specifically, MATHLIB covers:

- Trigonometric and hyperbolic functions on real numbers.
- Logarithmic functions on real numbers.
- Conversion of real numbers to and from binary coded decimal format.
- 80x87 co-processor specific functions.

Mou — (OS/2 Only)

The Mou module is the interface to the OS/2 Mouse functions. The large number of functions available make it impossible to discuss here. Please refer to the appropriate OS/2 documentation for further details.

MsMouse

The MsMouse module provides an interface to the Microsoft Mouse driver. It contains the full set of interface routines for all the standard mouse calls under DOS.

OS2DEF — (OS/2 Only)

The OS2DEF module defines the data structures and constants. This module has no implementation as it contains no procedures. This module is needed by all the OS/2 service modules: Kbd, Gpi, Win, Spl, Vio and Mou.

Pic — (OS/2 Only)

The Pic module provides definitions of constants and functions for use with the OS/2 Window Manager.

PMErr — (OS/2 Only)

The PMErr module defines the constants for the error returns from OS/2 Presentation Manager. It has no implementation and is not discussed in detail here.

Process

The Process module implements a multi-process scheduler with time-sliced process switching and a semaphore based system for process synchronization. It covers:

- Starting and stopping the scheduler.
- Starting up processes.
- Synchronization by means of semaphores.

ShtHeap

The ShtHeap module allows you to define and use a *short heap*. A short heap is an a dynamically usable area of memory within the current segment. It is particularly useful when the dynamic memory requirements of a module are small. The module uses the memory allocation facilities of the operating system.

Spl — (OS/2 Only)

The Spl module gives the programmer access to the OS/2 Presentation Manager print spooler. This module is not discussed in detail here and you should refer to the appropriate OS/2 documentation for further information.

Storage

The Storage module is a dynamic heap management module. It provides:

- Allocation and deallocation of memory from a heap.
- Information on availability of memory within a heap.

Str

The Str module implements string handling procedures using open array parameters (ARRAY OF CHAR). The features implemented are:

- String concatenation, appending, insertion and deletion.
- String comparison.
- String searching and matching.
- Conversion of strings to and from numeric types.

SYSTEM

The SYSTEM module is unique in that it does not make use of any other module. Also, SYSTEM contains many compiler and implementation dependent features (such as Seg() and OfS()). Some of the features are actually implemented within the compiler rather than within the SYSTEM module. For this reason, the SYSTEM module is a *pseudo-module*. The SYSTEM module covers:

- Support for using the specific features of the 80x86 family of micro-processor.
- Definitions for data structures associated with the 80x86 family of micro-processors.
- Support for concurrent processes.

Vio — (OS/2 Only)

The Vio module provides access to the OS/2 screen I/O facilities. This module is implemented in the OS2jpi import library. Because of the large number of functions provided, this module is not defined in detail here and you should refer to the appropriate OS/2 documentation for further information.

Win — (OS/2 Only)

The Win module provides the interface to the OS/2 Presentation Manager window management functions. It is implemented in the pmjpi import library. Due the extensive nature of the facilities provided, this module is not discussed here and you should refer to the appropriate OS/2 Presentation Manager documentation for further information.

Window —

The Window module implements a powerful text-based windowing environment for your programs. By directing output to defined areas of the screen (known as *windows*) you can create overlapping output areas which act as individual screens anywhere on the physical screen. The Window module includes:

- Creating, destroying, opening and closing windows.
- Setting window attributes such as color, frame and title.
- Rearranging the size, position and layering of the displayed windows.
- Inserting and deleting lines from windows.
- Palette windows.
- Writing to windows using the IO module.

OS/2 Library Interface

As mentioned above, a number of the library modules provide interfaces to OS/2 and Presentation Manager. These have been implemented to allow programmers to use the services provided by the OS/2 system *Dynamic Link Libraries*.

In order to fully understand these services and to discover their function you should refer to the appropriate OS/2 documentation. This documentation includes: *IBM Operating System/2 Technical Reference*, Ed Iacobucci's *OS/2 Programmer's Guide* (Osbourne/McGraw Hill, 1988) and Ray Duncan's *Advanced OS/2 Programming*. These volumes are extensive and complete and provide background you need to write OS/2 and PM programs.

When using the TopSpeed Modula-2 OS/2 interface libraries, it is important to note the naming convention which has been used. The OS/2 standard is to name the functions by simply prefixing the name of the module (for example, DOS) to the procedure name (for example, Beep) to obtain the full name of the procedure (in this case, DosBeep).

This convention could have been achieved in TopSpeed Modula-2 by extensive use of the name pragma:

```
(*# name( prefix => OS/2_lib ) *)
```

However, this would have required you to remember a new syntax as Modula-2 normally separates the procedure name from the module name with a dot ('.'). For this reason, the OS/2 library modules have been implemented to follow the Modula-2 naming convention. What this means in practice is that the OS/2 function call:

```
MouClose
```

is referenced as:

```
Mou.Close
```

in TopSpeed Modula-2. This means that cross-references between the above mentioned publications and the TopSpeed .DEF files should be attempted *after* stripping the module name from the functions described in the documentation.

The OS/2 DLLs for which TopSpeed Modula-2 provides an interface are:

Mou	Mouse driver.
Kbd	Keyboard driver.
Gpi	Graphic program interface for PM.
Vio	Character oriented display interface.
Win	Window manager for PM.
Dos	OS/2 kernel services.
Spl	Print Spooler for PM.

Modula-2 Core Library

Appendix 2 of Niklaus Wirth's *Programming in Modula-2*, lists a number of *Standard Utility Modules*. While most of the functionality of these modules is duplicated elsewhere in the TopSpeed system, there are numerous books and tutorials containing examples based on Wirth's standard modules.

TopSpeed Modula-2 includes an implementation of many these modules, as follows:

Terminal	Simple I/O interface to the screen and keyboard.
FileSystem	A file I/O module.
InOut	A sequential text file processing module with formatted I/O.
RealInOut	An extension to InOut for processing REAL numbers.
MathLib0	A basic library of mathematical functions.

These modules are all available for `IMPORT` into your programs.

The details of these modules are not described in this manual. For further information please refer to the book.

CHAPTER 5

LIBRARY REFERENCE

This chapter describes the procedures constituting each of the modules in the TopSpeed Modula-2 library. The modules are listed in alphabetical order to enable a quick reference to the library routines.

Each module description begins with an explanation of the purpose of the module. Any variables, records and special considerations are documented in this section. This is followed by details of the individual procedures in the module, again in alphabetical order.

As far as possible, each procedure is illustrated with a commented example. Certain simple procedures (for example `Sqrt`, which calculates the square root of a number) do not have examples, as their behavior is easily inferred from their description.

MODULE *BiosIO* (DOS Only)

The `BiosIO` module provides direct access to the BIOS console service routines. This allows you to not only directly read characters from the keyboard (with and without echoing the resulting character to the screen), but also to detect the state of the *modifier* keys (*Shift*, *Ctrl*, etc).

Note: This module can only be used with DOS programs. If you are making OS/2 programs, then use the `Kbd` module instead.

BiosIO Reference

The following are the individual procedures defined within the BiosIO module.

KeyPressed — Has a key been pressed?

PROCEDURE KeyPressed() : BOOLEAN;

Like the KeyPressed procedure in the IO module, this procedure returns TRUE if the user has pressed a key (i.e., the keyboard buffer contains at least one character). It will continue to return TRUE while there is a key waiting. A call to RdKey or RdChar (see below), will clear a single key from the buffer.

RdKey — Read character without screen echo

PROCEDURE RdKey() : CHAR;

This procedure reads a single character from the BIOS keyboard buffer and returns it. The character is *not* displayed on the screen.

RdChar — Read character with screen echo

PROCEDURE RdChar() : CHAR;

This procedure reads a single character from the BIOS keyboard buffer and returns it. The character is displayed on the screen at the current cursor position, in the current color.

KBFlags — Determine state of ‘modifier’ keys

```
PROCEDURE KBFlags() : KBFSet;
TYPE
  KBFSet = SET OF
    ( RShift, LShift, Ctrl, Alt, Scroll, Num,
      Cap, Ins );
```

This function allows you to determine the state of the *modifier* and *toggle* keys on the keyboard. A call to KBFlags returns a set with the relevant members set if they are currently being held down (for RShift, LShift, Ctrl and Alt) or if they are ON (for Scroll, Num, Cap and Ins). These abbreviations correspond to the following keys:

RShift	The right-hand SHIFT key.
LShift	The left-hand SHIFT key.
Ctrl	The CONTROL key.
Alt	The ALT key.
Scroll	The SCROLL LOCK key.
Num	The NUM LOCK key.
Cap	The CAPS LOCK key.
Ins	The INS key on the Numeric/Cursor Control pad.

For example:

```
IMPORT BiosIO, IO;
...
VAR
  KBState : BiosIO.KBFSet;
...
KBState := BiosIO.KBFlags();
IF (LShift IN KBState) THEN
  IO.WrStr("The left shift key is being
    pressed");
END;
...
```

MODULE FIO

Introduction

The FIO module contains procedures for file handling, directory handling and file input/output operations. These procedures are described below.

The contents of FIO are intimately bound up with the workings of the operating system. This relationship implies:

- File and directory names must conform to conventions of the operating system.
- Once open, files are referred to by a small positive integer known as a *file handle*.
- File errors are reported using the standard operating system error numbers.
- The *standard file handles* of the operating system are open when your program commences.

File Handles

Before a file can be used (i.e., read from or written to) it must have been *opened* using one of the procedures Open, Create or Append. These procedures return a value of type File, which is a small positive integer known as a *file handle*. Each open file has a different file handle, allowing the operating system to distinguish between them. The value of the file handle for a particular file is not significant nor guaranteed (the *standard* file handles are an exception to this).

After a file has been opened, and a file handle established for it, subsequent actions on the file are carried out using that file handle.

All files should be closed when your program has finished using them or, at the very latest, before your program terminates. The Close procedure is used to close the file associated with a particular file handle.

File Positioning

All files are regarded by the FIO module as sequential streams of bytes, i.e., they are *sequential files*. However, it is possible to access different parts of a file by changing the value of the *file position* for a particular file.

When a file is opened, the file position is set to zero (i.e., the first byte in a file is numbered as 0). As the file is read from (or written to) the file position is incremented by one for each byte read (or written). It is possible to change the current file position (forwards or backwards in the file) by using the Seek procedure. The Seek procedure takes a LONGCARD parameter which specifies the next reading (or writing position). This allows the FIO module to work with files up to about $2^{32}-1$ Bytes (over 4,000Mb) which should be sufficient for most purposes.

Buffers

By default, the files managed by FIO are unbuffered. This means that every read (or write) causes a disk access. This is the safest way to write files (all new information is written directly to the disk). However, this safety involves a time penalty. You can provide a *buffer* for an open file using the AssignBuffer procedure, described below.

Definitions in FIO.DEF

The FIO.DEF file defines one TYPE and several constants and variables. These are used to control the behavior of the file handling and I/O routines as well as to inspect the results of these routines. Some of the variables are duplicated in the IO modules where their function is identical. The definitions are:

```

CONST
  DiskFull = 0FOH;
  (* Error code if Write fails with disk full*)
  (* DOS standard handles *)
  StandardInput = 0;
  StandardOutput = 1;
  ErrorOutput = 2;
  AuxDevice = 3;
  PrinterDevice = 4;
TYPE
  File = CARDINAL;(* File handle type *)
VAR
  RunTimeError : PROCEDURE( LONGCARD,
                           CARDINAL,
                           ARRAY OF CHAR);
  (* Run-time error handler *)
  IOcheck : BOOLEAN;
  (*
    If TRUE then I/O errors will terminate
    program
  *)
  EOF          : BOOLEAN;
  Separators   : Str.CHARSET;
  OK           : BOOLEAN;
  ChopOff      : BOOLEAN;
  Eng          : BOOLEAN;
  (*
    If TRUE then Engineering notation will
    be used for REALs
  *)

```

Predefined File Handles

The predefined file handles (StandardInput, StandardOutput, ErrorOutput, AuxDevice and PrinterDevice) are defined when your program begins execution and you do not have to open these files in order to use them. The files opened by you program will receive file handles in the range 5 to MaxOpenFiles, inclusive.

Input/Output Error Handling

Before a file can be used (i.e., read) the variable IOcheck determines the behavior of the FIO module in the event of an input/output error.

If IOcheck is TRUE then the procedures listed below will terminate your program in the event of an error. If IOcheck is FALSE then you can check the result of the procedure by a call to IOresult. The default setting for IOcheck is TRUE.

The procedures affected by the state of IOcheck are:

Open	Append
Create	Close
Truncate	GetPos
Seek	Size
Erase	Rename
ChDir	MkDir
RmDir	GetDir
ReadFirstEntry	
ReadNextEntry	

All recoverable run-time errors are handled by calling the procedure associated with the procedure variable RunTimeError. The default handler outputs an error message and aborts the process. In doing so the file ERRORINF.\$\$\$ is created. See Appendix A: '*Run-time Error Codes*'.

You can easily define your own error handler by assigning a different procedure to the variable RunTimeError. The procedure takes three parameters. The first is the address of the fault as a LONGCARD, the second is the error code, a CARDINAL, and the third is the error message, ARRAY OF CHAR.

Formatted Input/Output

Before a file can be used (i.e., read) the variable OK is set by all the formatted I/O procedures as well as WrBin and RdBin. If OK is TRUE, it indicates that the operation was successful. A failure of one of these procedures can be caused by more than normal input/output errors.

Each of the formatted output routines takes a width parameter. This specifies the maximum width for the formatted output. The behavior of these routines when this width is exceeded by the length of the formatted value to be printed is controlled by the `ChopOff` variable. If `ChopOff` is `TRUE`, the value is not written and a string of '?' is written instead. This string of '?' is as long as the specified width. If `ChopOff` is `FALSE` the width parameter is ignored when it is less than the actual size of the value and the value is written in as much space as required.

Other Variables

The other variables in the FIO module perform the following functions:

`Eng` indicates whether or not real numbers are output using engineering notation (see `RealToStr` for more information).

`EOF` indicates whether or not the end of a file was reached during the most recent read operation.

`Separators` is a set of delimiters used by the `RdItem` procedure. The default value of `Separators` is:

```
Str.CHARSET {  
    CHR(9),  
    CHR(10),  
    CHR(13),  
    CHR(26),  
}
```

That is, the default separators are *TAB*, *LF*, *CR*, the DOS end-of-file marker and *SPACE*.

`ShareMode` Determines the file sharing mode used when a file opened or created.

Sequential File Processing

The example program below demonstrates the use of FIO to copy one text file to another. It uses some procedures from the IO module and the Lib module, to print messages and to get the file names from the command line.

```

MODULE FileCopy;

IMPORT FIO, IO, Lib;
PROCEDURE CopyFile(In, Out: FIO.File): LONGCARD ;
(*
  Copies one open
  file to another,
  returning a count
  of the number of
  characters copied
*)
VAR
  ch : CHAR;
  (* Character read *)
  nb : LONGCARD;
  (* Counter *)
BEGIN
  nb := 0;
  LOOP
    ch := FIO.RdChar(In);
    (* Read a single char *)
    IF FIO.EOF THEN
      (* Exit LOOP on EOF *)
      EXIT;
    END;
    FIO.WrChar(Out,ch);
    (* Write the character *)
    INC(nb);
    (* nb := nb + 1; *)
  END;
  RETURN nb;
  (* Return no. of CHARs *)
END CopyFile;
PROCEDURE Error( Msg : ARRAY OF CHAR );
(* Prints an error message and HALTs the program *)
BEGIN
  IO.WrStr('*** ERROR : ');
  (* Displays a string *)
  IO.WrStr(Msg);
  IO.WrStr(' ***');
  IO.WrLn;
  (* Prints a newline *)
  HALT;
END Error;

```

```

PROCEDURE CheckParams(VAR In, Out: ARRAY OF CHAR );
(*
  Checks that the program has been used
  as: FILECOPY <input-file> <output-file>
  and that <input-file> exists and that
  <output-file> doesn't exist.
  The parameters are set to the input
  and output file names, respectively
*)
BEGIN
  IF (Lib.ParamCount() <> 2) THEN
  (*
    The ParamCount procedure returns a
    count of number of command line
    arguments
  *)
    Error("Usage - 'FILECOPY <input> <output>'");
  END;
  (*
    The ParamStr procedure fetches
    parameter strings from the command line
  *)
  Lib.ParamStr(In,1);
  Lib.ParamStr(Out,2);
  IF NOT FIO.Exists(In) THEN
    Error("Input file does not exist");
  END;
  IF FIO.Exists(Out) THEN
    Error("Output file already exists");
  END;
END CheckParams;
VAR (* Main program variables *)
  InFile,
  OutFile: FIO.File;
  (* Handles *)
  InFileName,
  OutFileName: ARRAY [0..64] OF CHAR;
  (* File Names *)
  IOerr : CARDINAL;
  (* IOresult *)
  BytesCopied : LONGCARD;
BEGIN
  FIO.IOcheck := FALSE;
  (* Program will trap errors *)
  CheckParams(InFileName,OutFileName);
  InFile := FIO.Open(InFileName);
  IOerr := FIO.IOresult();
  IF (IOerr <> 0) THEN
    Error("Cannot open input file");
  END;
  OutFile := FIO.Create(OutFileName);
  IOerr := FIO.IOresult();
  IF (IOerr <> 0) THEN
    Error("Cannot create output file");
  END;
  BytesCopied := CopyFile(InFile, OutFile);
  IO.WrStr("Copied ");
  IO.WrLngCard(BytesCopied,12);
  IO.WrStr(" bytes");
  IO.WrLn;
END FileCopy.

```

This example demonstrates the use of file handles and the FIO.Wrsimpletype procedures. Note that the program makes sure that the output file doesn't exist before attempting the copy. This means it must use FIO.Create to open the output file as FIO.Open will only work when the file specified already exists.

Note: The program does not specify what errors it found on Create or Open, this was done to keep the example short.

Random File Processing

The following example demonstrates how to use the FIO module to read and write a random (binary file). The program is simple: it creates a file consisting of 20 records (of type Person), containing some example data. It then reads back the data and displays it in the following ways:

- In record order.
- All the even-numbered records in ascending order of record number.
- All the odd-numbered records in descending order of record number.

Between the first and second displays, the program reads all the records and adds 1 to the Age field.

This serves to illustrate how a record structure can be stored to disk and retrieved. A word of warning, though, you should never store POINTER or ADDRESS type fields in a file as their value will almost certainly be invalid when they are read back in at a later time.

The example uses procedures from the IO module to display the information on the screen and the Lib module to generate random numbers.

```
MODULE RandDemo;
IMPORT FIO, IO, Lib;
CONST
  FileName = "PERSON.DAT";
  (* File to use *)
  AgeLo = 15;
  (* These constants are used *)
  AgeHi = 60;
  (* in the GenRec procedure *)
  HgtLo = 1.5;
  (* to generate the data *)
  HgtHi = 2.0;
  WgtLo = 200;
  WgtHi = 400;
```

```

TYPE
  NameStr = ARRAY [1..10] OF CHAR;
  SexType = ( Female, Male );
  Person   = RECORD
    Name : NameStr;
    Age  : SHORTCARD;
    Height: REAL;
    Sex  : SexType;
    Weight: CARDINAL;
  END;

VAR
  (* Program Variables *)
  FileH: FIO.File;
  (* File Handle *)
  CurrR: Person;
  (* Current Record *)
  CurrP: LONGCARD;
  (* Current Record No. *)

PROCEDURE SeekRec( RecNo : LONGCARD );
(*
  This procedure moves the file to
  record number RecNo (1 to 20)
*)
VAR
  pos: LONGCARD; (* Actual position *)
BEGIN
  pos := (RecNo - 1) * SIZE(Person);
  FIO.Seek(FileH, pos);
  CurrP := RecNo;
END SeekRec;
PROCEDURE WriteRec(KeepPos : BOOLEAN );
(*
  Writes the current record (CurrR) at
  the current file position, advancing
  the record pointer unless KeepPos
  is TRUE
*)
BEGIN
  FIO.WrBin(FileH, CurrR, SIZE(Person));
  IF KeepPos THEN
    SeekRec(CurrP);
  ELSE
    INC(CurrP);
  END;
END WriteRec;
PROCEDURE ReadRec(KeepPos : BOOLEAN );
(*
  Reads the current record (CurrR) at
  the current file position, advancing
  the record pointer unless KeepPos
  is TRUE
*)
BEGIN
  FIO.RdBin(FileH, CurrR, SIZE(Person));
  IF KeepPos THEN
    SeekRec(CurrP);
  ELSE
    INC(CurrP);
  END;
END ReadRec;

```

```

PROCEDURE GenRec;
(*
  Generates a record using random values
*)
VAR
  c: CHAR;
  i: CARDINAL;
BEGIN
  WITH CurrR DO
    c := CHR(Lib.RANDOM(26) + ORD('A'));
    FOR i := 1 TO 10 DO
      Name[i] := c;
    END;
    Age := SHORTCARD(Lib.RANDOM(AgeHi -
AgeLo) + AgeLo);
    Weight := Lib.RANDOM(WgtHi - WgtLo) + WgtLo;
    Height := (Lib.RAND() * (HgtHi - HgtLo)) + HgtLo;
    IF (Lib.RANDOM(100) < 50) THEN
      Sex := Female;
    ELSE
      Sex := Male;
    END;
  END;
END GenRec;
PROCEDURE DisplayRec;
(* Prints a record on the screen *);
BEGIN
  WITH CurrR DO
    IO.WrStr("RecNo  : ");
    IO.WrLngCard(CurrP,2);
    IO.WrLn;
    IO.WrStr("Name   : ");
    IO.WrStr(Name);
    IO.WrLn;
    IO.WrStr("Age     : ");
    IO.WrShtCard(Age,2);
    IO.WrStr(" years."); IO.WrLn;
    IO.WrStr("Weight  : ");
    IO.WrCard(Weight,3);
    IO.WrStr(" Kg."); IO.WrLn;
    IO.WrStr("Height  : ");
    IO.WrReal(Height,1,5);
    IO.WrStr(" m."); IO.WrLn;
    IO.WrStr("Sex     : ");
    IF (Sex = Female) THEN
      IO.WrStr("Female");
    ELSE
      IO.WrStr("Male");
    END;
    IO.WrLn;
    IO.WrLn;
  END;
END DisplayRec;

```

```

PROCEDURE InitFile;
(* Opens file and initializes the data *)
VAR
  i : CARDINAL;
BEGIN
  FileH := FIO.Create(FileName);
  FOR i := 1 TO 20 DO
    GenRec;
    WriteRec(FALSE);
  END;
END InitFile;
VAR
  i : LONGCARD;
BEGIN
  Lib.RANDOMIZE;
  (* Init random number generator *)
  InitFile;
  SeekRec(1);
  (* Go to top Record *)
  FOR i := 1 TO 20 DO
    (* Display all records *)
    SeekRec(i);
    ReadRec(TRUE);
    DisplayRec;
  END;
  SeekRec(1);
  FOR i := 1 TO 20 DO
    (* Update all Age *)
    ReadRec(TRUE);
    INC(CurrR.Age);
    WriteRec(FALSE);
  END;
  FOR i := 2 TO 20 BY 2 DO
    SeekRec(i);
    ReadRec(TRUE);
    DisplayRec;
  END;
  FOR i := 19 TO 1 BY -2 DO
    SeekRec(i);
    ReadRec(TRUE);
    DisplayRec;
  END;
END RandDemo.

```

Accessing the Printer using FIO

The standard file handle `PrinterDevice` is available to every program using `FIO`. This handle is opened at the start of the program and requires no initialization.

If you want to print data from your program on a printer attached to the computer, you simply use the appropriate `Wrtype` procedure from the `FIO` module. For example:

```

FIO.WrStr(PrinterDevice,'Hello, world');
FIO.WrLn(PrinterDevice);
FIO.WrStr(PrinterDevice,'There are ');
FIO.WrCard(PrinterDevice,5,1);
FIO.WrStr(PrinterDevice,' lines in this example.');
```

This will print the following on the printer:

```
Hello, world
There are 5 lines in this example.
```

FIO Reference

The following pages describe the individual procedures defined within the FIO module.

Append — Open existing file at end

PROCEDURE Append(
 Name: ARRAY OF CHAR): File;

The Append procedure is used to open a file like Open but the file is positioned at the end so that any new data is added to the file and any data already in the file is preserved. The file must exist for this procedure to function correctly. For example:

```
IMPORT FIO;
VAR
  PersonnelFile : FIO.File;
BEGIN
  ...
  PersonnelFile := FIO.Append('PERSON.DAT');
  FIO.WrStr(PersonnelFile, 'This is some new data');
  ...
```

This will open the file PERSON.DAT, if it exists and write the string 'This is some new data' at the end of the file. You can also read from the file by using the Seek procedure to change the file position.

Under DOS 3.0 and higher, with SHARE installed, or under OS/2, the variable ShareMode may be set. This specifies the file sharing mode:

ShareNoInherit	File is private to process.
ShareCompat	Compatibility mode.
ShareDenyRW	Deny read and write.
ShareDenyWR	Deny write.
ShareDenyRD	Deny read.
ShareDenyNone	Deny none.

The default value of ShareMode is ShareCompat.

AppendHandle — Add a handle to the file system

```
PROCEDURE AppendHandle(
    F: File;
    ReadOnly: BOOLEAN);
```

This procedure appends file handle *F*, obtained directly from an operating system call, to file system. This must be done to ensure that the FIO module functions recognize the handle.

ReadOnly should be TRUE if the file was opened with read only attribute.

AppendStream — Get file handle from stream

```
PROCEDURE AppendStream( St: FileInf ): File;
```

This procedure converts a C stream descriptor into a Modula-2 File. The returned value can then be used with the other procedures in this module. C stream handles are returned by C library functions such as *fopen*. The *GetStreamPointer* procedure performs the opposite conversion.

AssignBuffer — Assign a buffer to an open file

```
PROCEDURE AssignBuffer(
    F: File;
    VAR Buf : ARRAY OF BYTE);
```

This procedure assigns a buffer to an open file. For the most efficient use of file buffering the buffer size should be calculated as follows:

$$\text{BufferSize} = (N * 512) + \text{BufferOverhead}$$

where *N* is at least 2. The constant *BufferOverhead* is defined in FIO.DEF. For example:

```
IMPORT FIO;
CONST
  BufferSize = 1024 + FIO.BufferOverhead;
VAR
  FileBuffer : ARRAY [1..BufferSize] OF BYTE;
  PersonnelFile : FIO.File;
BEGIN
  ...
  PersonnelFile := FIO.Open('PERSON.DAT');
  FIO.AssignBuffer(PersonnelFile, FileBuffer);
  ...
```

This opens the file PERSON.DAT and assigns a buffer to it of *BufferSize* bytes. Note that you must open a file before assigning a buffer to it and that each buffered file must have a separate buffer.

ChDir — Change Directory

PROCEDURE ChDir(Name : ARRAY OF CHAR);

This procedure changes the current working directory, like the OS/2 CD command. Name specifies the name of the directory you wish to change to. If IOcheck is TRUE, then this procedure displays an error message if the specified directory does not exist.

If the specified path contains a drive name, that drive will become the current drive.

Close — Close an open file

PROCEDURE Close(F : File);

This procedure closes an open file making it no longer available for use. It is an error to attempt to use a file once it has been closed. For example:

```
FI0.Close(PersonnelFile);
```

This closes the file opened in the previous two examples.

If the file was buffered, it will be flushed.

Any files remaining open on program termination will automatically be closed.

Create — Create and open new file

PROCEDURE Create(Name: ARRAY OF CHAR): File;

The Create procedure is used to create and open new files. If the file already exists then all the data is deleted and you start with an empty file. For example:

```
IMPORT FIO;
VAR
  PersonnelFile : FIO.File;
BEGIN
  ...
  PersonnelFile := FIO.Create('PERSON.DAT');
  FIO.WrStr(PersonnelFile,
            'This is a new file');
  ...
```

This will create a new file called PERSON.DAT. If the file already exists, its contents will be deleted before the file is opened. You can then write new data to the file.

Under DOS 3.0 and higher, with SHARE installed, or under OS/2, the variable ShareMode may be set. This specifies the file sharing mode:

ShareNoInherit File is private to process.

ShareCompat Compatibility mode.

ShareDenyRW Deny read and write.

ShareDenyWR Deny write.

ShareDenyRD Deny read.

ShareDenyNone Deny none.

The default value of ShareMode is ShareCompat.

Erase — Delete a file

PROCEDURE Erase(Name : ARRAY OF CHAR);

The Erase procedure deletes a file from the disk. The file must not be currently open. For example:

```
FIO.Erase('PERSON.DAT');
```

This deletes the file PERSON.DAT.

GetDir — Get current working directory

```
PROCEDURE GetDir(  
    Drive: SHORTCARD;  
    VAR Name : ARRAY OF CHAR );
```

This procedure returns, in Name, the full pathname of the current working directory on the drive specified by Drive. Drive takes the value 0 for the default drive, 1 for drive A, 2 for drive B, etc. For example:

```
VAR  
    CurrDir : ARRAY [1..80] OF CHAR;  
...  
FIO.GetDir(3,CurrDir);
```

This returns the current directory for drive C in the character array CurrDir.

GetDrive — Get current drive

```
PROCEDURE GetDrive() : SHORTCARD;
```

This procedure returns a number indicating the current default drive. The value is 1 for drive A, 2 for Drive B, etc.

GetFileDate — Finds the file date

```
PROCEDURE GetFileDate( F : File ) : LONGCARD;
```

Returns the date, as a LONGCARD, of the file opened as handle F.

This function returns a value suitable for easy comparison. If you require the file date and time in a format suitable for outputting, use GetFileStamp.

GetFileStamp — Get file stamp

```
PROCEDURE GetFileStamp(  
    F: File;  
    VAR b: FileStamp): BOOLEAN;
```

This procedure determines the date and time information for file *f*. The result is stored in a record *b*, of type *FileStamp*.

```
TYPE  
    FileStamp = RECORD  
        Year, Month, Day : SHORTCARD;  
        Hour, Min, Sec   : SHORTCARD;  
    END;
```

The procedure returns TRUE if the operation is successful.

GetPos — Get current file position

```
PROCEDURE GetPos( F : File ) : LONGCARD;
```

The *GetPos* procedure returns a file's current file position. If the file is currently positioned at its start, then this function returns 0.

For an example of the use of *GetPos* see the *Truncate* procedure.

GetStreamPointer — Get stream buffer handle

```
PROCEDURE GetStreamPointer(F: File): FileInf;
```

This procedure is used to convert a Modula-2 file handle into a C stream descriptor. The returned value may then be passed to C functions such as *fgetc*. C stream handles may be converted into Modula-2 file handles with the *AppendStream* procedure.

IOresult — Determine result of last operation

PROCEDURE IOresult() : CARDINAL;

This procedure returns a value indicating the success or failure of an I/O operation. It can be called after most operations and will return 0 if all went smoothly. If an error occurs, it will return an error code. Note that the global variable IOcheck must be FALSE in order to use this procedure. For example:

```
IMPORT FIO, IO;
VAR
  f : File;
  ior : CARDINAL;
BEGIN
  ...
  FIO.IOcheck := FALSE;
  f := FIO.Open('C:\DATA\PEOPLE.DTA');
  ior := FIO.IOresult();
  IF (ior = 0) THEN
    IO.WrStr('File Open OK');
  ELSE
    IO.WrStr('Error opening file; Code = ');
    IO.WrCard(ior,4);
  END;
  IO.WrLn;
  ...
```

This example attempts to open a file and reports any error when doing so. Notice that the returned value of IOresult must be assigned to a variable if it is to be used later on.

MkDir — Make a new directory

PROCEDURE MkDir(Name: ARRAY OF CHAR);

This procedure creates a new directory in the same way as the MD command. Name specifies the name of the new directory. If IOcheck is TRUE, then the procedure displays an error message if the specified directory cannot be created.

Open — Open an existing file

PROCEDURE Open(Name: ARRAY OF CHAR) : File;

The Open procedure opens the file specified by the Name parameter and returns a new file handle for the open file. The named file must already exist. If the file does not exist, and IOcheck is FALSE then the Open procedure returns MAX(CARDINAL) instead of a valid file handle. If IOcheck is TRUE and the file does not exist, your program will terminate with an error message.

Once the file has been opened the file position is set to 0, i.e., the beginning of the file.

Any read operation on a file opened using this procedure will retrieve the data at the current file position (if any). Any write operation will overwrite the information at the current file position (if any). For example:

```
IMPORT FIO;
VAR
  PersonnelFile : FIO.File;
BEGIN
  ...
  PersonnelFile := FIO.Open('PERSON.DAT');
  ...
```

This will open the file PERSON.DAT (if it exists) and make it available using the file handle stored in PersonnelFile. You can then read data from the file or write to it. If you write to it then existing data will be overwritten, unless you use the Seek procedure to position the file at the end.

Under DOS 3.0 and higher, with SHARE installed, or under OS/2, the variable ShareMode may be set. This specifies the file sharing mode:

ShareNoInherit File is private to process.

ShareCompat Compatibility mode.

ShareDenyRW Deny read and write.

ShareDenyWR Deny write.

ShareDenyRD Deny read.

ShareDenyNone Deny none.

The default value of ShareMode is ShareCompat.

OpenRead — Open an existing file for Reading

PROCEDURE OpenRead(Name: ARRAY OF CHAR): File;

The OpenRead procedure, like Open, opens the file specified by the Name parameter and returns a new file handle for the open file. The named file must already exist. If the file does not exist, and IOcheck is FALSE then the OpenRead procedure returns MAX(CARDINAL) instead of a valid file handle. If IOcheck is TRUE and the file does not exist, your program will terminate with an error message.

Once the file has been opened the file position is set to 0, i.e., the beginning of the file. The file can only be read by your program, an I/O error will occur if you attempt to write to the file.

Under DOS 3.0 and higher, with SHARE installed, or under OS/2, the variable ShareMode may be set. This specifies the file sharing mode:

ShareNoInherit File is private to process.

ShareCompat Compatibility mode.

ShareDenyRW Deny read and write.

ShareDenyWR Deny write.

ShareDenyRD Deny read.

ShareDenyNone Deny none.

The default value of ShareMode is ShareCompat.

RdBin — Read binary data

PROCEDURE RdBin(

F: File;
VAR Buf: ARRAY OF BYTE;
Sz: CARDINAL): CARDINAL;

This procedure reads at most Sz bytes from the file F into the buffer Buf. It returns a value indicating how many bytes were actually read. For example:

```
VAR
  Data : ARRAY [0..127] OF BYTE;
  InFile : FIO.File;
  BytesRead : CARDINAL;
...
BytesRead := FIO.RdBin(InFile,Data,128);
```

This attempts to read 128 bytes from the file InFile into the buffer Data. If there were at least 128 bytes left in the file, then BytesRead would be set to 128, otherwise it will be set to a lower figure. This enables you to detect end-of-file.

RdItem — Read separated string from file

PROCEDURE RdItem(F: File; VAR S:
ARRAY OF CHAR);

This procedure uses the global set variable Separators to determine when to stop reading input. Unlike RdStr, the programmer has control over what characters will cause the procedure to stop reading the file. When one of the characters in Separators is encountered, reading the file ceases and S is zero terminated. If S is filled before this occurs then it is not zero terminated. For example:

Given the following input file opened with handle InFile:

Hello there, everyone

Then the following program fragment:

```
VAR
  s1, s2, s3 : ARRAY [0..40] OF CHAR;
  InFile : FIO.File;
...
FIO.RdItem(InFile,s1);
FIO.RdItem(InFile,s2);
FIO.RdItem(InFile,s3);
```

would, using the default value of Separators, set s1 to 'Hello', s2 to 'there,' and s3 to 'everyone'.

Rdsimpletype — Formatted input

```
PROCEDURE RdChar(F: File): CHAR;  
PROCEDURE RdBool(F: File): BOOLEAN;  
PROCEDURE RdShtInt(F: File): SHORTINT;  
PROCEDURE RdInt(F: File): INTEGER;  
PROCEDURE RdLngInt(F: File): LONGINT;  
PROCEDURE RdShtCard(F: File): SHORTCARD;  
PROCEDURE RdCard(F: File): CARDINAL;  
PROCEDURE RdLngCard(F: File): LONGCARD;  
PROCEDURE RdShtHex(F: File): SHORTCARD;  
PROCEDURE RdHex(F: File): CARDINAL;  
PROCEDURE RdLngHex(F: File): LONGCARD;  
PROCEDURE RdReal(F: File): REAL;  
PROCEDURE RdLngReal(F: File): LONGREAL;
```

These procedures all take a single parameter, a file handle *F* that specifies the file to be read. Each of the procedures return a value of a simple type as shown in the declarations above.

The file is read as a stream of text using *RdItem* by all the procedures except *RdChar*. Each character sequence in the file is delimited by characters from the global set variable *Separators*. When a character string has been read it is converted to the appropriate type by using the conversion procedures in the *Str* module.

The *RdHex*, *RdShtHex* and *RdLngHex* read numbers in hex format and return the appropriate *CARDINAL* value.

The *RdBool* procedure returns *TRUE* if the character sequence read is 'TRUE' (all capitals) and *FALSE* otherwise.

The syntax for *REAL* numbers can be found in Chapter 2.

The global variable *OK* is set *TRUE* if the read operation was successful, i.e., the input data was in the correct format.

Note: When creating multi-thread programs the function *ThreadOK* should be used.

RdStr — Read a string from a file

PROCEDURE RdStr(F: File; VAR S: ARRAY OF CHAR);

This procedure reads characters from the file specified by the handle F. The characters are stored in S until one of the following conditions is met:

- An end-of-file character (ASCII 26) is read. In this case, S is zero terminated.
- A line feed character (ASCII 10) is read. In this case S is likewise zero terminated.
- The string S is full, that is, $HIGH(S) + 1$ characters have been read without encountering one of the other two conditions. In this case, S is not zero terminated.

ReadFirstEntry — Begin directory scan

```
PROCEDURE ReadFirstEntry(
    DirName: ARRAY OF CHAR;
    Attr: FileAttr;
    VAR D: DirEntry): BOOLEAN;
```

This procedure is used in conjunction with ReadNextEntry to scan through a directory. Both ReadFirstEntry and ReadNextEntry use the following type definitions from FIO:

```
TYPE
    PathTail = ARRAY [0..12] OF CHAR;
    FileAttr = SET OF
        (readonly, hidden,
         system, volume,
         directory, archive );
    DirEntry = RECORD
        rsvd:
            ARRAY [0..20] OF SHORTCARD;
        (* reserved *)
        attr : FileAttr;
        (* Attributes *)
        time : CARDINAL;
        (* File time *)
        date : CARDINAL;
        (* File date *)
        size : LONGCARD;
        (* File size *)
        Name : PathTail;
        (* File name *)
    END;
```

ReadFirstEntry is used to find files that match the pattern specified in DirName. DirName contains a file path, possibly including the ‘?’ and ‘*’ wildcards.

The Attr parameter narrows the search to only look for entries in the specified directory with matching attributes. If Attr is empty (or just the readonly and archive bits are set) then the procedure will only consider normal files for matching with the pattern in DirName. If you want to take into account directories, hidden files or system files, then you will have to set the appropriate bits in Attr. If the volume attribute is set the only the volume label is returned.

The procedure returns TRUE if it finds at least one item which matches both the attributes and pattern string. In this case the parameter D will contain the file name of the first matching entry. Once D has been initialized in this way, you can use ReadNextEntry to find the next matching file. You must call ReadFirstEntry before calling ReadNextEntry.

If no match can be found, then ReadFirstEntry returns FALSE. For example:

```
(*
  This example also illustrates the use
  of ReadNextEntry
*)

IMPORT FIO, IO;

VAR
  Ent : FIO.DirEntry;
  Found : BOOLEAN;

BEGIN
  ...
  Found := FIO.ReadFirstEntry("c:\com\*.exe",
                             FIO.FileAttr(FIO.system, FIO.hidden, Ent ));
  WHILE Found DO
    IO.WrStr(Ent.Name);
    IO.WrLn;
    Found := ReadNextEntry(Ent);
  END;
  ...
```

This prints all the hidden, system file in the directory 'c:\com' within the extension '.exe'. Note that due to operating system limitations, you cannot have more than one ReadNextEntry and ReadFirstEntry pair active at one time.

ReadNextEntry — Find next matching file

```
PROCEDURE ReadNextEntry(  
    VAR D: DirEntry): BOOLEAN;
```

This procedure works together with ReadFirstEntry. It takes, as a parameter, a directory entry, D, which was initialized by a call to ReadFirstEntry or has been used by ReadNextEntry.

The procedure returns TRUE if further matches can be found and FALSE otherwise. See ReadFirstEntry for an example of its use.

Rename — Rename a file

```
PROCEDURE Rename(  
    CurrentName,  
    NewName: ARRAY OF CHAR );
```

This procedure renames the file CurrentName as NewName. For this function to be successful the file specified by CurrentName must not be open and there must not be another file in the same directory as CurrentName which already has the name NewName. For example:

```
FIO.Rename( 'PERSON.DAT', 'PEOPLE.DTA' );
```

This renames the file PERSON.DAT to PEOPLE.DAT.

RmDir — Remove a directory

```
PROCEDURE RmDir( Name : ARRAY OF CHAR );
```

This procedure deletes a directory, like the rm command. Name specifies the name of the directory you wish to delete. It is an error to attempt to delete a directory which still contains files or sub-directories. It is also an error to attempt to delete the current working directory, even if it is empty. If such an error occurs and IOcheck is TRUE, then this procedure prints an error message.

Seek — Set new file position

PROCEDURE Seek(F : File; Pos : LONGCARD);

The Seek procedure repositions a file so that the next I/O operation will occur at a different place in the file.

For an example of the use of Seek see the Truncate procedure.

SetDrive — Set default drive

PROCEDURE SetDrive(
Drive: SHORTCARD): SHORTCARD;

This procedure sets the current default drive using 1 for drive A, 2 for drive B, etc. It returns the total number of drives available on the system.

SetFileDate — Change a file's date

PROCEDURE SetFileDate(F: File; D: LONGCARD);

This procedure sets a new date for the file opened on handle F.

Size — Determine the size of a file

PROCEDURE Size(F : File) : LONGCARD;

The Size procedure returns the current size of a file. Note that it does not affect the file position and that you use a file handle not a file name. This means that the file must be open before you use Size to determine the current size of a file.

For an example of the use of Size see the Truncate procedure.

ThreadEOF — Get thread's EOF status

PROCEDURE ThreadEOF(): BOOLEAN;

In a multi-thread program, the variable EOF may be overwritten between being set and then read by a particular thread. The only reliable way to inspect the result of a procedure or function that sets EOF is to use the return value of ThreadEOF.

ThreadOK — Get thread's OK status

PROCEDURE ThreadOK(): BOOLEAN;

In a multi-thread program, the variable OK may be overwritten between being set and read by a particular thread. The only reliable way to inspect the result of a procedure or function that sets OK is to use the return value of ThreadOK.

Truncate — Truncate a file

PROCEDURE Truncate(F : File);

The Truncate procedure shortens a file. The new file size is determined by the current file position. Thus, if the current file position is byte 12346 and Truncate is called, the new file will be only 12346 bytes long and all the data after this point will have been lost. For example:

```
(* This example truncates a file to half its current size *)
IMPORT FIO;
CONST
  FileName = 'PERSON.DAT';
VAR
  CurrentSize : LONGCARD;
  Fhandle : FIO.File;
BEGIN
  ...
  Fhandle := FIO.Open(FileName);
  CurrentSize := FIO.Size(Fhandle);
  FIO.Seek(Fhandle,CurrentSize DIV 2);
  FIO.Truncate(Fhandle);
  FIO.Close(Fhandle);
  ...
```

This example also demonstrates the use of GetPos, Size and Seek.

Firstly the required file is opened and its size found using the Size procedure. Then Seek is used to position the file half-way along its length. The Truncate procedure is then called. Note that the file handle is used in Truncate and not the file name. Finally the file is closed.

WrBin — Write binary data

```
PROCEDURE WrBin(
    F: File;
    Buf: ARRAY OF BYTE;
    Sz : CARDINAL );
```

This procedure writes a block of “raw”, unformatted, binary data to the file F. The data is specified by Buf and Sz indicates how many bytes to write to the file. For example:

```
VAR
    Data : ARRAY [0..63] OF BYTE;
    Out  : FIO.File;
...
FIO.WrBin(Out,Data,64);
```

This would write 64 bytes of binary data from the buffer Data to the file Out.

WrCharRep — Write character repeatedly

```
PROCEDURE WrCharRep(
    F: File;
    V: CHAR;
    N: CARDINAL );
```

This procedure writes the character V on file F, N times. For example:

```
FIO.WrCharRep(StandardOutput,'*',20);
```

This would produce the following output:

```
*****
```

WrLn — Start a new line

```
PROCEDURE WrLn( F : File );
```

This procedure writes a newline sequence (carriage return plus line-feed) sequence to the file. For example:

```
FIO.WrLn(PrinterDevice);
```

Wrsimpletype — Formatted output

```
PROCEDURE WrChar(F: File; V: CHAR);
PROCEDURE WrBool(
    F: File;
    V: BOOLEAN;
    L: INTEGER);
PROCEDURE WrShtInt(
    F: File;
    V: SHORTINT;
    L: INTEGER);
PROCEDURE WrInt(
    F: File;
    V: INTEGER;
    L: INTEGER);
PROCEDURE WrLngInt(
    F: File;
    V: LONGINT;
    L: INTEGER);
PROCEDURE WrShtCard(
    F: File;
    V: SHORTCARD;
    L: INTEGER );
PROCEDURE WrCard(
    F: File;
    V: CARDINAL;
    L: INTEGER );
PROCEDURE WrLngCard(
    F: File;
    V: LONGCARD;
    L: INTEGER );
PROCEDURE WrShtHex(
    F: File;
    V: SHORTCARD;
    L : INTEGER );
PROCEDURE WrHex(
    F: File;
    V: CARDINAL;
    L: INTEGER );
PROCEDURE WrLngHex(
    F: File;
    V: LONGCARD;
    L: INTEGER );
PROCEDURE WrReal(
    F: File;
    V: REAL;
    P: CARDINAL;
    L : INTEGER );
PROCEDURE WrLngReal(
    F: File;
    V: LONGREAL;
    P: CARDINAL;
    L: INTEGER);
PROCEDURE WrFixReal(
```

```

                                F: File;
                                V: REAL;
                                P: CARDINAL;
                                L: INTEGER);
PROCEDURE WrFixLngReal(
                                F: File;
                                V: LONGREAL;
                                P: CARDINAL;
                                L: INTEGER);

```

All these procedures (except WrChar) call WrStrAdj. The Wrsimpletype procedures (except WrChar) take the following parameters:

F	a file handle
V	a value of the type to be written
L	a field width

The value V is written in a field of width ABS(L) on file F. If L is negative then the formatted data will be left adjusted, otherwise it is right adjusted. All the procedures (except WrChar and WrBool) call the appropriate conversion routine from the Str module to obtain a string representation of V.

In addition the WrReal and WrLngReal take an extra parameter P which indicates the precision of the value to be written. The meaning of this is identical to the Precision parameter for RealToStr.

WrFixReal and WrFixLngReal write their output in a fixed point format. See FixRealToStr. For example:

```

VAR
  Out : FIO.File;
FIO.WrChar(Out,'a');
FIO.WrLn(Out);
FIO.WrBool(Out,TRUE,-15);
FIO.WrLn(Out);
FIO.WrBool(Out,TRUE,15);
FIO.WrLn(Out);
FIO.WrShtInt(Out,7,-15);
FIO.WrInt(Out,27,-15);
FIO.WrLngInt(Out,2777777,-15);
FIO.WrLn(Out);
FIO.WrShtInt(Out,7,15);
FIO.WrInt(Out,27,15);
FIO.WrLngInt(Out,2777777,15);
FIO.WrLn(Out);
FIO.WrShtCard(Out,35,-15);
FIO.WrCard(Out,3555,-15);
FIO.WrLngCard(Out,355555,-15);
FIO.WrLn(Out);
FIO.WrShtCard(Out,35,15);
FIO.WrCard(Out,3555,15);

FIO.WrLngCard(Out,355555,15);
FIO.WrLn(Out);

```

```

FIO.WrShtHex(Out,39,-15);
FIO.WrHex(Out,3999,-15);
FIO.WrLngHex(Out,399999,-15);
FIO.WrLn(Out);
FIO.WrShtHex(Out,39,15);
FIO.WrHex(Out,3999,15);
FIO.WrLngHex(Out,399999,15);
FIO.WrLn(Out);
FIO.WrReal(Out,35.5555,5,-15);
FIO.WrLngReal(Out,356789.55556789,5,-15);
FIO.WrLn(Out);
FIO.WrReal(Out,35.5555,5,15);
FIO.WrLngReal(Out,356789.55556789,5,15);
FIO.WrLn(Out);
(* Contents of the file associated with *)
(* the file handle Out: *)
(*          1          3          *)
(*-----5-----0-----*)
a
TRUE
      TRUE
7          27          2777777
          7          27          2777777
35          3555          355555
          35          3555          355555
27          F9F          61A7F
          27          F9F          61A7F
3.5556E+1          3.5679E+5
          3.5556E+1          3.5679E+5

```

WrStr — Write String to file

PROCEDURE WrStr(F : File; V : ARRAY OF CHAR);

Writes the string V to the file associated with the handle F. For example:

```
FIO.WrStr(PersonnelFile,"Smith, John");
```

Writes the string "Smith, John" to PersonnelFile.

WrStrAdj — Write formatted string

PROCEDURE WrStrAdj(

F: File;

S: ARRAY OF CHAR;

L: INTEGER);

WrStrAdj writes the string S to the file F using ABS(L) as the field width. If S is longer than ABS(L) and the global variable ChopOff is TRUE, then a string of length ABS(L) containing only '?' is written instead of S. If ChopOff is FALSE, then the entire string is written. If L is negative then the string will be left adjusted, otherwise it will be right adjusted. For example:

```
FIO.WrStrAdj(StandardOutput,'Rover',10);
```

```
FIO.WrLn(StandardOutput);
```

```
FIO.WrLn(StandardOutput,'Rover',-10);
```

This will produce the following output:

```
      Rover  
Rover
```

MODULE FIOR

Introduction

The FIOR module is an extension to FIO which provides facilities for file management using the TopSpeed redirection system as used by the TopSpeed Environment. If you use this module then your program will initially attempt to load the file TS.RED as the redirection file. It will search for this in, firstly, the current working directory and, if not found there, then in the directory where the program is actually located.

Alternatively, if the environment variable TSRED is initialized, its contents will be used as the default redirection file name.

You can change the redirection file with the ReadRedirectionFile procedure. Note that this routine reads the redirection file into memory and then forgets about the file. If the redirection file is changed it must be explicitly read back into memory using the ReadRedirectionFile procedure. You should further note that redirection will only be applied to filenames; if any directory information is included then no redirection will be applied (i.e., redirection will be applied to 'ORDERS.DAT', but not to 'DATA\ORDERS.DAT').

The FIOR.DEF file redeclares the type File (for file handles) as:

```
TYPE  
  File : FIO.File;
```

This makes it identical with the FIO File type and means that it can be used with the file manipulation procedures of FIO.

You should also note that the IOresult procedure of this module is identical in behavior to the IOresult procedure in FIO except that it only applies to the FIOR procedures. In order to verify the success (or otherwise) of procedures in FIOR, you must use FIOR.IOresult. Note that you still use the FIO variable IOcheck to specify the action to be taken on an error.

FIOR Reference

The following are the individual procedures defined within the FIOR module.

AddExtension — Manipulate extensions

ChangeExtension

RemoveExtension

```

PROCEDURE AddExtension(
    VAR s: ARRAY OF CHAR;
    ext: ExtStr );
PROCEDURE ChangeExtension(
    VAR s: ARRAY OF CHAR;
    ext: ExtStr);
PROCEDURE RemoveExtension(
    VAR s: ARRAY OF CHAR);

```

These procedures allow you to manipulate the extension part of a filename. The filename (s in the procedure declarations above) can consist of a full pathname or just a simple name and extension.

The type ExtStr is defined in FIOR.DEF as:

```

TYPE
    ExtStr = ARRAY [0..2] OF CHAR;

```

The procedure AddExtension adds an extension to a filename without an extension. If s already has an extension (even just a single '.'), then nothing happens.

The procedure ChangeExtension changes the extension on a filename. It functions exactly like a RemoveExtension followed by an AddExtension. Thus, it will add an extension to a filename that doesn't have one as well as altering one that does.

The procedure RemoveExtension removes the extension from a filename. If s has no extension, nothing happens. For example:

```

VAR
    fn1, fn2, fn3 : ARRAY [0..40] OF CHAR;
...
fn1 := "FILENAME";
fn2 := "FILENAME.EXT";
fn3 := "FILENAME.DAT";
FIOR.AddExtension(fn1,"XXX");      (* fn1 = "FILENAME.XXX" *)
FIOR.RemoveExtension(fn2);        (* fn2 = "FILENAME"      *)
FIOR.ChangeExtension(fn3,"DAT");
    (* fn3 = "FILENAME.DAT" *)
...

```

Create — Create file using redirection

PROCEDURE Create(name: ARRAY OF CHAR): File;

This procedure is identical in behavior to the Create function in FIO. It opens a new file of the name specified by name. It is not an error for the file to already exist, but if it does, its contents are lost.

The difference from FIO.Create is that this procedure uses the currently loaded redirection file to determine where to create the file. FIO.Create always creates a file in the current working directory unless a full pathname is given. For example:

```
(*
  Assuming that the current redirection
  file contains a line like this:

  *.dat = x:\user; c:\data
*)
VAR
  f: FIOR.File;
BEGIN
  ...

  f := FIOR.Create("TEST.DAT");
  ...

  (*
    This would create a new file TEST.DAT
    in the directory x:\user
  *)
```

Erase — Delete file using redirection

PROCEDURE Erase(name : ARRAY OF CHAR);

This procedure is identical to the Erase procedure in the FIO module except that the redirection file is used to locate the file. This procedure deletes the named file.

ExpandPath — Expand path using redirection file

PROCEDURE ExpandPath(
 path: ARRAY OF CHAR;
 VAR fullpath: ARRAY OF CHAR);

ExpandPath takes a partial path such as 'src' and expands it to its full representation, such as 'c:\ts\src'. It can cope with references to both the current directory ('.') and the parent directory ('..').

FindNewPath — Find path for file using redirection

FindPath

```

PROCEDURE FindNewPath(
  name : ARRAY OF CHAR;
      VAR fullname : ARRAY OF CHAR )
  :BOOLEAN;
PROCEDURE FindPath(
  name : ARRAY OF CHAR;
      VAR fullname : ARRAY OF CHAR )
  :BOOLEAN;

```

These two procedures expand a partial filename (name) into a full path string, fullname, following the rules specified in the current redirection file.

FindNewPath finds the path string for a new file (i.e., one that does not already exist).

FindPath finds the path string for an existing file. For example:

```

(* Assuming the redirection file contains:
  *.dat = c:\data; d:\data; x:\share\data
  *.dta = c:\private; x:\private\suzan
  and that the file 'person.dat' is in d:\data
*)
...
VAR
  fp1, fp2 : ARRAY [0..63] OF CHAR;
...
FIOR.FindPath("person.dat",fp1);
      (* fp1 = "D:\DATA\PERSON.DAT" *)
FIOR.FindNewPath("people.dta",fp2);
      (* fp2 = "C:\PRIVATE\PEOPLE.DTA" *)

```

IOresult — Determine error code

```

PROCEDURE IOresult() : CARDINAL;

```

IOresult reports the success or failure of the previous file operation. It returns zero if the operation succeeded.

IsExtension — Check file extension

```
PROCEDURE IsExtension(
    s: ARRAY OF CHAR;
    ext: ExtStr): BOOLEAN;
```

IsExtension returns TRUE if the file in s has the extension given in ext, otherwise it returns FALSE. See AddExtension for a definition of ExtStr. For example:

```
IsMOD := IsExtension("E:\TS\SRC\DATABASE.MOD", "MOD");
(* IsMOD is set TRUE *)
```

MakePath — Path and file manipulation

SplitPath

```
PROCEDURE MakePath(
    VAR path: ARRAY OF CHAR;
    head, tail: ARRAY OF CHAR );

PROCEDURE SplitPath(
    path: ARRAY OF CHAR;
    VAR      head, tail: ARRAY OF CHAR);
```

These two procedures join and split the components of a path, respectively. MakePath takes an existing path string (for example, 'c:\ts') as its head and a directory name (for example, 'src') and joins them together (to make, in our example, 'c:\ts\src') in the variable path. SplitPath does the opposite; it takes a full path string in path (for example 'c:\com\ext') and splits it into head and tail (in this example, 'c:\com\' and 'ext', respectively).

Open — Open file using redirection

```
PROCEDURE Open( name : ARRAY OF CHAR ) : File;
```

The Open procedure is used to open a file in read-only mode. In this way it is identical to the Open procedure in the FIO module. This version of Open, however, uses the redirection file to determine the full path string of the file. The FIO version of Open only looks in the current directory.

ReadRedirectionFile — Establish new redirection file

```
PROCEDURE ReadRedirectionFile(  
    name: ARRAY OF CHAR);
```

This procedure reads the redirection file specified by name into memory and uses the redirection instructions it contains as the basis for all future redirections. This allows you to load a different set of redirection parameters for your application from the one used by the TopSpeed Environment.

MODULE FloatExc

Introduction

This module enables floating-point exception handling. By default, if an 80x87 floating-point exception occurs, the program will be terminated, and the ERRORINF.??? file created. The contents of this file can be used to locate the exception. See Appendix A : *'Run-time Error Codes'*

A floating-point exception is caused by, for example, attempted division by zero, or a result that is larger than the floating point processor (or emulator) can cope with.

This module can be used to install a user defined handler to modify this default behavior.

FloatExc Reference

The following are the individual procedures defined within the FloatExc module.

DisableExceptionHandling — Disable Exception Handling

PROCEDURE DisableExceptionHandling;

This procedure disables 80x87 Exception Handling.

EnableExceptionHandling — Enable Exception Handling

PROCEDURE EnableExceptionHandling;

This procedure enables 80x87 exception handling

InstallHandler — Installs exception handler

```
PROCEDURE InstallHandler(H: FloatErrHandler);
```

The procedure installs an exception handler of the following type:

```
TYPE  
  FloatErrHandler = PROCEDURE(  
    INTEGER, INTEGER);
```

When an exception occurs, the handler will be called with two arguments.

The first is the signal type which is always 8. This indicates that a floating point exception has occurred.

The second is the exception type, which will be one of the following values:

FPE_INVALID Invalid operation.

FPE_DENORMAL Denormal operand

FPE_ZERODIVIDE Divide by zero.

FPE_OVERFLOW Overflow.

FPE_UNDERFLOW Underflow.

FPE_INEXACT Loss of precision

Note: The exception handling mechanism always resets itself to its default action. Therefore, a handler should reinstall itself if a further exception is to be handled.

MODULE *FormIO*

Introduction

This module provides procedures for outputting numbers of arguments to the console using module IO. The format and type of the arguments is controlled by a format string of the following form:

```

FormatString ::= { Alpha | FieldSpecifier | SwitchChar }
Alpha ::= any ascii char except '%' and '\'
FieldSpecifier ::=
    '%' '%' |
    '%' ['-'] [WidthSpecifier] TypeSpecifier
WidthSpecifier ::= DecimalNumber [ '.' DecimalNumber ]
TypeSpecifier ::=
    'u' (* Unsigned *) |
    'i' (* Signed *) |
    'r' (* Real *) |
    'C' (* Character *) |
    'S' (* String *) |
    'h' (* Hex (unsigned) *) |
    'b' (* Boolean *) |
    'p' (* Pointer / Address *)
SwitchChar ::= '\' SwitchOptions
SwitchOptions ::=
    '\' (* \ *) |
    '%' (* % *) |
    'b' (* BS=CHR(8) *) |
    'f' (* FF=CHR(12)*) |
    'n' (* NL=CHR(13),CHR(10)*) |
    't' (* Tab = CHR(9)*) |
    'e' (* Esc = CHR(27)*) | CharCode
CharCode ::= DecimalNumber
DecimalNumber ::= Digit [ Digit [ Digit ] ]

```

All alphanumeric characters in the format string will be output. Any switch options will be interpreted and the appropriate character output. e.g.

```
FormIO.WrF('Hello World\n');
```

will produce the output 'Hello World' followed by a newline.

If a field specifier is encountered, the corresponding argument will be converted and output. e.g.

```

VAR
  C: CARDINAL;

BEGIN
  C := 4;
  WrF1('C is %u', C);

```

will produce the output 'C is 4'. As long as the correct type specifier is chosen, any size variable may be passed. For example, the 'u' specifier will accept SHORTCARD, CARDINAL and LONGCARD arguments.

An optional width specifier may be supplied. If the result of the conversion is less than the field width, the output will be padded, on the left, with spaces. If a '-' is specified before the field width, the output will be left justified. An optional precision may be specified after the field width. This determines the number of digits precision on real number conversions only.

Note: It is the programmer's responsibility to ensure that the actual arguments match those specified in the format string.

FormIO Reference

The following are the individual procedures defined within the FormIO module.

WrF — Output format string

```
PROCEDURE WrF(Pat: ARRAY OF CHAR);
```

The format string is output to the console.

WrF1 — Output format string and one argument

```
PROCEDURE WrF1(Pat: ARRAY OF CHAR;  
               P1: ARRAY OF BYTE);
```

The format string and one argument are output to the console.

WrF2 — Output format string and two arguments

```
PROCEDURE WrF2(Pat: ARRAY OF CHAR;  
               P1,P2: ARRAY OF BYTE);
```

The format string and two arguments are output to the console.

WrF3 — Output format string and three arguments

```
PROCEDURE WrF3(Pat: ARRAY OF CHAR;  
               P1,P2,P3: ARRAY OF BYTE);
```

The format string and three arguments are output to the console.

WrF4 — Output format string and four arguments

```
PROCEDURE WrF4(Pat: ARRAY OF CHAR;  
               P1,P2,P3,P4: ARRAY OF BYTE);
```

The format string and four arguments are output to the console.

WrF5 — Output format string and five arguments

```
PROCEDURE WrF5(Pat: ARRAY OF CHAR;  
               P1,P2,P3,P4,P5: ARRAY OF BYTE);
```

The format string and five arguments are output to the console.

MODULE Graph

Introduction

The Graph module implements a large number of procedures for using the graphics facilities of the various IBM and compatible screens.

There are a number of global variables and constants defined in the Graph.DEF file which are described under various headings below. Some are not supported under OS/2. Where this is the case a note has been included for reference.

Variables

Fill Mask

TYPE

FillMaskType = ARRAY [0..7] OF SHORTCARD;

The FillMaskType is used by SetFillMask and GetFillStyle to determine the pattern used to fill areas. It is regarded as an array of 8x8 bits. Where there is a 1 in the mask the pixel will be set on, otherwise it will be set off.

The pattern will be repeated over the area filled. For example:

```
CONST
BlockFill =
  FillMaskType(0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH);
(* All bits on *)
SpeckledFill =
  FillMaskType(0AAH,055H,0AAH,055H,0AAH,055H,0AAH,055H);
(* Alternate bits on:
  1 0 1 0 1 0 1 0 (= 0AAH)
  0 1 0 1 0 1 0 1 (= 055H)
  1 0 1 0 1 0 1 0 (= 0AAH)
  0 1 0 1 0 1 0 1 (= 055H)
  1 0 1 0 1 0 1 0 (= 0AAH)
  0 1 0 1 0 1 0 1 (= 055H)
  1 0 1 0 1 0 1 0 (= 0AAH)
  0 1 0 1 0 1 0 1 (= 055H)
  *)
```

Once a fill mask has been defined using SetFillMask, all the procedures able to fill shapes or draw filled shapes will use this value until a new one is set.

Line Style

All the graphics procedures that draw lines use the current line style to determine how lines will be drawn. This enables many types of line to be drawn, from solid lines to lightly dotted ones.

The current line style is set using the SetLineStyle procedure. This takes as a parameter a CARDINAL value regarded as a 16-bit word. These 16 bits define the line pattern to be used. The pattern is repeated along the length of the line taking a 1-bit to mean draw a pixel in the specified color and a 0-bit to mean use the background color. Thus a line style of 0FFFFH represents a solid line as all the bits are on and 0AAAAH represents a dotted line as the binary value is 1010101010101010 (alternate bits on and off).

You can use the GetLineStyle procedure to discover the current line style.

The VideoConfig Record

The VideoConfig record is used with the GetVideoConfig:

```

TYPE
VideoConfig = RECORD
  numxpixels   : CARDINAL;
  (* pixels on X axis *)
  numypixels   : CARDINAL;
  (* pixels on Y axis *)
  numtextcols  : CARDINAL;
  (* text columns available *)
  numtextrows  : CARDINAL;
  (* text rows available *)
  numcolors    : CARDINAL;
  (* actual colors *)
  bitsperpixel : CARDINAL;
  (* bits per pixel *)
  numvideopages: CARDINAL;
  (* available video pages *)
  mode         : CARDINAL;
  (* current video mode *)
  adapter      : CARDINAL;
  (* active display adapter *)
  monitor      : CARDINAL;
  (* active display monitor *)
  memory       : CARDINAL;
  (* video memory in K bytes *)
END;
```

Using this record enables you to determine the range of coordinates you can use with the graphics procedures. Graphics coordinates are defined as:

horizontal from 0 to (numxpixels - 1).

vertical from 0 to (numypixels - 1).

The origin is defined as (0,0). In the following descriptions, the notation (a,b) means a horizontal (x) coordinate of a and a vertical (y) coordinate of b. Graphics coordinates are used by all the drawing routines.

The text coordinates run from 1. The maximum row, or y, value is numtextrows and the maximum column, or x, value is numtextcols.

Supported Video Adapters and Screen Modes

Each video adapter supported is identified by a constant (set in the adapter field of VideoConfig):

```
CONST
_MDPA = 0001H;
(* Monochrome Display Adapter (MDPA) *)
_CGA  = 0002H;
(* Color Graphics Adapter (CGA) *)
_EGA  = 0004H;
(* Enhanced Graphics Adapter (EGA) *)
_VGA  = 0008H;
(* Video Graphics Array (VGA) *)
_MCGA = 0010H;
(* MultiColor Graphics Array (MCGA) *)
_HGC  = 0020H;
(* Hercules Graphics Card (HGC) *)
_MONO = 0001H;
(* Monochrome *)
_COLOR = 0002H;
(* Color (or Enhanced emulating) *)
_ENHCOLOR = 0004H;
(* Enhanced Color *)
_ANALOG = 0018H;
(* Analog *)
```

There are a number of video modes (set in the mode field) of VideoConfig associated with each adapter. These are defined by constants:

```

CONST
_DEFAULTMODE = MAX(CARDINAL);
(* original mode *)
_TEXTBW40    = 0;
(* 40 x 25 text, 16 grey *)
_TEXTC40     = 1;
(* 40 x 25 text, 16/8 color *)
_TEXTBW80    = 2;
(* 80 x 25 text, 16 grey *)
_TEXTC80     = 3;
(* 80 x 25 text, 16/8 color *)
_MRES4COLOR  = 4;
(* 320 x 200, 4 color *)
_MRESNOCOLOR = 5;
(* 320 x 200, 4 grey *)
_HRESBW     = 6;
(* 640 x 200, BW *)
_TEXTMONO    = 7;
(* 80 x 25 text, BW *)
_HERCMONO    = 8;
(* 720 x 348, BW for HGC *)
_MRES16COLOR = 13;
(* 320 x 200, 16 color *)
_HRES16COLOR = 14;
(* 640 x 200, 16 color *)
_ERESNOCOLOR = 15;
(* 640 x 350, BW *)
_ERESCOLOR  = 16;
(* 640 x 350, 4 or 16 color *)
_VRES2COLOR  = 17;
(* 640 x 480, BW *)
_VRES16COLOR = 18;
(* 640 x 480, 16 color *)
_MRES256COLOR = 19;
(* 320 x 200, 256 color *)

```

After graph mode has been initialized for an adapter (either explicitly or implicitly), the following variables are set to indicate the capabilities of the adapter:

```

VAR
Width    : CARDINAL;
(* x values are 0..Width-1 *)
Depth    : CARDINAL;
(* y values are 0..Depth-1 *)
NumColor : CARDINAL;
(* Colors are 0..NumColor-1 *)

```

Hercules Graphics

By default the graph module automatically detects the display hardware present. However some Hercules graphics cards are not possible to detect reliably. If you wish to use Hercules graphics call `InitHerc` before calling `SetVideoConfig`.

Text output in graphics mode is not available with Hercules cards.

Colors

Colors are specified in the Graph procedures using one of two methods:

- A **CARDINAL** value that specifies one of the colors from the current palette.
- A **LONGCARD** value that specifies the actual display color.

The current palette is defined using the **RemapPalette**, **RemapAllPalette** or **SelectPalette** procedures. The number of colors in a palette and their default values depends on the video adapter in use and the graphics mode used.

When using the **SelectPalette** procedure, the following palettes are available for the indicated display modes and adapters:

Adapter	Mode	Palette	Colors Available
_CGA	_MRES4COLOR	0	Blue, Red and Light gray
		1	Light blue, Light red and White
_EGA	_MRES4COLOR	0	Green, Red and Brown
		1	Cyan, Magenta and Light gray
		2	Light green, Light red and Yellow
		3	Light cyan, Light magenta and White
	_MRESNOCOLOR	0	Green, Red and Brown
		1	Light green, Light red and Yellow
		2	Light cyan, Light magenta and White
		3	Same as palette 0

The colors available correspond to color numbers 1, 2 and 3 (color 0 is always the background color). After the palette has been selected for these graphics modes (using the `SelectPalette` procedure), the color numbers (0 - 3) should be used as the color parameter for all the drawing procedures.

Thus, to draw a rectangle in Red, on a CGA adapter:

```
x:= Graph.SetVideoMode(_MRES4COLOR);
y:= Graph.SelectPalette(0);
(* Palette 0 *)
Graph.Rectangle(10,10,100,100,2,FALSE);
(* The color is 2 (Red) *)
```

When using the EGA or VGA adapters in 16 (or more) color mode (for example, graphics modes `_ERESCOLOR` and `_VRES16COLOR`) the palette is set using the `RemapPalette` and `RemapAllPalette` procedures. These procedures associate a `LONGCARD` value with each color number (the color numbers run from 0 to 15). The `LONGCARD` value represents three bytes giving the relative intensity of the blue, green and red components of the colors. This can be seen from the following constants, defined in `Graph.DEF`, that give color values for the more common screen colors:

```
_BLACK      = 0000000H;
_BLUE       = 02A0000H;
_GREEN      = 0002A00H;
_CYAN       = 02A2A00H;
_RED        = 000002AH;
_MAGENTA    = 02A002AH;
_BROWN      = 000152AH;
_WHITE      = 02A2A2AH;
_GRAY       = 0151515H;
_LIGHTBLUE  = 03F1515H;
_LIGHTGREEN = 0153F15H;
_LIGHTCYAN  = 03F3F15H;
_LIGHTRED   = 015153FH;
_LIGHTMAGENTA = 03F153FH;
_LIGHTYELLOW = 0153F3FH;
_BRIGHTWHITE = 03F3F3FH;
```

Thus `_BLACK` is 0 because this means no color components at all, `_BRIGHTWHITE` is 03F3F3FH as this means all the color components are at their highest (03FH) value and `_BLUE` just has the blue component switched on.

In the following descriptions, color number means a `CARDINAL` value between 0 and the maximum number of colors available for the video mode (3 for CGA, 15 for EGA and 15 or 255 for VGA). The term color value means the `LONGCARD` value specifying the relative values of the red, green and blue components of the color.

Graph Reference

The following are the individual procedures defined within the Graph module.

Arc — Draw an arc

```
PROCEDURE Arc(  
    x0, y0, a0, b0,  
    x1, y1, x2, y2: CARDINAL;  
    Color: CARDINAL );
```

Draws an arc in the color number specified by the Color parameter. The arc drawn is a fragment of an ellipse. The arc is specified using the parameters (x0,y0), a0, b0, (x1,y1) and (x2,y2) in the following manner:

(x0,y0)	Specifies the center of the ellipse that contains the arc.
a0	Specifies the length of the semi-major axis of the ellipse that contains the arc.
b0	Specifies the length of the semi-minor axis of the ellipse that contains the arc.
(x1,y1)	Specifies the co-ordinates for the starting point of the arc.
(x2,y2)	Specifies the co-ordinates for the ending point of the arc.

The arc is drawn from (x1,y1) counterclockwise to (x2,y2).

This procedure is not supported under OS/2.

Circle — Draw a circle

Disc

TrueCircle

TrueDisc

```
PROCEDURE Circle(  
    x0, y0, r: CARDINAL;  
    c: CARDINAL);  
  
PROCEDURE Disc(  
    x0, y0, r : CARDINAL;  
    FillColor: CARDINAL );  
  
PROCEDURE TrueCircle(  
    x0, y0, r: CARDINAL;  
    c: CARDINAL );  
  
PROCEDURE TrueDisc(  
    x0, y0, r : CARDINAL;  
    FillColor: CARDINAL );
```

These procedures draw circles. Circle and TrueCircle draw the outline of a circle whereas Disc and TrueDisc draw filled circles. Circle and Disc assume that the display is square, i.e. there are the same number of horizontal pixels as vertical pixels. Since this is not true for any of the adapters supported, all but the smallest circles will appear elliptical. The coordinates (x0,y0) specifies the center of the circle and r specifies its radius. The circle is drawn in the color number given by c and, for Disc and TrueDisc, filled with this color using the current fill mask (see SetFillMask).

Disc and Circle are faster than TrueDisc and TrueCircle as the latter procedures compensate for the aspect ratio of the screen by altering the horizontal and vertical radius so the shapes appear circular.

ClearScreen — Clear the screen

PROCEDURE ClearScreen(Area: CARDINAL);

This procedure clears all or part of the graphics screen in the current background color (see SetBkColor). The Area parameter specifies the area to be cleared and is one of the following values defined in Graph.DEF:

_GCLEARSCREEN clears the whole of the graphics screen.
 _GVIEWPORT clears the current viewport.
 _GWINDOW clears the current text window (see SetTextWindow).

This procedure is not supported under OS/2.

Cube — Draw a cube

PROCEDURE Cube(
 top: BOOLEAN;
 x1, y1,
 x2, y2, depth: CARDINAL;
 Color: CARDINAL;
 Fill: BOOLEAN);

This procedure draws a cube. The parameters have the following meaning:

top	If TRUE a top is drawn on the cube. Specifying FALSE allows several cubes to be stacked.
(x1,y1)	This pair specifies the coordinates of the upper left corner of the front face of the cube.
(x2,y2)	This pair specifies the coordinates of the lower right corner of the front face of the cube.
Depth	Specifies the two-dimensional distance to project the cube backwards, apparently “into the screen”.
Color	Specifies the color number for the cube to be drawn in.
Fill	If FALSE then only a skeleton cube is drawn, otherwise the cube is filled using the current fill mask (see SetFillMask). If Top is FALSE, the top is not filled.

This procedure is not supported under OS/2.

DisplayCursor — Show/Hide cursor

PROCEDURE DisplayCursor(
Toggle: BOOLEAN): BOOLEAN;

The DisplayCursor procedure determines whether or not the cursor will be turned back on when the program finishes a graphics routine (the cursor is always off while the graphics routines are drawing shapes). The cursor is automatically turned off when the program enters graphics mode. Toggle should be set TRUE to turn the cursor on.

The procedure returns the old state of the cursor (i.e., TRUE if the cursor is currently ON, FALSE otherwise).

This procedure is not supported under OS/2.

Ellipse — Draw an ellipse

PROCEDURE Ellipse(
x0, y0 : CARDINAL;
a0, b0 : CARDINAL;
Color : CARDINAL;
Fill : BOOLEAN);

Draws an ellipse in the color number specified by the Color parameter. The ellipse is specified using the parameters (x0,y0), a0 and b0 in the following manner:

(x0,y0)	Specifies the center of the ellipse.
a0	Specifies the length of the semi-major axis of the ellipse.
b0	Specifies the length of the semi-minor axis.
Color	Specifies the color number used to draw the ellipse.
Fill	If TRUE, the ellipse is filled with the current fill mask (see Graph.SetFillMask).

FloodFill — Fill region

StackFill

```
PROCEDURE FloodFill(  
    x, y: CARDINAL;  
    Color: CARDINAL;  
    Boundary: CARDINAL );  
  
PROCEDURE StackFill(  
    x, y: CARDINAL;  
    Color: CARDINAL;  
    Boundary: CARDINAL );
```

This pair of procedures can be used to fill regions on the screen, using the current fill mask (see SetFillMask). The parameters have the following meanings:

(x,y)	These are the coordinates of a seed-point within the area to be filled. This is simply a point not on the boundary of the shape that you want to be filled.
Color	This is the color number to be used.
Boundary	This specifies the color number of the boundary of the area to be filled. The procedures use this value to determine the edge of the shape.

The StackFill procedure is faster when the fill pattern is sparse (i.e., contains very few pixels) and when the shape to be filled is convex.

These procedures are not supported under OS/2.

GetBkColor — Get/Set current background color

SetBkColor

```
PROCEDURE GetBkColor() : LONGCARD;  
PROCEDURE SetBkColor(  
    Color: LONGCARD): LONGCARD;
```

The GetBkColor procedure returns the current background color, as a color value (not a palette color number). See ‘*Colors*’, above, for more information.

The SetBkColor procedure takes a color value, Color, as a parameter and sets a new background color. It returns the old background color value.

These procedures are not supported under OS/2.

GetFillMask — Get current fill mask

```
PROCEDURE GetFillMask(VAR Mask: FillMaskType);
```

The GetFillMask procedure sets the variable Mask to the current fill mask (see “Fill Mask”, above). This represents the bit pattern used to fill shapes using FloodFill, StackFill or when the Fill parameter of some procedures is set TRUE.

This procedure is not supported under OS/2.

GetImage — Get pixel area from screen

```
PROCEDURE GetImage(  
    x1, y1, x2, y2: CARDINAL;  
    Buffer: ADDRESS);
```

The GetImage procedure copies graphic information from the screen into a memory areas pointed to by Buffer. The area of the screen to be copied are given by the coordinates (x1,y1) (for the upper left of the area) and (x2,y2) (for the lower right of the area). The information copied is specific to the graphics mode being used and, therefore, an image cannot be redisplayed in any other mode (using PutImage).

The area of memory pointed to by Buffer must be large enough to hold the required information. Rather than provide a large number of formulas for calculating the size, the procedure ImageSize is provided that returns the size, in bytes, of the memory block needed to contain an image. The value returned can then be used with the ALLOCATE procedure (see the Lib module) to dynamically allocate the memory.

This procedure is not supported under OS/2.

GetLineStyle — Get current line style

```
PROCEDURE GetLineStyle() : CARDINAL;
```

This procedure returns the current line style (see '*Line Style*', above). The line style is a 16-bit value that is used when any of the graphics routines draws a line.

This is not supported under OS/2.

GetTextColor — Get current text color

```
PROCEDURE GetTextColor() : CARDINAL;
```

This procedure returns the current color number used for displaying text. Text is displayed on the graphics screen using this color irrespective of the drawing operations. The default value is the highest color number available for the current graphics mode.

This procedure is not supported under OS/2.

GetTextPosition — Get current text output position

PROCEDURE GetTextPosition() : TextCoords;

This procedure returns the next position at which text will be printed using the OutText procedure. The TextCoords record has the following format:

```
TYPE
  TextCoords = RECORD
    row, col: INTEGER;
  END;
```

The row field gives the row number (starting from 1) and the col field gives the column number (starting from 1). These coordinates are text coordinates and the upper value possible depends on the current video mode. You can use the GetVideoConfig record to determine these values.

This procedure is not supported under OS/2.

GetVideoConfig — Get video configuration

PROCEDURE GetVideoConfig(VAR V: VideoConfig);

This procedure returns details of the current video configuration in V. The VideoConfig record is defined above (see '*The VideoConfig Record*').

The VideoConfig record allows you to determine the size of the screen for both text and graphic operations, thus allowing you to adjust the display depending on the current video mode. In this way your programs can be made sensitive to the user's hardware configuration allowing you to scale images to fit the screen.

This procedure is not supported under OS/2.

GraphMode — Switch to graphics mode

PROCEDURE GraphMode;

This procedure switches the display adapter into graphics mode. It is used to switch back to graphics mode after a call to TextMode. You can also use the SetVideoMode procedure to switch between text and graphics mode. This latter method is recommended as it allows you to automatically return the screen to its original state.

HLine — Draw a horizontal line

```
PROCEDURE HLine(  
    x, y, x2: CARDINAL;  
    FillColor : CARDINAL );
```

This procedure draws a horizontal line on the screen in the color number FillColor. The line starts at coordinates (x,y) and extends to (x2,y). This procedure is used by the StackFill and FloodFill procedures.

ImageSize — Determine memory size of pixel area

```
PROCEDURE ImageSize(  
    x1, y1,  
    x2, y2: CARDINAL): LONGCARD;
```

In order to successfully use the GetImage procedure, you need to know the size of the memory block required to store the image. This procedure carries out the calculation and returns a value that can be used to allocate memory (using, perhaps, the ALLOCATE procedure in the Lib module).

The ImageSize procedure calculates the amount of memory required, in bytes, to store the image from (x1,y1) (the upper left of the image) to (x2,y2) (the lower right of the image).

This procedure is not supported under OS/2.

Initscreeentype — Initialization routines

```
PROCEDURE InitCGA();  
PROCEDURE InitEGA();  
PROCEDURE InitVGA();  
PROCEDURE InitHerc();  
PROCEDURE InitGraph();
```

These procedures (except for InitGraph) perform hardware specific initialization on the graphics adapter. The InitGraph procedure uses the appropriate initialization for the graphics adapter fitted to the machine.

If you use the SetVideoMode procedure, these procedures are unnecessary.

The procedures InitHerc and InitGraph are not supported under OS/2.

Line — Draw a line

```
PROCEDURE Line(  
                x1, y1, x2, y2: CARDINAL;  
                Color: CARDINAL );
```

The Line procedure draws a line, in the current line style, from (x1,y1) to (x2,y2) in the color number Color.

OutText — Output text to screen

```
PROCEDURE OutText( Text: ARRAY OF CHAR );
```

The OutText procedure is used to write text on the screen. The text is given by the parameter Text. The text is printed in the current text color (set using the SetTextColor procedure) at the current text position. The text position can be set using the SetTextPosition procedure. If a new position is not set, any new text is printed immediately after the last.

The GetTextPosition procedure enables you to determine the current text position.

This procedure is not supported under OS/2.

Pie — Draw a pie chart slice

```
PROCEDURE Pie( x0, y0, a0, b0,  
              x1, y1, x2, y2 : CARDINAL;  
              Color : CARDINAL;  
              Fill : BOOLEAN );
```

Draws an pie slice in the color number specified by the Color parameter. The slice drawn is a fragment of an ellipse. The pie is specified using the parameters (x0,y0), a0, b0, (x1,y1) and (x2,y2) in the following manner:

(x0,y0)	Specifies the center of the ellipse that contains pie slice.
a0	Specifies the length of the semi-major axis of the ellipse that contains the pie slice.
b0	Specifies the length of the semi-minor axis of the ellipse that contains the pie slice.
(x1,y1)	Specifies the co-ordinates for the starting point of the pie slice (curved) edge.
(x2,y2)	Specifies the co-ordinates for the ending point of the pie slice (curved) edge. The pie slice arc is drawn from (x1,y1) counterclockwise to (x2,y2).
Color	Specifies the color number used to draw the slice.
Fill	If TRUE the pie slice is filled, otherwise only the outline is drawn.

This procedure is not supported under OS/2.

Plot — Set a single pixel

```
PROCEDURE Plot( x, y: CARDINAL;  
               Color: CARDINAL);
```

The Plot procedure draws a single pixel in color number Color at position (x,y).

Point — Get the value of a single pixel

PROCEDURE Point(x, y : CARDINAL) : CARDINAL;

The Point procedure returns the color number of the point at (x,y).

Polygon — Draw a polygon

PROCEDURE Polygon(
 n: CARDINAL;
 px, py: ARRAY OF CARDINAL;
 FillColor: CARDINAL);

This procedure draws an n sided filled polygon in the color number given by FillColor. The arrays px and py give the points of the polygon. The px array contains the x-coordinates and the py array contains the y-coordinates. Each of these arrays must be defines as:

```
TYPE
  CoordArray = ARRAY [0..NoOfSides - 1] OF CARDINAL;
```

Each element of the array px is paired with the corresponding element of the py array to give the coordinates of one of the vertices of the polygon. For example:

```
TYPE
  CoordArray = ARRAY [0..2] OF CARDINAL;
```

```
VAR
  xcoord, ycoord : CoordArray;
...
xcoord := CoordArray(20,40,60);
ycoord := CoordArray(100,50,100);
Graph.Polygon(3,xcoord,ycoord,1);
```

```
(* Draws a triangle with points at:
```

```
      (40,50)
     /      \
    /          \
   /              \
  (20,100)-(60,100)
```

```
*)
```

PutImage — Put a pixel area on to the screen

PROCEDURE PutImage(

x, y: CARDINAL;
Buffer: ADDRESS;
Action: CARDINAL);

This procedure is used in conjunction with GetImage. It is used to display a previously saved image on the screen. The (x,y) parameters specify the coordinates of the upper left corner of the image. It is unnecessary to specify the other corner of the image as the data saved by GetImage includes the width and height of the image. The parameter Buffer must point to an area of memory where the image was previously stored using the GetImage procedure.

The Action parameter is used to produce different effects when displaying the image, it can have the following values (all defined as constants in the Graph.DEF file):

<code>_GPSET</code>	Copies the image from the buffer to the screen exactly as it was retrieved using GetImage. This is used to obtain an exact copy of one area in another part of the screen.
<code>_GPRESET</code>	Copies a negative or inverse form of the image onto the screen, replacing the existing image in the area. Each point in the image is copied with the inverse of the color number it was saved with.
<code>_GAND</code>	Overlays the image with any points already on the screen. Each pixel is logically ANDed with the pixel already on the screen. Portions of the screen may be erased as a result of this operation.
<code>_GOR</code>	Superimposes the image on whatever is already on the screen. The original contents of the area are not erased.
<code>_GXOR</code>	This action can be used in the two-color video modes to produce animation. When an image is _GXORed with itself, it disappears and can then be placed elsewhere on the screen using the _GOR or _GPSET actions.

This procedure is not supported under OS/2.

Rectangle — Draw rectangle

PROCEDURE Rectangle(

 x1, y1, x2, y2: CARDINAL;

 Color: CARDINAL;

 Fill: BOOLEAN);

Draws a rectangle. The rectangle is drawn from the upper left coordinates given by (x1,y1) to the lower right coordinates given by (x2,y2). The rectangle is drawn in color number Color in the current line style and, if Fill is TRUE it will be filled in the current fill mask pattern.

This procedure is not supported under OS/2.

RemapPalette — Specify new palette element for adapter

RemapAllPalette

```

PROCEDURE RemapPalette(
    Pixel: CARDINAL;
    Color: LONGCARD) : LONGCARD;

PROCEDURE RemapAllPalette(
    Colarray: ARRAY OF LONGCARD
) : CARDINAL;

```

These procedures require an EGA or VGA adapter and are used to alter the current palette. On a CGA adapter, the SelectPalette procedure should be used.

The palette can be thought of as an array of numcolors elements (where numcolors is a component of the VideoConfig record returned by GetVideoConfig). The adapter uses this array to map the color numbers onto color values. The default palette in 16 color modes is:

Number	Color	Number	Color
0	Black	8	Dark grey
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	Bright white

For modes supporting less than 16 colors, the defaults are still taken from this table with, for example, 4-color mode using color number 0 to 3.

The RemapAllPalette procedure takes an array of LONGCARD values specifying the color components of all the elements in the palette (see ‘Colors’, above, for information on how these color components are structured). The array must be numcolors in size. The procedure returns 0 on success and MAX(CARDINAL) in the event of an error. An error return will occur if you attempt to use the procedure on any adapter other than an EGA or VGA.

The RemapPalette procedure changes the display color value of color number Pixel to Color. It returns the old color value of Pixel in the palette, or MAX(LONGCARD) in the event of an error. The value of Pixel must be between 0 and (numcolors - 1).

These procedures are not supported under OS/2.

SelectPalette — Choose a new palette for adapter

PROCEDURE SelectPalette(Palnum : CARDINAL) : CARDINAL;

The SelectPalette procedures selects palettes for the CGA adapter or for the EGA adapter when emulating CGA. The value of palnum depends on the adapter and the video mode. The following table, as already given in “Colors” above, shows the valid palettes for the allowed combinations of adapters and modes:

Adapter	Mode	Palette	Colors Available
_CGA	_MRES4COLOR	0	Blue, Red and Light gray
		1	Light blue, Light red and White
_EGA	_MRES4COLOR	0	Green, Red and Brown
		1	Cyan, Magenta and Light gray
		2	Light green, Light red and Yellow
		3	Light cyan, Light magenta and White
	_MRESNOCOLOR	0	Green, Red and Brown
		1	Light green, Light red and Yellow
		2	Light cyan, Light magenta and White
		3	Same as palette 0

This procedure is not supported under OS/2.

SetActivePage — Switch video pages

SetVisualPage

```
PROCEDURE SetActivePage(
    Page: CARDINAL): CARDINAL;
PROCEDURE SetVisualPage(
    Page: CARDINAL): CARDINAL;
```

The EGA and VGA adapters can support up to 256Kb of memory to allow multiple pages of screen memory. This allows several different screen images to be stored, although only one will be shown on the monitor. CGA adapters can only support multiple pages in text mode. The numvideopages field of the VideoConfig record can be used to determine how many pages are available on a particular system.

These two procedures allow you to specify which page is displayed on the screen (SetVisualPage) and which page will be affected by graphics procedures (SetActivePage). When the active page and the visual page refer to the same page, the effect of all graphics commands are immediately visible to the user. When they are different, no changes take place until the active page is displayed.

This allows the programmer to carry out complex drawing operations “out of sight” and only display the result. This method can be used to produce far less messy animation sequences than constantly redrawing on one screen.

The basic technique is to draw on the, non-visible, active page, then make that page the display page and draw the next frame on another page: For example:

```
VAR
    ActivePage : CARDINAL;
    VisualPage : CARDINAL;
    FrameNo : CARDINAL;
    Dummy : CARDINAL;
...

ActivePage := 1;
VisualPage := 0;
FOR FrameNo := 1 TO MaxFrames DO
    Dummy := Graph.SetVisualPage(VisualPage);
    Dummy := Graph.SetActivePage(ActivePage);
    DrawFrame(FrameNo);
    IF (ActivePage = 1) THEN
        ActivePage := 0;
        VisualPage := 1;
    ELSE
        ActivePage := 1;
        VisualPage := 0;
    END;
END;
```

Both procedures take a single parameter, Page, with a value between 0 and (numvideopages - 1). They both return a value giving their previous value.

These procedures are not supported under OS/2.

SetClipRgn — Set clipping region

PROCEDURE SetClipRgn(x1, y1, x2, y2: CARDINAL);

This procedure allows you to set a clipping region. Once this region has been set, all graphics will be clipped, i.e., lines, fills, etc, that fall outside this area will not be displayed. Using this procedure does not affect graphics already drawn. The (x1,y1) parameters are the coordinates of the upper left corner of the clipping region and the (x2,y2) parameters are the coordinates of the lower right corner of the region. For example:

```
VAR
    Config : Graph.VideoConfig;
...
Graph.GetVideoConfig(Config);
Graph.SetClipRgn(10,10,100,100);
(* Set a clipping region *)
...
(* Draw in clipped region *)
...
WITH Config DO      (* Restore full screen *)
    Graph.SetClipRgn(0,0,numxpixels -
                      1,numypixels - 1);
END;
```

This procedure is not supported under OS/2.

SetFillMask — Set new fill mask

PROCEDURE SetFillMask(Mask: FillMaskType);

This procedure sets a new fill mask. The type FillMaskType is defined as:

```
TYPE
  FillMaskType = ARRAY [0..7] OF SHORTCARD;
```

The FillMaskType is used to determine the pattern used to fill areas. It is regarded as an array of 8x8 bits. Where there is a 1 in the mask the pixel will be set on, otherwise it will be set off. The pattern will be repeated over the area filled. For example:

```
CONST
  BlockFill =
    FillMaskType(0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH);
    (* All bits on *)
  SpeckledFill =
    FillMaskType(0AAH,055H,0AAH,055H,0AAH,055H,0AAH,055H);
    (* Alternate bits on:
      1 0 1 0 1 0 1 0 (= 0AAH)
      0 1 0 1 0 1 0 1 (= 055H)
      1 0 1 0 1 0 1 0 (= 0AAH)
      0 1 0 1 0 1 0 1 (= 055H)
      1 0 1 0 1 0 1 0 (= 0AAH)
      0 1 0 1 0 1 0 1 (= 055H)
      1 0 1 0 1 0 1 0 (= 0AAH)
      0 1 0 1 0 1 0 1 (= 055H)
    *)
```

Once a fill mask has been defined using SetFillMask, all the procedures able to fill shapes or draw filled shapes will use this value until a new one is set.

This procedure is not supported under OS/2.

SetLineStyle — Set new line style

PROCEDURE SetLineStyle(Mask: CARDINAL);

This procedure sets a new line style that will be used by all subsequent drawing commands.

All the graphics procedures that draw lines use the current line style to determine how lines will be drawn. This enables many types of line to be drawn, from solid lines to lightly dotted ones.

The current line style is set using the SetLineStyle procedure. This takes as a parameter a CARDINAL value regarded as a 16-bit word. These 16 bits define the line pattern to be used. The pattern is repeated along the length of the line taking a 1-bit to mean draw a pixel in the specified color and a 0-bit to mean use the background color. Thus a line style of 0FFFFH represents a solid line as all the bits are on and 0AAAAH represents a dotted line as the binary value is 10101010101010 (alternate bits on and off).

You can use the GetLineStyle procedure to discover the current line style.

This procedure is not supported under OS/2.

SetTextColor — Set new text color

PROCEDURE SetTextColor(
Color: CARDINAL): CARDINAL;

The SetTextColor procedure sets the color that will be used by OutText to display textual information in graphics mode. The default is the highest color number available for the current video adapter and mode. The Color parameter is the color number in the current palette that will be used for the text. The procedure returns the old text color number.

This procedure is not supported under OS/2.

SetTextPosition — Set new text output position

```
PROCEDURE SetTextPosition( row, col: CARDINAL
                          ): TextCoords;
```

The SetTextPosition procedure defines the screen position at which the next text output using OutText will occur. The row parameter indicates the screen line and col the screen column. The procedure returns a record of type TextCoords:

```
TYPE
  TextCoords = RECORD
    row, col: INTEGER;
  END;
```

The row field gives the row number (starting from 1) and the col field gives the column number (starting from 1). These are text coordinates; the upper value possible depends on the current video mode. You can use the GetVideoConfig record to determine these values.

This procedure is not supported under OS/2.

SetTextWindow — Set text window boundaries

```
PROCEDURE SetTextWindow( r1, c1,
                          r2, c2: CARDINAL );
```

The SetTextWindow procedure is used to define an area of the screen (a window) where text output will take place, the default is the whole screen. The coordinates (r1,c1) are the text coordinates of the upper left corner of the window (r1 is the row, c1 is the column) and (r2,c2) is the coordinates of the bottom right corner. See SetTextPosition and GetTextPosition for a description of text coordinates.

The Wrapon procedure is used to determine what happens when text extends beyond the edge of the window. If the wrapping is ON, then the text will be wrapped on to the next line. If it is OFF, the text will be clipped at the edge of the window.

This procedure is not supported under OS/2.

SetVideoMode — Set new video mode

PROCEDURE SetVideoMode(
 Mode: CARDINAL): BOOLEAN;

This procedure is used to change the video mode. It should be used in preference to the GraphMode and TextMode procedures. The Mode parameter should have one of the following values (depending on the adapter installed):

Mode Parameter	Description
_DEFAULTMODE	Original mode when program started
_TEXTBW40	40 x 25 text, 16 levels of grey
_TEXTC40	40 x 25 text, 16 or 8 colors
_TEXTBW80	80 x 25 text, 16 levels of grey
_TEXTC80	80 x 25 text, 16 or 8 color
_MRES4COLOR	320 x 200 graphics, 4 colors
_MRESNOCOLOR	320 x 200 graphics, 4 levels of grey
_HRESBW	640 x 200 graphics, monochrome
_TEXTMONO	80 x 25 text, monochrome
_HERCMONO	720 x 348 graphics, monochrome for Hercules
_MRES16COLOR	320 x 200 graphics, 16 colors
_HRES16COLOR	640 x 200 graphics, 16 colors
_ERESNOCOLOR	640 x 350 graphics, monochrome
_ERESCOLOR	640 x 350 graphics, 4 or 16 colors
_VRES2COLOR	640 x 480 graphics, monochrome
_VRES16COLOR	640 x 480 graphics, 16 colors
_MRES256COLOR	320 x 200 graphics, 256 colors

The procedure returns FALSE if the mode selected is not supported by the hardware.

SetVideoMode is not supported under OS/2.

TextMode — Switch to text mode

PROCEDURE TextMode;

This procedure switches the adapter into text mode. It is preferable to use the SetVideoMode procedure.

You can restore graphics mode with the GraphMode procedure.

Wrapon — Toggle text wrapping

```
PROCEDURE Wrapon( Opt : BOOLEAN ) : BOOLEAN;
```

This procedure affects the behavior of text that would extend beyond the boundaries of the current text window (see `SetTextWindow`). If `Opt` is `TRUE`, then text will be wrapped onto the next line of the window if it would extend over the current window's right-hand edge. If `Opt` is `FALSE`, the text will be clipped.

The procedure returns the previous setting.

Wrapon is not supported by OS/2.

MODULE IO

Introduction

The IO module provides a set of routines for providing input and output. As default these operations are assigned to the standard input (usually your keyboard) and the standard output (usually your screen). You can redefine these mediums and send input or/and output to files or other devices.

The IO Definition File

The definition file for the IO module contains a number of global variables, constants and types, as follows:

```
CONST
    MaxRdLength = 256;

TYPE
    WrStrType = PROCEDURE(ARRAY OF CHAR);
    RdStrType = PROCEDURE(VAR ARRAY OF CHAR);
    CHARSET = SET OF CHAR;

VAR
    RdLnOnWr: BOOLEAN;
    Prompt: BOOLEAN;
    InputRedirected: BOOLEAN;
    OutputRedirected: BOOLEAN;
    WrStrRedirect: WrStrType;
    RdStrRedirect: RdStrType;
    Separators: CHARSET;
    OK: BOOLEAN;
    ChopOff: BOOLEAN;
    Eng: BOOLEAN;
```

Redirecting Input and Output

One of the most powerful features of the IO module is the ability to be able to redirect the other input and output functions. This is accomplished by simply assigning a procedure to the two global variables `WrStrRedirect` and `RdStrRedirect`. These two procedures are called by all the I/O procedures in this module after conversion to a string (for output) or before any conversion takes place (for input). Thus the procedure `WrCard` first converts a `CARDINAL` to a string and then calls the procedure specified by `WrStrRedirect` to the actual output. As a default this is assigned to the normal text screen. For input, using, for example, `RdCard`, the procedure first calls `RdStrDirect` to obtain characters for input (by default, from the keyboard), before attempting any conversion.

As an example, let's see how to modify the behavior of the IO module to read and write from files instead of the screen:

```

MODULE IOdemo;
IMPORT IO, FIO, Str;
VAR
  InputFile,
  OutputFile: FIO.File;
  LineNo: CARDINAL;
  Line : ARRAY [0..IO.MaxRdLength] OF CHAR;
PROCEDURE MyInput(
  VAR InputString: ARRAY OF CHAR);
(*
  This does the input from a file,
  always dropping the first seven
  characters which are assumed to
  contain a line number and a space
*)
BEGIN
  FIO.RdStr(InputFile,InputString);
  IF FIO.EOF THEN
    InputString[0] := CHR(26);
    InputString[1] := CHR(0);
  ELSE
    Str.Delete(InputString,0,6);
  END;
END MyInput;
PROCEDURE MyOutput(
  OutputString: ARRAY OF CHAR);
(*
  This does the output to a file. It
  puts a six-digit line number and a?
  space before each line
*)
BEGIN
  FIO.WrCard(OutputFile,LineNo,-7);
  FIO.WrStr(OutputFile,OutputString);
  FIO.WrLn;
END MyOutput;
VAR
  Line
BEGIN
  IO.WrStr("Starting...");
  (* This goes on the screen *)
  IO.WrLn;
  InputFile := FIO.Open("INPUT.DAT");
  OutputFile := FIO.Create("OUTPUT.DAT");
  IO.RdStrRedirect := MyInput;
  IO.WrStrRedirect := MyOutput;
  IO.Prompt := FALSE;
  IO.RdLnOnWr := FALSE;
  IO.InputRedirected := TRUE;
  IO.OutputRedirected := TRUE;
  LineNo := 0;
  LOOP
    IO.RdStr(Line);
    IF (Line[0] = CHR(26)) THEN
      EXIT
    END;
    INC(LineNo,5);
    Str.Caps(Line);
    IO.WrStr(Line);
  END;
  FIO.Close(OutputFile);
  FIO.Close(InputFile);
END IOdemo;

```

This example uses procedures from the modules FIO and Str. You should consult the relevant module documentation in this section for further information on how these procedures work.

The constant MaxRdLength specifies the maximum length of line that IO can handle at once. The variable Prompt controls whether or not the IO module outputs a prompt (a '?') when it has no more characters in its line buffer.

If the variable RdLnOnWr is TRUE, then IO will flush its internal read buffer after every write. Clearly this is not satisfactory in the general case when building several lines of output from one line of input. You should set this FALSE to avoid any problems.

If you simply want to redirect the standard input or standard output, or both, to a file then the procedures RedirectInput and RedirectOutput can be used.

Given an input file (INPUT.DAT) like the following:

```
000012 This file is numbered
      10 but the lines don't fall into
000003 any particular order
```

The program above would produce the file OUTPUT.DAT containing the following:

```
5      THIS FILE IS NUMBERED
10     BUT THE LINES DON'T FALL INTO
15     ANY PARTICULAR ORDER
```

The other variables and types in the IO.DEF file are described in the explanations of the procedures that use them. Note that the procedure RdKey is not affected by redirection.

IO Reference

The following are the individual procedures defined within the IO module.

EndOfRd — Skip delimiters

```
PROCEDURE EndOfRd( Skip : BOOLEAN ) : BOOLEAN;
```

This procedure skips characters from the input buffer if Skip is TRUE. In particular, EndOfRd skips the characters that are members of the CHARSET Separators. The default value for Separators is:

```
CHARSET { CHR(9),      (* Tab *)
          CHR(10),     (* Line Feed *)
          CHR(13),     (* Carriage Return *)
          CHR(26),     (* End of file *)
          ' ',          (* Space *)
        }
```

EndOfRd returns TRUE if all the characters in the current line buffer have been read. It will do this even if SKIP is FALSE. Thus, this procedure can be used to determine if there any more characters on the current input line.

If there are still characters (other than Separators) on the current input line, then EndOfRd returns FALSE. For example:

Given the input string:

```
I said, "There was a young Vicar from Leeds"
```

And the following loop:

```
IO.RdItem(Item);
WHILE NOT IO.EndOfRd(TRUE) DO
  IO.WrStr(Item);
  IO.RdItem(Item);
END;
IO.WrLn;
```

Then the output would be (assuming the default Separators):

```
Isaid,"TherewasayoungVicarfromLeeds"
```

KeyPressed — Determine if a key has been pressed

```
PROCEDURE KeyPressed() : BOOLEAN;
```

This procedure determines if there any more characters waiting on the standard input. If this is so, then KeyPressed returns TRUE. Otherwise, it returns FALSE.

RdItem — Read delimited input string

```
PROCEDURE RdItem( VAR V : ARRAY OF CHAR );
```

This procedure uses the global set variable `Separators` to determine when to stop reading input. When one of the characters in `Separators` is encountered, reading the file ceases and `V` is zero terminated. If `V` is filled before this occurs then it is not zero terminated. The default value for `Separators` is:

```
CHARSET { CHR(9), (* Tab *)
          CHR(10), (* Line Feed *)
          CHR(13), (* Carriage Return *)
          CHR(26), (* End of file *)
          ' ', (* Space *)
        }
```

For example given the following input file opened with handle `InFile`:

```
Here comes everybody! "And, a hero called Earwicker."
```

Then the following program fragment:

```
VAR
  str : ARRAY [0..40] OF CHAR;
  i   : CARDINAL;
...
FOR i := 1 TO 8 DO
  IO.RdItem(str);
  IO.WrStr(str);
  IO.WrLn;
END;
```

would, using the default value of `Separators`, output:

```
Here
comes
everybody!
"And,
a
hero
called
Earwicker."
```

RdKey — Read a single character

PROCEDURE RdKey() : CHAR;

This procedure reads characters directly from the standard input. If there are no characters available then it waits for one.

When input is from the keyboard only, RdKey will return CHR(0) if any of the extended keys are pressed. The extended keys include the cursor keys and function keys. A second call to RdKey will, in this case only, return the extended keyboard code of the key. For example:

```
VAR
  ch : CHAR;

WHILE NOT IO.KeyPressed() DO
  END;
ch := IO.RdKey();
IF (ch = CHR(0)) THEN
  ch := IO.RdKey();
  IF (ch = CHR(59)) THEN
    IO.WrStr("Function Key 1 (F1) pressed");
  ELSE
    IO.WrStr("Some other extended key");
  END;
ELSIF (ch = CHR(59)) THEN
  IO.WrStr("The semi-colon (;) pressed");
ELSE
  IO.WrStr("Some other ordinary key was pressed");
END;
```

RdLn — Skip rest of input line

PROCEDURE RdLn;

This procedure skips the remainder of the current input line.

Rdsimpletype — Formatted input of simple types

```
PROCEDURE RdChar(): CHAR;  
PROCEDURE RdBool(): BOOLEAN;  
PROCEDURE RdShtInt(): SHORTINT;  
PROCEDURE RdInt(): INTEGER;  
PROCEDURE RdLngInt(): LONGINT;  
PROCEDURE RdShtCard(): SHORTCARD;  
PROCEDURE RdCard(): CARDINAL;  
PROCEDURE RdLngCard(): LONGCARD;  
PROCEDURE RdShtHex(): SHORTCARD;  
PROCEDURE RdHex(): CARDINAL;  
PROCEDURE RdLngHex(): LONGCARD;  
PROCEDURE RdReal(): REAL;  
PROCEDURE RdLngReal(): LONGREAL;
```

These procedures read from the standard input and return a value as indicated by the declarations above.

All the procedures (except `RdChar`) call the procedure `RdItem` to obtain a group of characters delimited by members of the set specified by `Separators` (see `RdItem` for a description of `Separators`). The string thus read is then converted using one of the conversion procedures in the `Str` module.

The procedures `RdShtHex`, `RdHex` and `RdLngHex` read `CARDINAL` values in hexadecimal format.

The procedure `RdBool` returns `TRUE` if the input string is `'TRUE'` (and it must be in upper case), otherwise it returns `FALSE`.

The global variable `OK` is set `TRUE` if no errors were detected in the conversion.

RdStr — Input string

```
PROCEDURE RdStr(VAR V : ARRAY OF CHAR);
```

`RdStr` reads a string from the standard input and returns it in `V`. Characters are read until either `V` is full or a carriage return (`CHR(13)`) is read. In the latter case only, `V` is zero terminated.

RedirectInput — I/O Redirection to file

RedirectOutput

```
PROCEDURE RedirectInput(
    FileName: ARRAY OF CHAR);
PROCEDURE RedirectOutput(
    FileName: ARRAY OF CHAR);
```

These two procedures change the standard input and standard output, respectively. Each closes the current stream and causes future input or output to be directed to the named file.

To direct the streams back to the console (screen or keyboard), the following calls should be used:

```
IO.RedirectInput("CON");
IO.RedirectOutput("CON");
```

Note: The standard input and output may be redirected from the command line using the redirection symbols, thus:

```
PROGRAM >out.fil <in.fil
```

Using these procedures will close files used like this.

ThreadOK — Get OK variable

```
PROCEDURE ThreadOK(): BOOLEAN;
```

In a multi-thread program, the variable OK may be overwritten between being set and read by a particular thread. The only reliable way to inspect the result of a procedure or function that sets OK is to use the return value of ThreadOK.

WrCharRep — Write repeated character

```
PROCEDURE WrCharRep(V: CHAR; Count: CARDINAL);
```

Outputs Count occurrences of the character V to the standard output. For example:

```
IO.WrCharRep('=',20);
(* produces the following output:
```

```
=====
*)
```

WrLn — Start new line

```
PROCEDURE WrLn;
```

Writes a new line sequence (CHR(10) followed by CHR(13)) to the standard output.

Wrsimpletype — Formatted output of simple types

```
PROCEDURE WrChar(V: CHAR );
PROCEDURE WrBool(
    V: BOOLEAN;
    L: INTEGER);
PROCEDURE WrShtInt(
    V: SHORTINT;
    L: INTEGER );
PROCEDURE WrInt(
    V: INTEGER;
    L: INTEGER );
PROCEDURE WrLngInt(
    V: LONGINT;
    L: INTEGER );
PROCEDURE WrShtCard(
    V: SHORTCARD;
    L: INTEGER );
PROCEDURE WrCard(
    V: CARDINAL;
    L: INTEGER );
PROCEDURE WrLngCard(
    V: LONGCARD;
    L: INTEGER );
PROCEDURE WrShtHex(
    V: SHORTCARD;
    L: INTEGER );
PROCEDURE WrHex(
    V: CARDINAL;
    L: INTEGER );
PROCEDURE WrLngHex(
    V: LONGCARD;
    L: INTEGER );
PROCEDURE WrReal(
    V: REAL;
    P: CARDINAL;
    L: INTEGER );
PROCEDURE WrLngReal(
    V: LONGREAL;
    P: CARDINAL;
    L: INTEGER );
PROCEDURE WrFixReal(
    V: REAL;
    P: CARDINAL;
    L: INTEGER);

PROCEDURE WrFixLngReal(
    V: LONGREAL;
    P: CARDINAL;
    L: INTEGER);
```

All these procedures (except WrChar) call WrStrAdj.

The Wrsimpletype procedures (except WrChar) take the following parameters:

V a value of the type to be written
L a field width

The value V is written in a field of width ABS(L) to the standard output. If L is negative then the formatted data will be left adjusted, otherwise it is right adjusted. All the procedures (except WrChar and WrBool) call the appropriate conversion routine from the Str module to obtain a string representation of V.

In addition the WrReal and WrLngReal take an extra parameter P which indicates the precision of the value to be written. The meaning of this is identical to the Precision parameter for RealToStr.

WrFixReal and WrFixLngReal write their output in a fixed point format. See FixRealToStr. For example:

```
IO.WrChar('a');
IO.WrLn;
IO.WrBool(TRUE,-15);
IO.WrLn;
IO.WrBool(TRUE,15);
IO.WrLn;

IO.WrShtInt(7,-15);
IO.WrInt(27,-15);
IO.WrLngInt(2777777,-15);
IO.WrLn;
IO.WrShtInt(7,15);
IO.WrInt(27,15);
IO.WrLngInt(2777777,15);
IO.WrLn;

IO.WrShtCard(35,-15);
IO.WrCard(3555,-15);
IO.WrLngCard(355555,-15);
IO.WrLn;
IO.WrShtCard(35,15);
IO.WrCard(3555,15);
IO.WrLngCard(355555,15);
IO.WrLn;
IO.WrShtHex(39,-15);
IO.WrHex(3999,-15);
IO.WrLngHex(399999,-15);
IO.WrLn;
IO.WrShtHex(39,15);
IO.WrHex(3999,15);
IO.WrLngHex(399999,15);
IO.WrLn;
IO.WrReal(35.5555,5,-15);
IO.WrLngReal(356789.55556789,5,-15); IO.WrLn;
IO.WrReal(35.5555,5,15);
IO.WrLngReal(356789.55556789,5,15); IO.WrLn;
```

```

(* Contents of the standard output *)
(*      1      3      *)
(*-----5-----0-----*)
a
TRUE
      TRUE
7      27      2777777
      7      27      2777777
35      3555      355555
      35      3555      355555
27      F9F      61A7F
      27      F9F      61A7F
3.5556E+1      3.5679E+5
      3.5556E+1      3.5679E+5

```

WrStr — Output string

```
PROCEDURE WrStr( S : ARRAY OF CHAR );
```

Writes the string S to the standard output.

WrStrAdj — Adjust length of string before output

```
PROCEDURE WrStrAdj(
    S: ARRAY OF CHAR;
    L: INTEGER );
```

WrStrAdj writes the string S to the file standard output using ABS(L) as the field width. If S is longer than ABS(L) and the global variable ChopOff is TRUE, then a string of length ABS(L) containing only '?' is written instead of S. If ChopOff is FALSE, then the entire string is written. If L is negative then the string will be left adjusted, otherwise it will be right adjusted. For example:

```

IO.WrStrAdj('Spot',10);
IO.WrLn;
IO.WrLn('Spot',-10);

```

This will produce the following output:

```

      Spot
Spot

```

MODULE Lib

Introduction

The Lib module contains a wide range of general purpose procedures for performing an assortment of tasks.

Because of its general purpose nature, the definition module contains a wide variety of definitions, most of which are applicable to only a few functions. For this reason, the definition file is not discussed here.

Run-time Error Support

When a Run-time error occurs in any of the library modules, a call is made to a Run-time error procedure. This procedure is the one currently associated with the procedure variable RunTimeError, defined in Lib.DEF.

The default Run-time error procedure produces the file ERRORINF.\$\$\$ and terminates the program. If you want to install your own Run-time error handler, simply assign the procedure name to the variable RunTimeError. See Appendix A : *'Run-time Error Codes'*.

Lib Reference

The following are the individual procedures defined within the IO module.

AddAddr — Address Arithmetic Procedures

SubAddr

IncAddr

DecAddr

```
PROCEDURE AddAddr(  
    A: ADDRESS;  
    increment: CARDINAL): ADDRESS;  
  
PROCEDURE SubAddr(  
    A: ADDRESS;  
    decrement: CARDINAL): ADDRESS;  
  
PROCEDURE IncAddr(  
    VAR A: ADDRESS;  
    increment: CARDINAL);  
  
PROCEDURE DecAddr(  
    VAR A: ADDRESS;  
    decrement: ADDRESS );
```

These four procedures perform address arithmetic for the 80x86 family of processors.

A is the address. increment is the value to add to A and decrement is the value to subtract from A.

SubAddr and AddAddr return the new address, leaving the value of A unaffected. IncAddr and DecAddr perform the arithmetic on the address, altering the value of A.

Under DOS, all four of these functions normalize the resultant address. A normalized address consists of an offset between 0 and 15, with the relevant segment adjusted accordingly. This means that addresses created using these routines may be safely compared for equality (using =, # and <>) and ordering (using >, >=, < and <=). For example:

```

VAR
  ShiftAddress : ADDRESS;
  Reference : LONGREAL;
...
ShiftAddr := AddAddr(ADR(Reference),4);
ShiftAddr := SubAddr(ADR(Reference),8);
IncAddr(ShiftAddr,16);
DecAddr(ShiftAddr,32);

```

If we assume that Reference is store at address 1000, decimal. Then ShiftAddr takes on the following values: 1004 ($1000 + 4$), 992 ($1000 - 8$), 1008 ($992 + 16$), 976 ($1008 - 32$).

Compare — Compare memory blocks

```

PROCEDURE Compare(
  Source, Dest: ADDRESS;
  Len: CARDINAL): CARDINAL;

```

Compares the two blocks of memory starting at Source and Dest, both of Len bytes. The procedure looks for the first byte where the two blocks differ and returns this as an offset from zero. The procedure will return Len if the two blocks are identical. For example:

```

Res := Lib.Compare(
  ADR("abcdefghi"),
  ADR("abcdeghu"),9);
(* Res = 5 *)

```

Cpuld — Determine CPU type

```
PROCEDURE CpuId(VAR r: CpuRec);
```

The CpuId procedure determines the type of the CPU on which the program is running and the type of the maths coprocessor, if any. The result is returned in r, which is defined as follows:

```
TYPE
  CpuKind = (cpu_Unknown, cpu_V20,
             cpu_V30, cpu_8088,
             cpu_8086, cpu_80188,
             cpu_80186, cpu_80286,
             cpu_80386 );
  FpuKind = (fpu_none, fpu_8087,
             fpu_80287, fpu_80387);
  CpuRec   = RECORD
    cpu: CpuKind;
    fpu: FpuKind;
  END;
```

This enables you to make sure that your program only runs on the hardware configuration it was written for. For example:

```
IMPORT Lib, IO;
...
VAR
  CPU : Lib.CpuRec;
...
Lib.CpuId(CPU);
IF (CPU.cpu <> Lib.cpu_80386)
  OR (CPU.fpu <> Lib.fpu_80387) THEN
  IO.WrStr(
    "Won't run without a 80386 & 80387");
  HALT;
END;
...
```

Delay — Timed delay

```
PROCEDURE Delay( Time : CARDINAL );
```

Calling this procedure delays your program by Time milliseconds. For example:

```
Lib.Delay(60000); (* a 1 minute delay *)
```

DisableBreakCheck — Switch off CtrlBrk

PROCEDURE DisableBreakCheck;

This procedure disables the control-break handler. After a call to this procedure, your program can no longer be aborted with *Ctrl-Brk*. You can switch on back on using *EnableBreakCheck*. The check setting is ON by default.

Dos — Call DOS

PROCEDURE Dos(VAR R : SYSTEM.Registers);

Allows you to access DOS services through the DOS function handler (int 21h). Consult your DOS documentation for details of the various function calls. Also see the SYSTEM Module for a description of SYSTEM.Registers.

Note: **This procedure is effective under DOS only.**

EnableBreakCheck — Switch on CtrlBrk checking

PROCEDURE EnableBreakCheck;

This procedure enables the control-break handler. After a call to this procedure, your program can be aborted with *Ctrl-Brk*. You can switch on back off using *DisableBreakCheck*. The check setting is OFF by default.

The run-time message displayed, and ERRORINF.\$\$\$ file created, will be the same as that for a *UserBreak* call. See Appendix A : '*Run-time Error Codes*'.

Environment — Get Environment Variable by Number

```
PROCEDURE Environment(
    N: CARDINAL): CommandType;
```

The Environment procedure returns a pointer to the Nth operating system environment string. The type CommandType is defined in Lib.DEF as:

```
TYPE
    CommandType = POINTER TO ARRAY [0..126] OF CHAR;
```

The example below shows how to get and display an OS/2 environment string using Environment. For example:

```
VAR
    CString : Lib.CommandType;
    WhichString : CARDINAL;
...
CString := Lib.Environment( WhichString);
IO.WrStr(CString^);
(*
    which might print 'PROMPT=$p$g',
    for example
*)
```

EnvironmentFind — Find environment variable by name

```
PROCEDURE EnvironmentFind(
    Name: ARRAY OF CHAR;
    VAR Result: ARRAY OF CHAR );
```

This procedure searches the environment string table of the current process looking for one beginning with the specified variable Name. It returns, in Result, the environment string with the variable name and the '=' stripped off. For example:

```
(* Print the value of the current 'PATH' *)
VAR
    PathVal : ARRAY [1..80] OF CHAR;
...
Lib.EnvironmentFind("PATH",PathVal);
IO.WrStr(PathVal);
(*
    If you machine's environment table
    contained:
        PATH=C:\OS/2;C:\COM;C:\EXE;C:\TS;
    Then the above fragment would print:
        C:\OS/2;C:\COM;C:\EXE;C:\TS;
    on the screen
*)
```

Exec — Execute another program

PROCEDURE Exec(

Command: ARRAY OF CHAR;
 Param: ARRAY OF CHAR;
 Env: ExecEnvPtr): CARDINAL;

The Exec procedure enable you to run other programs from within your own programs. The pathname of the program to run should be provided in the Command argument. The Param argument should contain the rest of the command line for the program, where appropriate. The type ExecEnvPtr is defined in Lib.DEF, as follows:

```
TYPE
  ExecEnvPtr = POINTER TO ARRAY OF ADDRESS;
```

This is used to pass a set of environment strings to the called program. If you simply wish the child process to inherit your set of environment strings then the Env parameter should be set to NIL. If you do wish to pass a new set of environment strings, then the array pointed to by Exec should be initialized so that each ADDRESS element of the array in turn points to a zero terminated string containing the environment strings in the form:

```
variable_name = string_of_chars
```

The end of the environment string pointers should be marked by a NIL pointer.

Exec returns a CARDINAL value which reflects the exit code of the program in question. If Exec is unable to run the program for any reason, it will return MAX(CARDINAL). For example:

```
(*
  The following code fragment runs the
  following command:
    c:\dos\xcopy a: b:
*)

VAR
  ExitCode : CARDINAL;
...
ExitCode := Lib.Exec(
  "c:\dos\xcopy",
  "a: b:",NIL);
```

ExecCmd — Execute a DOS or OS/2 command

```
PROCEDURE ExecCmd(
    Command: ARRAY OF CHAR): CARDINAL;
```

The ExecCmd procedure runs an operating system command (such as date or dir). Command should be set to the command line, plus arguments, of the command you wish to run. The procedure returns the exit status of the command, indicating whether it was a success or a failure. For example:

```
(*
  To run the DIR command from within
  your program:
*)
VAR
  ExitStatus : CARDINAL
...
ExitStatus := ExecCmd("dir");
```

Execute — Execute a program

```
PROCEDURE Execute(Name: ARRAY OF CHAR;
    CommandLine: ARRAY OF CHAR;
    StoreAddr: ADDRESS;
    StoreLen: CARDINAL): CARDINAL;
```

The procedure executes a program. Name is the full pathname (including extension) of the program you wish to execute. CommandLine holds the command line parameters, if any, that you wish to pass to the program. You must also provide the address of the start of an area of memory (StoreAddr) and the size of the area (StoreLen) where the program can execute. This storage could be allocated using the Storage module procedure ALLOCATE.

Execute returns the DOS return code. Zero means that the program ran and terminated normally; for other values consult your DOS documentation.

The procedure Exec carries out much the same action and is considerably simpler to use.

FatalError — Terminate process with error message

```
PROCEDURE FatalError(S: ARRAY OF CHAR);
```

The string S is written to StandardOutput and the program is terminated by calling HALT.

Fill — Fill memory with byte

```
PROCEDURE Fill(
    Dest: ADDRESS;
    Count: CARDINAL;
    Value: BYTE);
```

Stores Value in the Count consecutive bytes starting at Dest. This is a fast way of initializing areas of memory in your program. The procedure WordFill is faster when Count is even. For example:

```
Lib.Fill(str1, Str.Length(str1), 47);
```

This fills str1 with the character ‘/’ (ASCII code 47).

GetDate — System date & time procedures

GetTime

SetDate

SetTime

```
PROCEDURE GetDate(
    VAR Year, Month,
    Day: CARDINAL;
    VAR DayOfWeek: DayType);

PROCEDURE GetTime(
    VAR Hrs, Mins,
    Secs, Hsecs: CARDINAL);

PROCEDURE SetDate(
    Year, Month,
    Day: CARDINAL) : BOOLEAN;

PROCEDURE SetTime(
    Hrs, Mins,
    Secs, Hsecs: CARDINAL) BOOLEAN;
```

These four procedures allow you to determine and change the system date and time.

The date procedures use the following enumerated type, defined in Lib.DEF:

```
TYPE
    DayType = (Sunday, Monday, Tuesday,
               Wednesday, Thursday,
               Friday, Saturday);
```

The parameters to each procedure are self-explanatory (Hsecs stands for hundredths of a second).

HashString — Compute hash value from string

```
PROCEDURE HashString(  
    S: ARRAY OF CHAR;  
    Range: CARDINAL): CARDINAL;
```

The HashString procedure calculates a hash value based on S in the range 0 to Range - 1. For any string the hash value over the same range is always the same, so this procedure can be used to map strings onto numbers for fast indexing. For example:

```
VAR  
    HashNr : CARDINAL;  
...  
HashNr := Lib.HashString("Hello",26);  
(* HashNr = 4 *)  
HashNr := Lib.HashString("hello",26);  
(* HashNr = 14 *)
```

Intr — Software Interrupt

```
PROCEDURE Intr(  
    VAR R: SYSTEM.Registers;  
    I: CARDINAL);
```

Allows you to make a software interrupt directly, i.e., to bypass the DOS function handler (int 21h). I is the interrupt number. Consult your DOS documentation for details of the various interrupts. Also see the SYSTEM Module for a description of SYSTEM.Registers.

Note: This procedure is only available under DOS.

MathError — MATHLIB Error Handling

```
PROCEDURE MathError( e : Exception ) : INTEGER;
```

This is the default error handling procedure for MATHLIB. The format of Exception and the meaning of its component fields is described in the MATHLIB module.

Move — move memory bytes

```
PROCEDURE Move(  
    Source, Dest: ADDRESS;  
    Count : CARDINAL);
```

This procedure copies a block of Count bytes from Source to Dest. The copy direction is chosen so that overlapping blocks are copied correctly. The operation is independent of type so the information at Dest will only be in the proper form if you move the appropriate number of bytes. If you are moving an even number of bytes you will find that WordMove is considerably faster. For example:

```
Lib.Move(ADR(val1),ADR(val2),4);
```

This moves four bytes from the start of val1's memory area to the start of val2's memory area.

NoSound — Turn off speaker

```
PROCEDURE NoSound;
```

Under DOS, this procedure turns off the computer's speaker, thus ending a tone generated with Sound. For example:

```
Lib.Sound(1000);  
Lib.Delay(500);  
Lib.NoSound;  
(*  
    This sounds a 1KHz tone for  
    approximately 0.5 seconds  
*)
```

ParamCount — Command line parameter count

```
PROCEDURE ParamCount() : CARDINAL;
```

This procedure returns the number of arguments on the command line. The program name is not counted as one of the command line arguments. For example, if you started the program pctest with the following command line:

```
pctest one two three four
```

then ParamCount would return 4 as the number of command line arguments.

ParamStr — Get a command line argument

```
PROCEDURE ParamStr( VAR S : ARRAY OF CHAR; N :
                    CARDINAL );
```

This procedure returns, in S, the Nth parameter from the command line. The first argument is number 1. For example:

```
(*
  If a program is called with the
  following command line:
      pctest one two three four
  Then:
*)

VAR
  CArg : ARRAY [1..40] OF CHAR;
...
ParamStr(CArg,3);
(* CArg = 'three' *)
```

QSort — General purpose sorting routines

```
PROCEDURE QSort(
    N: CARDINAL;
    Less: CompareProc;
    Swap: SwapProc);

PROCEDURE HSort(
    N: CARDINAL;
    Less: CompareProc;
    Swap: SwapProc);
```

These two procedures implement two different sorting algorithms. Their parameter specifications are, however, identical, so they will be discussed together. The types CompareProc and SwapProc are defined in Lib.DEF, thus:

```
TYPE
  CompareProc = PROCEDURE(
    CARDINAL,
    CARDINAL) : BOOLEAN;
  SwapProc    = PROCEDURE(
    CARDINAL,
    CARDINAL);
```

Both the sort procedures are designed to work with arrays that can be referenced with a CARDINAL index. [1..N] is the number of elements to be sorted. The parameters passed to Less and Swap are the indexes of the elements of the array that the sort procedure is currently interested in.

The procedure passed as the Less parameter will be called whenever the sort routine wishes to compare two values. It should return TRUE if the element referenced by the first parameter is less than the element referenced by the

second parameter. The sort procedure is not interested in how the comparison works, merely that it is possible to place the elements in order.

The procedure passed as the Swap parameter will be called whenever the sort routine decides that two elements need exchanging. The two parameters are the indexes of the elements to exchange.

For example:

```

MODULE SortTest;
IMPORT Lib;
VAR
  ar: ARRAY [1..100] OF INTEGER;
  (* The array to be sorted *)
PROCEDURE Compare(
  i, j: CARDINAL): BOOLEAN;

BEGIN
  RETURN (ar[i] < ar[j]);
END Compare;

PROCEDURE SwapThem(i, j: CARDINAL);
VAR
  tmp: INTEGER;
BEGIN
  tmp := ar[i];
  ar[i] := ar[j];
  ar[j] := tmp;
END SwapThem;

PROCEDURE LoadArray(ar: ARRAY OF INTEGER);
END
  (*
    This procedure simply loads values
    into the array ar
  *)
END LoadArray;

BEGIN
  LoadArray(ar);
  (* Get the Data *)
  Lib.QSort(100, Compare, SwapThem);
  (* and sort it *)
END SortTest.

```

We could just as well have used HSort (heapsort) instead of QSort (quicksort) in this example.

So what is the difference between QSort and HSort? Quicksort is generally faster but can slow down dramatically in the worst case when the elements are already sorted, or very nearly so. Although heapsort is slower for randomly ordered data, it has a very small variance in speed. On the other hand quicksort is a stable algorithm, it does not swap keys that are equal.

RAND — Generate random Real Number

PROCEDURE RAND(): REAL;

Returns a random REAL number, x, in the range: $0.0 \leq x < 1.0$. For example:

```
VAR
  RVal: REAL;
...
RVal := RAND();
(* RVal = 0.100154, for example *)
```

RANDOM — Generate random Cardinal

PROCEDURE RANDOM(Range: CARDINAL): CARDINAL;

Returns a random CARDINAL number between 0 and Range - 1, inclusive. For example:

```
VAR
  RVal : CARDINAL;
...
RVal := RANDOM(10000);
(* RVal = a random number between 0 and 9999 *)
```

RANDOMIZE — Initialize Random number generator

PROCEDURE RANDOMIZE;

This procedure initializes the pseudo-random number generator. It must be called before you use RAND or RANDOM to generate random numbers. If you do not call RANDOMIZE then the other two routines will generate the same sequence of numbers every time your program is run.

ScanL — Search Memory for Byte Left

```
PROCEDURE ScanL(
    Dest: ADDRESS;
    Count: CARDINAL;
    Value: BYTE): CARDINAL;
```

ScanL searches for the first occurrence of the byte Value in the block of memory starting at Dest and length Count bytes working downwards in memory from Dest. The procedure returns the positive position relative to Dest, starting at zero. If Value is not found then the procedure returns Count.

The procedure ScanR is similar but searches in the opposite direction. For example:

```
str1 := "abcdefghijklmnopqrst";
Res := Lib.ScanL(
    ADR(str1[10]),
    10,
    SHORTCARD('d'));
(* Res = 7 *)
Res := Lib.ScanL(
    ADR(str1[10]),
    10,
    SHORTCARD('k'));
(* Res = 0 *)
```

ScanNeL — Search Memory for unequal byte Left

```
PROCEDURE ScanNeL(
    Dest: ADDRESS;
    Count: CARDINAL;
    Value: BYTE ): CARDINAL;
```

ScanNeL searches for the first byte before Dest that differs from Value in the block of memory starting at Dest and length Count bytes working downwards in memory. The procedure returns the positive position relative to Dest, starting at zero. If no such differing byte is found Value is not found then the procedure returns Count.

The procedure ScanNeR is similar but searches in the opposite direction. For example:

```
str1 := "aaaaabbbbb";
Res := Lib.ScanNeL(
    ADR(str1[9]),
    10,
    SHORTCARD('a'));
(* Res = 5 *)
str1 := "aaaaaaaaaa";
Res := Lib.ScanNeL(
    ADR(str1[9]),
    10,
    SHORTCARD('b'));
(* Res = 0 *)
```

ScanNeR — Search Memory for unequal Byte Right

```
PROCEDURE ScanNeR(
    Dest: ADDRESS;
    Count: CARDINAL;
    Value: BYTE): CARDINAL;
```

ScanNeR searches for the first byte after Dest that differs from Value in the block of memory starting at Dest and length Count bytes. The procedure returns the position relative to Dest, starting at zero. If no such differing byte is found Value is not found then the procedure returns Count.

The procedure ScanNeL is similar but searches in the opposite direction. For example:

```
Res := Lib.ScanNeR(
    ADR('aaaaabbbcc'),
    10,
    SHORTCARD('a'));
(* Res = 5 *)
Res := Lib.ScanNeR(
    ADR('aaaaaaaaa'),
    10,
    SHORTCARD('a'));
(* Res = 10 *)
```

ScanR — Search Memory for Byte Right

```
PROCEDURE ScanR(
    Dest: ADDRESS;
    Count: CARDINAL;
    Value: BYTE ): CARDINAL;
```

ScanR searches for the first occurrence of the byte Value in the block of memory starting at Dest and length Count bytes. The procedure returns the position relative to Dest, starting at zero. If Value is not found then the procedure returns Count.

The procedure ScanL is similar but searches in the opposite direction. For example:

```
Res := Lib.ScanR(
    ADR('klmnopqrst'),
    10,
    SHORTCARD('p'));
(* Res = 5 *)
Res := Lib.ScanR(
    ADR('klmnopqrst'),
    10,
    SHORTCARD('x'));
(* Res = 10 *)
```

SetJump — Long Jumps

LongJump

```
PROCEDURE SetJump(  
    VAR Lbl: LongLabel): CARDINAL;  
PROCEDURE LongJump(  
    VAR Lbl: LongLabel;  
    result : CARDINAL );
```

Long jumps are a restricted way of achieving non-local GOTOs. They allow the flow of execution to be transferred from one procedure to another, without using the normal call/return mechanism. The primary use of long jumps is to deal with error situations.

The type, LongLabel, is defined in Lib.DEF.

A variable of type LongLabel is used to hold information on the context of the long jump - the exact details do not concern us here.

The two procedures, SetJump and LongJump are used to produce the long jump. SetJump is used to mark the position of a LongLabel in a piece of code and LongJump is used to transfer control to that label. Notice that unlike a normal LABEL and GOTO combination, SetJump must actually be called to set the position of the long jump - a normal GOTO is handled by the compiler.

When SetJump is called, it saves the state of the procedure in the LongLabel array and returns to the calling procedure with a value of 0. At some later time a call to LongJump will be made referring to the same Lbl variable. This restores the state saved in the Lbl variable. LongJump does not return to its caller but to the called of the corresponding SetJump. To the caller of SetJump it appears that SetJump has returned with the value specified by the result parameter of LongJump. The diagram (Figure 5.1) illustrates this.

The diagram also shows that the procedure containing the relevant SetJump must be active (i.e., it must not have returned) when LongJump is called. If you forget this rule, the results will be unpredictable.

You should also be careful that the compiler does not expect any variables to be held in the CPU's registers in the locality of the SetJump. This can be accomplished with the data pragma:

```
(*# data ( volatile => on ) *)
```

For example:

```

MODULE LJdemo;

IMPORT Lib, IO;
VAR
  LJbuff: Lib.LongLabel;
  ErrorFlag: BOOLEAN;
PROCEDURE p2;
BEGIN
  IO.WrStr("Entering p2"); IO.WrLn;
  IF ErrorFlag THEN
    Lib.LongJump(LJbuff,1);
  END;
  IO.WrStr("No Error"); IO.WrLn;
END p2;

PROCEDURE p1;
BEGIN
  IO.WrStr("Entering p1"); IO.WrLn;
  IF (Lib.SetJmp(LJbuff) # 0) THEN
    IO.WrStr("LongJump called - exiting");
    IO.WrLn;
    HALT;
  ELSE
    IO.WrStr("LongLabel set"); IO.WrLn;
  END;
  p2;
END p1;
BEGIN
  ErrorFlag := TRUE;
  p1;
END LJdemo.

```

When this program is run, it produces the following output:

```

Entering p1
LongLabel set
Entering p2
LongJump called - exiting

```

SetReturnCode — Specify return code

```
PROCEDURE SetReturnCode( code : CARDINAL );
```

This procedure allows you to set the DOS or OS/2 return code that will be passed back to the operating system when your program terminates. If your program started from the command line or in a .BAT or .CMD file, you can use the ERRORLEVEL code to determine the result of the program.

Sound — Sound speaker

```
PROCEDURE Sound( FreqHz : CARDINAL );
```

Under DOS, this procedure turns on the computer's speaker with a sound of FreqHz Hz (cycles per second). The sound continues until NoSound is called. For example:

```
Lib.Sound(1000); (* issues a 1KHz sound *)
```

SysErrno — Get last error code

```
PROCEDURE SysErrno(): CARDINAL;
```

The procedure returns code of the last operating system error to occur.

Terminate — Link termination procedures

```
PROCEDURE Terminate( P : PROC; VAR C : PROC );
```

Terminate is used to specify the actions you want your program to take when it terminates. A program can terminate either normally (by reaching the END of the main MODULE) or by a call to HALT. A call to Terminate causes the HALT procedure to call P instead of the normal termination function. When you make a call to Terminate it returns the procedure that would have been called in the variable C. Normally your program will call C after P has done its work. In this way you can build a chain of termination procedures.

If HALT is called in P then the program will stop immediately. For example:

```
MODULE TermTest;
IMPORT Lib, IO;
VAR
  Continue1: PROC;
  Continue2: PROC;
PROCEDURE Msg( Txt : ARRAY OF CHAR );
BEGIN
  IO.WrStr(Txt);
  IO.WrLn;
END Msg;
PROCEDURE Close1;
BEGIN
  Msg('Closedown No. 1');
  Continue1;
END Close1;
PROCEDURE Close2;
BEGIN
  Msg('Closedown No. 2');
  Continue2;
END Close2;
```

```
BEGIN
  Msg('Program started');
  Lib.Terminate(Close1,Continue1);
  Lib.Terminate(Close2,Continue2);
  HALT;
  Msg('This message will never appear');
END TermTest.
```

When the above program is run, the following output will occur:

```
Program started
Closedown No. 2
Closedown No. 1
```

UserBreak — Abort Program

PROCEDURE UserBreak;

A call to this procedure causes an error message to be printed, the ERRORINF.\$\$\$ file to be created, and the program terminated using HALT (see Chapter 2). See Appendix A : *'Run-time Error Codes'*.

This procedure is for use with DOS only.

WordFill — Fill memory with word

```
PROCEDURE Fill(
    Dest: ADDRESS;
    WordCount: CARDINAL;
    Value: BYTE );
```

Stores Value in the Count consecutive words starting at Dest. This is a fast way of initializing areas of memory in your program. The procedure Fill fills a memory area with bytes. For example:

```
Lib.WordFill(ADR(vall),4,32768);
```

This puts the bit-pattern for 32768 in the four consecutive words starting at the location of vall.

WordMove — Move memory words

```
PROCEDURE WordMove(  
    Source, Dest: ADDRESS;  
    WordCount: CARDINAL);
```

This procedure is similar to Move except that WordMove moves a machine word (2 bytes) at a time. WordCount specifies the number of words to be moved. This procedure is considerably faster than performing a Move on the same number of bytes. For example:

```
Lib.WordMove(ADR(val1),ADR(val2),4);
```

This moves eight bytes (four words) of memory from the start of val1's memory area to the start of val2's memory area.

WrDosError — Output error message

```
PROCEDURE WrDosError(ErrorNo: SHORTCARD);
```

ErrorNo is a an error code returned by an operating system call, or by SysErrno. The procedure outputs an associated error message to StandardOutput.

If no message text exists for the error, the following is output:

```
Unknown DOS Error : X
```

where X is the error number.

MODULE LIM

Introduction

The module LIM provides an interface to the EMS memory manager. You should be familiar with the techniques involved in using expanded memory before writing programs that use this module.

Variables

```
VAR
    LIMPresent : BOOLEAN;
    PageFr     : CARDINAL;
```

The variable LIMPresent will be set to TRUE if an EMS driver was detected at program startup. When the LIM module is initialized PageFr will be set to the segment value of the EMS page frame.

LIM Reference

The following are the individual procedures defined within the LIM module.

AllocatePages — Allocate expanded pages

```
PROCEDURE AllocatePages(n: CARDINAL): CARDINAL;
```

The procedure allocates n pages of expanded memory. The value returned is the handle that will be used to access those pages in subsequent operations.

DeAllocatePages — Deallocate expanded pages

```
PROCEDURE DeAllocatePages(Handle: CARDINAL);
```

The procedure deallocates the pages associated with handle n. Handle must be a value returned by AllocatePages. Handle will be invalid after this call.

FreePages — Get memory available

PROCEDURE FreePages() : CARDINAL;

The procedure returns the number of pages available for allocation via AllocatePages.

GetPageFrame — Get page frame

PROCEDURE GetPageFrame(): CARDINAL;

The procedure returns the segment value of the page frame used to access expanded memory.

GetStatus — Return error status of the EMS driver

PROCEDURE GetStatus() : SHORTCARD;

The procedure returns the current error status of the memory manager. This procedure should be called after any request to the EMS memory manger.

Code	Status
00H	Function successful.
80H	Internal EMS driver Error.
81H	Hardware failure.
82H	Busy.
83H	Invalid handle.
84H	Undefined function.
85H	No more EMS handles available.
86H	Error in save or restore of map context.
87H	Not enough logical pages physically available.
88H	Not enough pages available.
89H	Zero allocation.
8AH	Logical page number out of range.
8BH	Illegal physical page number.
8CH	Mapping save area full.
8DH	Mapping context save failed (already saved).
8EH	Mapping context restore failed (not saved).
8FH	Subfunction parameter undefined.

MapPage — Map physical and logical pages

```
PROCEDURE MapPage(  
    PPage, LPage,  
    Handle: CARDINAL): CARDINAL;
```

The procedure maps logical page LPage of expanded memory to physical page PPage. Handle must be valid a handle returned by AllocatePages.

MODULE MATHLIB

Introduction

This module implements some common mathematical functions. In addition the module contains some specific 80x87 procedures. The latter should only be required for specialized applications.

Error Handling

If you supply one of the MATHLIB procedures with invalid arguments (for example, trying to take the square root of a negative number with Sqrt(-1.0)) then the following procedure variable will be used to call an error handler:

```
VAR
  MathError : PROCEDURE(VAR Exception ) : INTEGER;
```

The Exception type is defined as follows:

```
TYPE
  Exception = RECORD
    type      : INTEGER;
    name      : ADDRESS;
    arg1,
    arg2,
    retval    : LONGREAL;
  END;
```

The type field specifies one of the following:

<code>_DOMAIN</code>	parameter is not in the valid domain for this procedure.
<code>_SING</code>	procedure value cannot be calculated here.
<code>_OVERFLOW</code>	result exceeds the allowed range.
<code>_UNDERFLOW</code>	result is too close to zero to be represented.
<code>_TLOSS</code>	total loss of precision.
<code>_PLOSS</code>	partial loss of precision.

The name field is a pointer to a nil-terminated string containing the name of the procedure that caused the error. Note that the name uses lower case.

arg1 and arg2 are the parameters to the procedure. Procedures that take only one parameter will only use arg1.

retval the return value for the procedure that caused the error.

If the MathError procedure returns 0, the procedure generating the error will return its default error value and set the system error variable.

If the MathError procedure returns non zero, the procedure generating the error will return the value in retval and will not set the system error variable.

MATHLIB Reference

The following are the individual procedures defined within the MATHLIB module.

Exp — Exponential function

PROCEDURE Exp(A : LONGREAL) : LONGREAL;

Exp computes the Exponential function for A, i.e., raises e to the power A. This is the inverse function of Log.

Log — Logarithmic Functions

Log10

PROCEDURE Log(A :LONGREAL): LONGREAL;
PROCEDURE Log10(A: LONGREAL): LONGREAL;

Log computes the natural logarithm (log to base e) of the argument A. The argument must be positive.

Log10 computes the logarithm of A to the base 10. The argument must be positive.

LongToBcd — Convert real number to BCD format

BcdToLong

```
PROCEDURE LongToBcd(A: LONGREAL): PackedBcd;
PROCEDURE BcdToLong(A: PackedBcd): LONGREAL;
```

The type PackedBcd is defined in MATHLIB.DEF as:

```
TYPE
  PackedBcd = ARRAY [0..9] OF SHORTCARD;
```

This represents an 17-digit decimal value (two decimal digits are stored per byte plus the sign) called a binary coded decimal (BCD). LongToBcd converts a LONGREAL value into the equivalent BCD integer with rounding. BcdToLong performs the opposite conversion, giving the LONGREAL version of the integer stored in BCD format.

MathError — MATHLIB Error Handling

```
PROCEDURE MathError( e : Exception ) : INTEGER;
```

This is the error handling procedure for MATHLIB.

The procedure name passed to the error handler is generic and not specific to the High Level Language function that called the math primitive, e.g.:

```
MATHLIB.Sqrt passes sqrt
```

Mod — Real Modulus

```
PROCEDURE Mod( X, Y : LONGREAL ) : LONGREAL;
```

Mod returns the remainder after subtracting Y from X as often as possible. More specifically the function returns the result of the following expression:

$$X - (Y * \lfloor \text{ABS}(X / Y) \rfloor)$$

The symbols $\lfloor \dots \rfloor$ represent the floor function. This represents the largest integer less than or equal to the value between the symbols – in this case the quotient of (X / Y) . The ABS function is used to ensure the result is rounded towards zero (because, for example, $\lfloor -12.5 \rfloor = -13$).

Pow — Raise to the power

PROCEDURE Pow(X, Y : LONGREAL) : LONGREAL;

Pow computes the value of X raised to the power Y.

Rexp — Split Real Number into Mantissa and Exponent

PROCEDURE Rexp(
VAR I :
INTEGER;
A: LONGREAL): LONGREAL;

Rexp splits the real number A into its exponent and mantissa parts. The exponent (for base two) is returned in I and the function's return value is the mantissa. For example:

```
result := Rexp(Exponent,79.37);
```

This sets result to 1.24016 and Exponent to 6. To check this, multiply 1.24016 by 64 ($64 = 2^6$).

Sin — Trigonometric Functions

Cos

Tan

ASin

ACos

ATan

ATan2

```
PROCEDURE Sin(A: LONGREAL): LONGREAL;  
PROCEDURE Cos(A: LONGREAL): LONGREAL;  
PROCEDURE Tan(A: LONGREAL): LONGREAL;  
PROCEDURE ASin(A: LONGREAL): LONGREAL;  
PROCEDURE ACos(A: LONGREAL): LONGREAL;  
PROCEDURE ATan(A: LONGREAL): LONGREAL;  
PROCEDURE ATan2(X, Y: LONGREAL): LONGREAL;
```

Sin, Cos and Tan implement the correspondingly named mathematical functions. The argument A to these procedures is expressed in radians (remember, 360° is equal to 2π radians). Sin and Cos return values in the range -1.0 to 1.0.

ASin, ACos and ATan return the arc sine, arc cosine and arc tangent, respectively. The argument to ASin and ACos must be in the range -1.0 to 1.0. ASin and ATan return values in the range $-(\pi / 2.0)$ to $(\pi / 2.0)$. ACos returns a value in the range 0 to π .

ATan2 returns the arc tangent of (Y/X) and the result is in the range $-\pi$ to π .

SinH — Hyperbolic Function

CosH

TanH

```
PROCEDURE SinH(A: LONGREAL): LONGREAL;  
PROCEDURE CosH(A: LONGREAL): LONGREAL;  
PROCEDURE TanH(A: LONGREAL): LONGREAL;
```

These functions implement the hyperbolic sine, hyperbolic cosine and hyperbolic tangent, respectively, of the argument A.

Sqrt — Square Root

```
PROCEDURE Sqrt(A: LONGREAL): LONGREAL;
```

This produces the square root of A. The argument must be positive or zero.

MODULE MsMouse

Introduction

The MsMouse module provides a full interface to the Microsoft Mouse driver. This allows your programs to make use of a Microsoft of Microsoft-compatible mouse in both text and graphic mode.

Data Types and Constants

There are number of special data types and constants associated with using the MsMouse module. Since they are used in several procedures in the module, they are described now rather than later. The full list, taken from MsMouse.DEF is as follows:

TYPE

```
MsData = RECORD
    left_pressed,
    middle_pressed,
    right_pressed : BOOLEAN;
    actions : CARDINAL;
    row,
    col : INTEGER;
END;
```

```
MsRange = RECORD
    max_col,
    min_col,
    max_row,
    min_row : INTEGER;
END;
```

```
Msmotion = RECORD
    vert,
    horiz : INTEGER;
END;
```

```
MsGraphcur = RECORD
    row,
    col : INTEGER;
    screen_mask: ARRAY [0..15] OF CARDINAL;
    cursor_mask: ARRAY [0..15] OF CARDINAL;
END;
```

```
Hardware = RECORD
    start_scan,
    end_scan : INTEGER;
END;
```

```

Software = RECORD
    screen_mask,
    cursor_mask : CARDINAL;
END;

```

```

MsTextCur = RECORD
    type: BOOLEAN;
    CASE : BOOLEAN OF
    | TRUE :
        H: Hardware;
    | FALSE :
        S: Software;
    END;
END;

```

```

MsSense = RECORD
    h_speed,
    v_speed,
    threshold: INTEGER;
END;

```

CONST

```

_MS_ON      = TRUE;
_MS_OFF     = FALSE;
_MS_HIDE    = FALSE;
_MS_SHOW    = TRUE;

_MS_MIDDLE  = 2;
_MS_LEFT    = 0;
_MS_RIGHT   = 1;

_MS_HARD    = TRUE;
_MS_SOFT    = FALSE;

```

The Microsoft mouse driver assumes that three buttons are present on the mouse - left, middle and right.

The MsData record is used to indicate the actions that the user has carried out with the mouse. The fields `left_pressed`, `middle_pressed` or `right_pressed` will be TRUE to indicate whether or not each of the buttons has been pressed. The field `actions` reports the number of times the requested action (pressing or releasing) has occurred. Which button action refers to depends on the procedure call (see `GetPress`, `GetRelease` and `GetStatus`). The fields `row` and `col` report the co-ordinates at which the action occurred. These co-ordinates are graphics based irrespective of the mouse state.

The MsRange record defines the range of co-ordinates over which the mouse cursor may move. The four fields, `min_col`, `min_row`, `max_col` and `max_row`, specify the minimum row and column and the maximum row and column, respectively. These co-ordinates are graphics based irrespective of the mouse state.

The MsMotion record is used to report the distance the mouse pointer has moved horizontally and vertically.

The MsGraphcur record specifies the shape of the graphics cursor. The row and col fields specify the location of the hot-spot in the cursor. The hot-spot is the point in the cursor that will be reported as being the position of the cursor. They can range from -16 to +16. The screen_mask and cursor_mask fields define the shape of the cursor. The contents of the screen_mask array will be displayed on the screen by using a bit-wise AND with the current contents – it is mostly set to all zeroes (which has no effect on the display) unless special effects are required. The cursor_mask, on the other hand, is bit-wise ORed with the screen display. Each of these arrays can be considered as a 16 x 16 grid of pixels defining the cursor's shape.

The Hardware and Software records are used to define the color and size of the text cursor. One of them is used with the MsTextCur record to define which type of cursor is to be used. The Hardware cursor simply specifies the start and end scan lines of a character cell (0 - 7 on a CGA screen) - a start_scan of 0 and end_scan of 7 defines a block cursor covering the entire character cell. The Software cursor contains two CARDINAL values that are combined with the character and color attributes of the character cell to obtain the cursor. The screen_mask field is ANDed with the character and color values and the cursor_mask field is XORed with them. When selecting the text cursor (see SetTextCursor) the type field of MsTextCur is used to select a hardware or software cursor.

The MsSense record is used to define the sensitivity of the mouse. The h_speed and v_speed fields are used as multipliers to translate the mouse's movements into mouse cursor movements. The threshold field is used to specify the speed at which the mouse's speed will double the movement of the cursor on the screen.

MsMouse Reference

The following are the individual procedures defined within the MsMouse module.

Cursor — Set the cursor mode

```
PROCEDURE Cursor(Mode: BOOLEAN);
```

This procedure switches the mouse cursor on and off depending on the setting of Mode. If Mode is set to `_MS_SHOW` the mouse cursor will be visible. If it is set to `_MS_HIDE` the mouse cursor will be hidden.

You can make multiple, nested calls to Cursor with the `_MS_HIDE`, but the same number of calls with `_MS_SHOW` will be needed to display the cursor again.

DriverSize — Bytes needed to store driver state

```
PROCEDURE DriverSize() : CARDINAL;
```

This procedure returns the number of bytes required to store the current state of the mouse driver. It is used in conjunction with `SaveDriver` and `RestoreDriver`.

GetMotion — Distance moved since last call

```
PROCEDURE GetMotion( VAR mp : MsMotion );
```

This procedure returns in mp the distance moved since the last call to `GetMotion`.

GetPage — Display Page

SetPage

```
PROCEDURE GetPage(): CARDINAL;  
PROCEDURE SetPage(Page: CARDINAL);
```

The procedure `GetPage` returns the number of the display page on which the mouse cursor appears. The range of possible values depends on the video adaptor installed in your computer.

The procedure `SetPage` sets the page on which the mouse cursor will appear. Again, the range of permissible values depends on the video adaptor installed in your computer.

GetPress — Get button press information

```
PROCEDURE GetPress( Button: INTEGER;  
                   VAR mp: MsData);
```

This procedure returns in `mp` information about the button presses for `Button`. The `actions` field of `mp` will contain the number of times `Button` has been pressed. `Button` has one of the values: `_MS_LEFT`, `_MS_MIDDLE` or `_MS_RIGHT`.

GetRelease — Get button release information

```
PROCEDURE GetRelease( Button: INTEGER;  
                    VAR mp: MsData);
```

This procedure returns in `mp` information about the button releases for `Button`. The `actions` field of `mp` will contain the number of times `Button` has been released. `Button` has one of the values: `_MS_LEFT`, `_MS_MIDDLE` or `_MS_RIGHT`.

GetSensitivity — Get/Set mouse sensitivity settings

SetSensitivity

```
PROCEDURE GetSensitivity(VAR mp: MsSense);  
PROCEDURE SetSensitivity(VAR mp: MsSense);
```

The GetSensitivity procedure returns in mp the current sensitivity of the mouse.

The SetSensitivity procedure sets new sensitivity values.

GetStatus — Get mouse status

```
PROCEDURE GetStatus( VAR mp : MsData );
```

This procedure returns in mp the current mouse status. The actions filed is, in this case, undefined.

LightPen — Set light pen emulation

```
PROCEDURE LightPen( Mode : BOOLEAN );
```

LightPen turns the light pen emulation of the mouse on and off depending on the value of Mode. Set Mode to `_MS_ON` to turn light pen emulation on or `_MS_OFF` to set it off.

Reset — Reset mouse driver

```
PROCEDURE Reset() : INTEGER;
```

This procedure resets the mouse driver to its initial state and returns the driver status. The return value indicates the number of buttons:

- 1 Two Buttons
- 0 Other than two buttons
- 3 Mouse Systems Mouse

Reset will return `MAX(INTEGER)` if no mouse driver is installed.

SaveDriver — Save/Restore Driver state

RestoreDriver

```
PROCEDURE SaveDriver(Buffer: ADDRESS);  
PROCEDURE RestoreDriver(Buffer: ADDRESS);
```

These two procedures allow you to save and restore the current mouse driver settings. This lets you leave the mouse driver in the same state as you found it; always a courtesy to other programs.

The Buffer parameter points to a memory area whose size must be determined using the DriverSize procedure.

SetDouble — Set double speed threshold

```
PROCEDURE SetDouble( Threshold : INTEGER );
```

This procedure sets the double-speed threshold for the mouse. This is the movement speed of the mouse above which the cursor movements will be doubled.

SetGraphCursor — Set graphics cursor shape

```
PROCEDURE SetGraphCursor(VAR mp: MsGraphcur );
```

This procedure uses mp to set the shape of the graphics cursor. Note that the graphics cursor only appears when the screen is in graphics mode and, likewise, the text cursor appears when the screen is in text mode. You can, however, set the shapes of these cursors whatever the screen mode.

SetInterrupt — Set interrupt call mask

```
PROCEDURE SetInterrupt(VAR mp: MsInterrupt);
```

This procedure sets the interrupt mask to the value contained in mp.mask. The interrupt handler whose address is contained in mp.IntFunc is installed to service mouse hardware interrupts.

SetMickeys — Define mickeys per click ratio

PROCEDURE SetMickeys(Vert, Horiz: INTEGER);

This procedure defines the mickeys per click ratio, both vertically and horizontally. The default values are 8 horizontally and 16 vertically.

SetPosition — Set cursor position

PROCEDURE SetPosition(Row, Col : INTEGER);

This procedure moves the mouse cursor to the specified position. Depending on the screen mode, Row and Col are either text or graphics co-ordinates.

SetRange — Set area for mouse cursor

PROCEDURE SetRange(VAR mp : MsRange);

This procedure sets the area to which the mouse cursor is to be confined. This can be any areas of the screen specified by mp. Depending on the screen mode, the co-ordinates are either text or graphics co-ordinates.

SetTextCursor — Set text cursor shape

PROCEDURE SetTextCursor(VAR mp: MsTextCur);

This procedure defines the size and color of the text Cursor using mp. Note that the graphics cursor will only appear when the screen is in graphics mode and that, likewise, the text cursor will appear when the screen is in text mode. You can, however, set the shapes of these cursors whatever the screen mode.

UpdateScreen — Specify update area

PROCEDURE UpdateScreen(x1, y1, x2, y2:
INTEGER);

The UpdateScreen procedure is used to tell the mouse driver that an area of the screen is about to be updated. The co-ordinates specify the upper-left of the area (x1 and y1) and the lower-right of the area (x2 and y2). Once the screen has been updated, the mouse cursor must be restored with an explicit call to Cursor.

MODULE Process

Introduction

The procedures in this module handle “concurrent” processes. TopSpeed Modula-2 achieves this by the use of time-slicing. Under this scheme each process is allowed to run for a time before being interrupted by a supervisor program (called the scheduler). The scheduler saves the machine environment of the process (the CPU registers, etc) and then passes control to the next process waiting in the queue. The scheduler decides which processes to run based on their priority, a number indicating “how important” the programmer thinks the process is: the lower the number, the higher the priority and zero means the highest priority of all. Only those processes with the currently highest priority will be running at any one time.

Processes can communicate with each other by using signals. A signal is a queue of messages with a name. A process monitors the state of the signal queue and takes actions depending on the state that it finds. Since the signals are under the control of the scheduler, such communication is perfectly safe. Much safer, in fact, than attempting to pass messages through process controlled global variables.

While simple in concept and straightforward in its implementation the time-sliced scheduler does require some care. Not all procedures are suitable for executing in such an environment without special consideration. Because the scheduler interrupts and restarts processes, the processes themselves must be entirely re-entrant. That is, there must be no chance that their environment will change while the scheduler has put them on hold. Unfortunately, this is not true of pieces of code that rely on interrupts, pieces of code like operating systems.

OS/2 is much better in this respect than DOS, as OS/2 is itself able to run concurrent processes. This can be done by directly employing the relevant OS/2 primitives in your module.

Where you require to make your code both DOS and OS/2 compatible this module must be used.

The solution to the problem is to prevent certain sections of the your code being interrupted by telling the scheduler that you’ve reached a critical point in the program. When the critical point is passed, you simply inform the scheduler of the fact and your process continues being time-sliced as before.

Obviously, if the time-slicing is going to be successful, you need to make sure that these critical regions are small and quick to execute. Otherwise no other process will get a look in.

Skeleton Multi-Thread Programs

All programs wishing to use the scheduler should be compiled with the Multi-Thread model. This ensures that the compiler generates code suitable for interruption by the scheduler.

A typical Multi-Thread program has the following structure:

1. A main module that starts the scheduler and initializes all the signals associated with the program. It may also initialize global variables required by the program. These global areas would be windows, data buffers and detailed message passing variables. The processes would be written to only access these data areas in response to a signal.
2. The main module would then launch all the processes associated with the initial phase of the application and quietly “go to sleep” to await the alarm call – a signal – telling it that the program was now ready to finish (after that, it would clean up the system and exit gracefully).
3. Each process is assigned a small task, such as waiting for keyboard input, displaying a clock, printing messages, etc. If they do too much they will take too long to get the job done.

Processes

Each process would be implemented as a LOOP. Before it starts, the process may want to do some of its own initialization, but this would typically be no more than opening a window or file.

The process would then sit and wait for the event that triggers its action. This could be an external event such as a keypress, or an internal event such as a signal.

As a result of this event, the process would send messages using signals to indicate that the event had occurred and then go back to waiting for the event again.

Example of Use

This simple example displays a single window on the screen and prints a random histogram in it. If a key other than *ESC* is pressed after it has been drawn, the window is moved and re-colored. If *ESC* is pressed, the program terminates.

The program has two processes `DrawHistogram` and `WaitForAKey`, named for the function they perform.

DrawHistogram, after drawing the random data, waits for WaitForAKey to signal (using the SIGNAL, KeyReady) that a key has been pressed. If one has, it checks whether or not the key was Esc, CHR(27), and takes the appropriate action.

WaitForAKey keeps checking if a key has been pressed. If one has, it sees if any process is waiting for the SIGNAL, KeyReady. If some process is, it stores the key and SENDs the appropriate signal. Notice particularly the use of Lock and Unlock around procedure calls involving the operating system (the IO module's RdKey, for example) and around accesses to global variables (CurrentKey in this case). This example also demonstrates how to use the Window module with the Process module. The program uses IO for output, Lib for random numbers and Window for screen window management.

```

MODULE ProcDemo;
IMPORT Process, IO, Window, Lib;
CONST
  Winit = Window.WinDef(
(* Initial Window Definition *)
    30, 8, 50, 16, Window.Yellow,
    Window.Blue,
    FALSE, FALSE, FALSE,
    TRUE, Window.DoubleFrame,
    Window.White, Window.Blue);
  WorkSpace = 1024;
(* Workspace size for processes *)

VAR
  Finished,
  KeyReady: Process.SIGNAL;
  CurrentKey: CHAR;

PROCEDURE WaitForAKey;
(* Process 1: Waits for a key press
   then checks to see if KeyReady is
   Awaited. If it is, then SENDs
   KeyReady after setting CurrentKey *)
VAR
  ch: CHAR;
(* Key pressed *)
  press: BOOLEAN;
(* If key pressed *)

```

```

BEGIN
  LOOP
    Process.Lock;
    press := IO.KeyPressed();
    Process.Unlock;
    IF press THEN
      Process.Lock;
      ch := IO.RdKey();
      Process.Unlock;
      IF Process.Awaited(KeyReady) THEN
(* Check to see if another process is waiting*)
        Process.Lock;
        CurrentKey := ch;
        Process.Unlock;
        Process.SEND(KeyReady);
      END;
    END;
    Process.Delay(2); (* Wait awhile *)
  END;
END WaitForAKey;
PROCEDURE DrawHistogram;
(*
  Process 2 : Open a window; Draw a
  random histogram and then WAIT for
  KeyReady. If the key pressed was ESC,
  signal Finished, else randomly move
  and re-color window.
*)
  VAR
    Win: Window.WinType;
    i : CARDINAL;
  PROCEDURE MoveWindow;
  VAR
    x1, y1,
    x2, y2 : Window.AbsCoord;
  BEGIN
    x1 := Window.AbsCoord(
      Lib.RANDOM(60));
    y1 := Window.AbsCoord(
      Lib.RANDOM(16));
    x2 := x1 + 20;
    y2 := y1 + 8;
    Window.Change(Win,x1,y1,x2,y2);
  END MoveWindow;
  PROCEDURE ReColorWindow;
  VAR
    fore, back : Window.Color;
  BEGIN
    fore := Window.Color(
      Lib.RANDOM(
        ORD(Window.White)));
    back := fore;
    WHILE (back = fore) DO
      back := Window.Color(
        Lib.RANDOM(
          ORD(Window.LightGray)) );
    END;
    Window.SetFrame(
      Win,
      Window.DoubleFrame,
      fore,back);
    Window.TextColor(fore);
    Window.TextBackground(back);
    Window.Clear;
  END ReColorWindow;

```

```

PROCEDURE DrawLine( y : CARDINAL );
BEGIN
    Window.GotoXY(1,Window.ReICoord(y));
    IO.WrCard(y,1);
    IO.WrStr(" : ");
    IO.WrCharRep('*',Lib.RANDOM(16));
END DrawLine;

BEGIN
    Win := Window.Open(Winit);
    (* Initialize window *)
    Window.Use(Win);
    LOOP
        FOR i := 1 TO 7 DO          (* Draw Histogram *)
            DrawLine(i);
            Process.Delay(4);        (* Slowly, for effect *)
        END;
        Process.WAIT(KeyReady);
        (* Wait for a key *)
        Process.Lock;
        IF (CurrentKey = CHR(27)) THEN
            Process.Unlock;
            Window.Close(Win);
            Process.SEND(Finished);
            LOOP
                (* Just sit doing nothing *)
            END;
        END;
        Process.Unlock;
        MoveWindow;
        ReColorWindow;
    END;
END DrawHistogram;

BEGIN (* Main Program *)
    Lib.RANDOMIZE;
    (* Initialize random number generator *)
    Window.SetProcessLocks(
        Process.Lock,
        Process.Unlock);
    (*
    The Window module requires the use of
    a LOCK and UNLOCK procedure to work
    with the scheduler
    *)
    Process.StartScheduler;
    (*
    Firstly, the SIGNALs need initializing *)
    Process.Init(KeyReady);
    Process.Init(Finished);
    (*
    The next two statements start the
    processes
    *)
    Process.StartProcess(
        WaitForAKey,
        Workspace,
        1);
    Process.StartProcess(
        DrawHistogram,
        Workspace,
        1);
    Process.WAIT(Finished);
    (* Wait for program to complete *)
    Process.StopScheduler;
END ProcDemo.

```

Process Reference

The following are the individual procedures defined within the Process module.

Awaited — Is there anyone waiting?

PROCEDURE Awaited(s : SIGNAL) : BOOLEAN;

Returns TRUE if there is at least one process waiting for the signal s.

Delay — Wait for a time

PROCEDURE Delay(t : CARDINAL);

Causes the current processes to be delayed for t time-slices. A time-slice is approximately 1/18th of a second for DOS programs. Under OS/2 1.2, the ticker rate is approximate 31ms.

Init — Initialize a signal

PROCEDURE Init(VAR s : SIGNAL);

Initializes a signal. That is, the procedure initializes the queue of messages it represents. The counter of the signal (the number of outstanding WAIT calls) is set to zero.

Lock — Lock current process

PROCEDURE Lock;

Prevents the current process from being rescheduled until there is a corresponding call to Unlock. Calls to Lock may be nested but there must always be a corresponding Unlock. These two procedures are used to guard shared resources such as the operating system, the screen and keyboard.

Notify — Reschedule when possible

PROCEDURE Notify(s : SIGNAL);

A call to Notify causes the a task waiting on signal s to be rescheduled when possible. If no process is waiting for s then the call has no effect. Unlike SEND it does not cause instant rescheduling.

SEND — Raise a signal

PROCEDURE SEND(s : SIGNAL);

A call to SEND for a particular signal will cause the first process awaiting that signal to become active. If no processes are waiting for the signal, the message is queued to be dealt with later.

StartProcess — Start a new process

PROCEDURE StartProcess(
 P: PROC;
 N: CARDINAL;
 Pr: CARDINAL);

Creates a new process. The process beings with a call to P. The process has priority Pr and uses a workspace of N bytes. N should be at least 0.5K larger than the required local memory space for DOS programs.

The higher the value of Pr, the higher the priority of the process. Processes must never have priorities of less than 1.

For OS/2 programs, the size can be as large or as small as required. If Pr is greater than, or equal to, the priority of the current process (i.e., the one that calls StartProcess) then the new process becomes active immediately.

StartScheduler — Start the Scheduler

PROCEDURE StartScheduler;

Starts the time-sliced scheduler. If it is already active, a call to this procedure has no effect.

StopProcess — Stop a process

PROCEDURE StopProcess();

Stops the calling thread.

StopScheduler — Stops the Scheduler

PROCEDURE StopScheduler;

Stops the scheduler. Note that the SEND and WAIT operations still function when the scheduler is stopped.

Unlock — Unlock current process

PROCEDURE Unlock;

This allows the current process to be rescheduled after being locked. It must always be paired with a call to Lock.

WAIT — Wait for a signal

PROCEDURE Wait(s : SIGNAL);

A call to WAIT makes the calling process wait until a SEND is issued to that signal.

MODULE *ShtHeap*

Introduction

The ShtHeap module contains procedures to manage a short heap. A short heap is a block of memory allocated by the operating system from the far heap. A block of memory can be allocated from the far heap using the ALLOCATE or SegAllocate procedures under in the Storage module while under DOS or from the AllocSeg procedure in the Dos module under OS/2.

The short heap must first be initialized (using the Initialize procedure). Then allocations can be made from the short heap and storage returned to it. The short heap can also be modified in size.

Constants

The following constants and types are defined in ShtHeap.DEF:

```

TYPE
  Segment = CARDINAL;
  Size    = CARDINAL;
  Pointer = SHORTADDR;
VAR
  Clear: BOOLEAN;
  (* Clear allocated memory *)
  Check: BOOLEAN;
  (* Abort program if allocate fails *)
  Debug : BOOLEAN;
  (* Test consistency on every call *)
CONST
  Nil: Pointer(-1);
  (* Returned if Allocate fails *)
  Align : 4;
  (* Pointer alignment *)

```

It is recommended that you use this module rather the storage module for DOS programs you will find it much easier to convert you programs to run under OS/2.

ShtHeap Reference

The following are the individual procedures defined within the IO module.

Allocate — Allocate block from short heap

```
PROCEDURE Allocate(
    H: Segment;
    VAR P: Pointer;
    Z: Size );
```

The Allocate procedure allocates Z bytes from the short heap H and sets P to point to the first byte of the allocated block. If there is not enough memory in the block to satisfy the request, then P is set to Nil.

Free — Free a block back to the short heap

```
PROCEDURE Free(
    H: Segment;
    VAR P: Pointer;
    Z: Size);
```

P is a pointer to a block previously allocated from the short heap H. Z is the size of the memory block that was allocated. The Free procedure returns that memory to the heap, thus allowing it to be re-used. To prevent its further use, the variable P is set to Nil.

Increase — Increase size of short heap

```
PROCEDURE Increase(H: Segment; Z: Size);
```

This procedure increase the size of the heap H to Z bytes. It should only be used on short heaps that have already been initialized. For example:

```
ShtHeap.Increase(MyHeap,8192);
```

Initialize — Initialize a short heap

```
PROCEDURE Initialize( H : Segment; Z : Size );
```

Initializes the segment H (of size Z) as a short heap. You cannot use a short heap until it has been initialized. For example:

```
IMPORT Storage, ShtHeap;
...
VAR
    SegAddress : FarADDRESS;
    MyHeap : CARDINAL;
...
SegAddress := Storage.FarALLOCATE(SegAddress,8192);
MyHeap := CARDINAL(Seg(SegAddress));
ShtHeap.Initialize(MyHeap,4096);
```

Note that you don't have to initialize all the allocated segment at one time.

Largest — Report largest available block

PROCEDURE Largest(H : Segment) : Size;

This procedure returns the size of the largest available block of memory that can be allocated from the heap H. This procedure can be used to see if there is enough memory to satisfy a call to Allocate.

Test — Test the integrity of a short heap

PROCEDURE Test(H : Segment);

This procedure tests the short heap H for integrity. If there is a fault in the short heap, a run-time error occurs.

Total — Total free space

PROCEDURE Total(H : Segment) : Size;

The Total procedure returns the total space available in a short heap. This may not be contiguous memory and available all in one block. Use the Largest procedure to determine the largest block available.

MODULE Storage

Introduction

The Storage module allocates memory from the far heap maintained by the operating system. It allows you to dynamically allocate memory for pointer variables at run-time.

Blocks of memory can be allocated and deallocated in any combination, i.e., blocks of memory do not need to be deallocated in the order they were allocated. It is naturally an error to attempt to deallocate memory that was not allocated from the far heap.

Variables

There are two important variables in the Storage module, `ClearOnAllocate` and `Check`.

The `BOOLEAN` variable `ClearOnAllocate` should be set `TRUE` if you want the `ALLOCATE` procedure to zero-fill memory that it allocates.

The `BOOLEAN` variable `Check` should be set `TRUE` if you want the `ALLOCATE` procedure to terminate the process on error. If `Check` is set to `FALSE` the pointer will be set to `NIL` if the allocation fails.

Near and Far Variants

There are two versions of each of the procedures in this module, the Far version and the Near version. The Far versions return far pointers and the Near versions return near pointers. (See “*The TopSpeed Developer’s Guide*” for further information on the difference between the near and far heap).

Under OS/2 the following procedures are not available and you should use the `ShtHeap` module instead:

```
FarMakeHeap  
FarHeapAllocate  
FarHeapDeallocate  
FarHeapAvail  
FarHeapTotalAvail  
FarHeapChangeSize  
FarHeapChangeAlloc
```

In previous versions of TopSpeed Modula-2, these procedures did not have the Far or Near prefix. In order to maintain compatibility, the TopSpeed Modula-2 alias facility is used in `Storage.DEF` to equate the names of the

procedures without the prefixes to the appropriate procedures with the prefixes for the memory model in question. This is the reason for the generic name given first in each description and used in the descriptions.

In addition, knowledge of the `heap_size` data pragma is recommended when considering these functions. (See “*The TopSpeed Developer’s Guide*” for a full description).

Storage Reference

The following are the individual procedures defined within the Storage module.

ALLOCATE — Allocate memory from the heap

```
PROCEDURE ALLOCATE( VAR a: ADDRESS;  
                    size: CARDINAL );
```

The ALLOCATE procedure allocates size bytes from the heap and returns a pointer to the first byte of the block in a. If there is not enough memory then the program will terminate with a Heap Overflow run-time error. If Check is set to FALSE, the program will not terminate, and a will be set to NIL.

NearAllocate — Allocate memory from the near heap

```
PROCEDURE NearAllocate(  
    VAR a: NearADDRESS;  
    size: CARDINAL);
```

The NearAllocate procedure allocates size bytes from the near heap and returns a pointer to the first byte of the block in a. If there is not enough memory then the program will terminate with a Heap Overflow run-time error. If Check is set to FALSE, the program will not terminate, and a will be set to NearNIL.

The maximum number of bytes that can be allocated cannot exceed 65536 bytes.

FarAllocate — Allocate memory from the far heap

```
PROCEDURE FarAllocate(  
    VAR a: FarADDRESS;  
    size: CARDINAL);
```

The FarAllocate procedure allocates size bytes from the far heap and returns a pointer to the first byte of the block in a. If there is not enough memory then the program will terminate with a Heap Overflow run-time error. If Check is set to FALSE, the program will not terminate, and a will be set to FarNIL.

Available — Is there enough memory?

PROCEDURE Available(size: CARDINAL): BOOLEAN;

The Available procedure returns TRUE if there is a block of at least size available in the heap. This procedure can be used to make sure that ALLOCATE will be able to deliver a block of memory.

NearAvailable — Is there enough memory?

PROCEDURE NearAvailable(
size: CARDINAL): BOOLEAN;

The NearAvailable procedure returns TRUE if there is a block of at least size available in the near heap. This procedure can be used to make sure that NearALLOCATE will be able to deliver a block of memory.

FarAvailable — Is there enough memory?

PROCEDURE FarAvailable(
size: CARDINAL): BOOLEAN;

The FarAvailable procedure returns TRUE if there is a block of at least size available in the far heap. This procedure can be used to make sure that FarALLOCATE will be able to deliver a block of memory.

DEALLOCATE — Return memory to the heap

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

The DEALLOCATE procedure returns previously allocated blocks of memory to the heap. In order to prevent further access to the memory block, the pointer a is set to NIL by this procedure.

NearDeallocate — Return memory to the near heap

```
PROCEDURE NearDeallocate(  
    VAR a: NearADDRESS;  
    size: CARDINAL);
```

The NearDeallocate procedure returns previously allocated blocks of memory to the near heap. In order to prevent further access to the memory block, the pointer a is set to NearNIL by this procedure.

FarDeallocate — Return memory to the far heap

```
PROCEDURE FarDeallocate(  
    VAR a: NearADDRESS;  
    size: CARDINAL );
```

The FarDeallocate procedure returns previously allocated blocks of memory to the far heap. In order to prevent further access to the memory block, the pointer a is set to FarNIL by this procedure.

HeapAllocate — Allocate memory from a heap

```
PROCEDURE HeapAllocate(  
    Source: HeapRecPtr;  
    VAR a: ADDRESS;  
    Size: CARDINAL );
```

This procedure allocates Size units from the allocated heap Source. The address of the allocated memory is placed in a. The heap, Source, must have been previously created with the MakeHeap procedure.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

If there is not enough space left in the heap, your program will be terminated with a Heap Overflow run-time error. However, if Check is set to FALSE, the program will terminate, and a will be set to NIL.

Under OS/2 this function is only available in small and medium models.

NearHeapAllocate — Allocate memory from near heap

```
PROCEDURE NearHeapAllocate(  
    Source: NearHeapRecPtr;  
    VAR a: NearADDRESS;  
    Size: CARDINAL );
```

This procedure allocates Size bytes from the allocated near heap Source. The address of the allocated memory is placed in a. The heap, Source, must have been previously created with the NearMakeHeap procedure.

If there is not enough space left in the heap, your program will be terminated with a Heap Overflow run-time error. However, if Check is set to FALSE, the program will terminate, and a will be set to NearNIL.

FarHeapAllocate — Allocate memory from far heap

```
PROCEDURE FarHeapAllocate(  
    Source: FarHeapRecPtr;  
    VAR a: FarADDRESS;  
    Size: CARDINAL);
```

This procedure allocates Size paragraphs from the allocated far heap Source. The address of the allocated memory is placed in a. The heap, Source, must have been previously created with the FarMakeHeap procedure.

If there is not enough space left in the heap, your program will be terminated with a Heap Overflow run-time error. However, if Check is set to FALSE, the program will terminate, and a will be set to FarNIL.

Under OS/2 this function is not available.

HeapAvail — Largest available block

PROCEDURE HeapAvail(
Source: HeapRecPtr): CARDINAL;

This procedure returns the size, in units, of the largest block remaining on the allocated heap Source. This procedure should be used to find out if the heap contains enough space to allocate from.

The heap, Source, must have been previously created using the procedure MakeHeap.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearHeapAvail — Largest available block

PROCEDURE NearHeapAvail(
Source: NearHeapRecPtr
): CARDINAL;

This procedure returns the size, in bytes, of the largest block remaining on the allocated heap Source. This procedure should be used to find out if the heap contains enough space to allocate from.

The heap, Source, must have been previously created using the procedure NearMakeHeap.

FarHeapAvail — Largest available block

PROCEDURE FarHeapAvail(
Source: FarHeapRecPtr
): CARDINAL;

This procedure returns the size, in paragraphs, of the largest block remaining on the allocated heap Source. This procedure should be used to find out if the heap contains enough space to allocate from.

The heap, Source, must have been previously created using the procedure FarMakeHeap.

Under OS/2 this function is not available.

HeapChangeAlloc — Change size of allocation without relocation

```
PROCEDURE HeapChangeAlloc(  
    Source: HeapRecPtr;  
    VAR a: ADDRESS;  
    OldSize,  
    NewSize: CARDINAL): BOOLEAN;
```

This procedure, like `HeapChangeSize` attempts to change the allocated size of the memory area `a`, from `OldSize` to `NewSize`. Unlike `HeapChangeSize`, however, this procedure will not relocate the block in memory, it will alter the size, if possible, by shrinking or growing the area. If this cannot be accomplished, these procedures return `FALSE`.

The memory block `a` must have been previously allocated from the heap `Source`.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearHeapChangeAlloc — Change size of allocation without relocation

```
PROCEDURE NearHeapChangeAlloc(  
    Source: NearHeapRecPtr;  
    VAR a: NearADDRESS;  
    OldSize,  
    NewSize: CARDINAL): BOOLEAN;
```

This procedure, like `NearHeapChangeSize` attempts to change the allocated size of the memory area `a`, from `OldSize` to `NewSize`. Unlike `NearHeapChangeSize`, however, this procedure will not relocate the block in memory, it will alter the size, if possible, by shrinking or growing the area. If this cannot be accomplished, these procedures return `FALSE`.

The memory block `a` must have been previously allocated from the heap `Source`.

FarHeapChangeAlloc — Change size of allocation without relocation

```
PROCEDURE FarHeapChangeAlloc(  
    Source: FarHeapRecPtr;  
    VAR a: FarADDRESS;  
    OldSize,  
    NewSize: CARDINAL): BOOLEAN;
```

This procedure, like `HeapChangeSize` attempts to change the allocated size of the memory area `a`, from `OldSize` to `NewSize`. Unlike `HeapChangeSize`, however, this procedure will not relocate the block in memory, it will alter the size, if possible, by shrinking or growing the area. If this cannot be accomplished, these procedures return `FALSE`.

The memory block `a` must have been previously allocated from the heap `Source`.

Under OS/2 this function is only available in small and medium models.

HeapChangeSize — Change size of allocation

```
PROCEDURE HeapChangeSize(  
    Source: HeapRecPtr;  
    VAR a: ADDRESS;  
    OldSize,  
    NewSize: CARDINAL);
```

This procedure attempts to change the size of the allocated block `a` from the heap `Source` from `OldSize` bytes to `NewSize` bytes. It will first attempt to do this without moving the block in memory. If this is not possible, it will move the block to a place on the heap and adjust the value of `a` accordingly.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearHeapChangeSize — Change size of allocation

```
PROCEDURE NearHeapChangeSize(  
    Source: NearHeapRecPtr;  
    VAR a: NearADDRESS;  
    OldSize,  
    NewSize: CARDINAL);
```

This procedure attempts to change the size of the allocated block a from the heap Source from OldSize bytes to NewSize bytes. It will first attempt to do this without moving the block in memory. If this is not possible, it will move the block to a place on the heap and adjust the value of a accordingly.

FarHeapChangeSize — Change size of allocation

```
PROCEDURE FarHeapChangeSize(  
    Source: FarHeapRecPtr;  
    VAR a: FarADDRESS;  
    OldSize,  
    NewSize: CARDINAL);
```

This procedure attempts to change the size of the allocated block a from the heap Source from OldSize bytes to NewSize bytes. It will first attempt to do this without moving the block in memory. If this is not possible, it will move the block to a place on the heap and adjust the value of a accordingly.

Under OS/2 this function is not available.

HeapDeallocate — Free memory back to a heap

```
PROCEDURE HeapDeallocate(  
    Source: HeapRecPtr;  
    VAR a: ADDRESS;  
    Size: CARDINAL );
```

This procedure returns a block of memory, a, to the heap Source. This memory then becomes available for re-use. Size specifies the size of the block.

The procedure will check to see if the memory released can be combined with other contiguous free areas on either side of the freed block. If it can, they will be combined. This means that small blocks will be recombined to make larger ones if possible.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearHeapDeallocate — Free memory back to a heap

```
PROCEDURE NearHeapDeallocate(  
    Source: NearHeapRecPtr;  
    VAR a: NearADDRESS;  
    Size: CARDINAL);
```

This procedure returns a block of memory, a, to the heap Source. This memory then becomes available for re-use. Size specifies the size of the block.

The procedure will check to see if the memory released can be combined with other contiguous free areas on either side of the freed block. If it can, they will be combined. This means that small blocks will be recombined to make larger ones if possible.

FarHeapDeallocate — Free memory back to a heap

```
PROCEDURE FarHeapDeallocate(  
    Source: FarHeapRecPtr;  
    VAR a: FarADDRESS;  
    Size: CARDINAL);
```

This procedure returns a block of memory, a, to the heap Source. This memory then becomes available for re-use. Size specifies the size of the block.

The procedure will check to see if the memory released can be combined with other contiguous free areas on either side of the freed block. If it can, they will be combined. This means that small blocks will be recombined to make larger ones if possible.

Under OS/2 this function is not available.

HeapTotalAvail — Total available memory on heap

```
PROCEDURE HeapTotalAvail(  
    Source: HeapRecPtr): CARDINAL;
```

This procedure returns the total number of units available on the heap Source. Note that this memory may not be contiguous and the value returned does not indicate that it is possible to allocate a block of that size from the heap. To find the largest block available, use the HeapAvail procedure.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearHeapTotalAvail — Total available memory on heap

```
PROCEDURE NearHeapTotalAvail(  
    Source : NearHeapRecPtr ) : CARDINAL;
```

This procedure returns the total number of bytes available on the heap Source. Note that this memory may not be contiguous and the value returned does not indicate that it is possible to allocate a block of that size from the heap. To find the largest block available, use the NearHeapAvail procedure.

FarHeapTotalAvail — Total available memory on heap

```
PROCEDURE FarHeapTotalAvail(  
    Source : FarHeapRecPtr ) : CARDINAL;
```

This procedure returns the total number of paragraphs available on the heap Source. Note that this memory may not be contiguous and the value returned does not indicate that it is possible to allocate a block of that size from the heap. To find the largest block available, use the FarHeapAvail procedure.

Under OS/2 this function is not available.

MakeHeap — Create a sub-heap

```
PROCEDURE MakeHeap(  
    Source : CARDINAL;  
    Size: CARDINAL): HeapRecPtr;
```

This procedure is used to define a new heap. The type HeapRecPtr is an opaque type used to hold information about the heap. Source is the segment where the heap is to be located, and must be the segment of an area of memory allocated by some other method (e.g. ALLOCATE) because MakeHeap does not allocate memory. Size is the size in units of the memory area.

In small and medium models the unit of allocation is bytes.

In all other models the unit of allocation is paragraphs.

Under OS/2 this function is only available in small and medium models.

NearMakeHeap — Create a sub-heap

```
PROCEDURE NearMakeHeap(  
    Source: CARDINAL;  
    Size: CARDINAL;  
    NearHeapRecPtr;
```

This procedure is used to define a new heap. The type NearHeapRecPtr is an opaque type used to hold information about the heap. Source is a pointer to where the heap is to be located, and must be a pointer to an area of memory allocated by some other method (e.g. NearALLOCATE) because NearMakeHeap does not allocate memory. Size is the size in units of the memory area.

FarMakeHeap — Create a sub-heap

```
PROCEDURE FarMakeHeap(  
    Source: CARDINAL;  
    Size: CARDINAL): FarHeapRecPtr;
```

This procedure is used to define a new heap. The type `FarHeapRecPtr` is an opaque type used to hold information about the heap. `Source` is the segment where the heap is to be located, and must be the segment of an area of memory allocated by some other method (e.g. `FarALLOCATE`) because `FarMakeHeap` does not allocate memory. `Size` is the size in paragraphs of the memory area.

A paragraph is a block of memory bytes in size.

Under OS/2 this function is not available.

SegAllocate — Allocate a segment

```
PROCEDURE SegAllocate(  
    Size: CARDINAL): CARDINAL;
```

The procedure allocates a segment of `Size` bytes from the far heap. The value returned depends on the operating system at run-time:

OS/2 A selector is returned.

DOS A segment value is returned.

If the request fails, the return value will be 0.

SegDeallocate — Deallocate a segment

```
PROCEDURE SegDeallocate(  
    Sel: CARDINAL;  
    Size: CARDINAL): CARDINAL;
```

The procedure deallocates a segment of `Size` bytes previously allocated by `SegAllocate`.

MODULE Str

Introduction

The procedures in this module allow you to control and manipulate character strings within TopSpeed Modula-2.

Procedures implemented include a wide range of string handling functions, including string manipulation and conversions between strings and numbers.

Strings in Modula-2

Modula-2 does not have any built-in string handling and there is no simple string type. Strings are implemented as the type `ARRAY [0..N] OF CHAR`. When a string literal is assigned to such a type it will be zero terminated by appending a character of value `CHR(0)` after the last visible character of the literal. This will always be the case unless the length of the literal is equal to the maximum size of the array.

All the procedures in this module that return a string (using a `VAR`) parameter will zero terminate the result unless the maximum size of the receiving array is equaled or exceeded. If the result is greater than the maximum size of the receiving array, the result will be truncated to fit.

The `Str` module regards the first character of a string as being in position zero as it regards the character array as having a lower bound of zero. Thus the second character is 1, the third 2, etc.

In order to simplify the examples, all identifiers beginning `str` should be regarded as having been defined as some `ARRAY OF CHAR`.

Definition File

The types defined in `Str.DEF` are:

```
TYPE
  CHARSET = SET OF CHAR;
  PosLen  = RECORD
    Pos, Len : CARDINAL;
  END;
```

The `CHARSET` type is used by the `Item` procedure to define the delimiters for the sub-field scan.

For details of the use of `PosLen`, see the `FindSubStr` procedure.

Conversion Procedures

The Str module contains a number of routines for converting strings to numbers and vice versa. These are:

IntToStr	StrToInt
CardToStr	StrToCard
RealToStr	StrToReal
FixRealToStr	

When converting from a string to a number the string may begin with a '+' or '-', however '-' is illegal for CARDINAL numbers. No leading or trailing spaces are allowed in the string.

A string resulting from a conversion will only contain a sign if the number was negative. All the conversion procedures have a BOOLEAN result parameter, OK. This is set TRUE if the conversion succeeded or FALSE otherwise.

When converting from a string to an integer or a cardinal, there is a Base parameter which allows you to choose the base (normally 10) used in the conversion. Base can have a value between 2 and 16 inclusive. The letters 'A' to 'F' are used to represent digits when the base is larger than 10. No 'H' should be added after hexadecimal numbers.

Str Reference

The following are the individual procedures defined within the Str module.

Append — Append a string

```
PROCEDURE Append(  
    VAR R: ARRAY OF CHAR;  
    S: ARRAY OF CHAR);
```

Appends the string S to the string R. If the combined length of the two strings is greater than the maximum length of R then only as many character from S as will fit are appended. For example:

```
str1 := 'ex';  
str2 := 'ample';  
Str.Append(str1,str2);  
(* str1 is now 'example')
```

Caps — Convert to upper case

```
PROCEDURE Caps( VAR S : ARRAY OF CHAR );
```

Converts all lower case letters in S to upper case. All other characters remain unchanged. For example:

```
str1 := 'This is a ***** string.';  
Str.Caps(str1); (* str1 is now  
                'THIS IS A ***** STRING.' *)
```

CardToStr — Convert Cardinal to String

```
PROCEDURE CardToStr(  
    V: LONGCARD;  
    VAR S: ARRAY OF CHAR;  
    Base: CARDINAL;  
    VAR OK: BOOLEAN);
```

Converts the LONGCARD value V into a string representation in the specified Base. OK is set to FALSE if S is too short to hold the resulting string. For example:

```
VAR  
    str1 : ARRAY [0..20] OF CHAR;  
    Done : BOOLEAN;  
...  
Str.CardToStr(1234567,str1,8,Done);  
(* str1 = "4553207" *)  
Str.CardToStr(1234567,str1,10,Done);  
(* str1 = "1234567" *)  
Str.CardToStr(1234567,str1,16,Done);  
(* str1 = "12D687" *)
```

CharPos — Find character in string

```
PROCEDURE CharPos(  
    S: ARRAY OF CHAR;  
    C: CHAR): CARDINAL;
```

CharPos returns position of first occurrence of character C in string S. If the character is not found it returns MAX(CARDINAL). This procedure is much faster than Pos when attempting to locate single characters. The procedure RCharPos is similar but searches S from the end. For example:

```
i := CharPos("abcabc",'c'); (* i = 2 *)
```

Compare — Compare two strings

```
PROCEDURE Compare(  
    S1, S2: ARRAY OF CHAR  
): INTEGER;
```

Compares two strings. The function compares the constituent characters of the strings S1 and S2 from left to right until a difference is found or the end of both strings is reached simultaneously.

If the two strings are identical, the procedure returns 0. If S1 is less than S2 then it returns -1, otherwise 1. For example:

```
VAR  
    i : INTEGER;  
...  
i := Str.Compare('hello','goodbye');  
(* i = 1 as 'hello' comes after 'goodbye' *)  
...  
i := Str.Compare('Hello','halt');  
(* i = -1 as 'H' comes before 'h' *)
```

Concat — Join two strings

```
PROCEDURE Concat(  
    VAR R: ARRAY OF CHAR;  
    S1, S2 : ARRAY OF CHAR);
```

This procedure concatenates S1 and S2 and returns the result in R. The strings S1 and S2 are unaffected by this operation. If the length of the result is too large to fit into R, then the second string (S2) is truncated. For example:

```
str1 := 'hello,';  
str2 := ' world';  
str3 := '***';  
Str.Concat(str3,str1,str2);  
(* str3 is now 'hello, world' *)
```

Copy — Copy a string

```
PROCEDURE Copy(  
    VAR R: ARRAY OF CHAR;  
    S: ARRAY OF CHAR);
```

This procedure copies the string S into R. If S is too long to fit into R then the result is truncated. For example:

```
str1 := 'new';  
str2 := 'version';  
Str.Copy(str2, str1);  
(* str2 now = 'new' *)
```

Delete — Delete character from a string

```
PROCEDURE Delete(  
    VAR R: ARRAY OF CHAR;  
    P, L : CARDINAL);
```

This procedure deletes the sequence of L characters in the string R starting with the character at position P. Delete has no effect if P is greater than Length(R). For example:

```
str1 := "abcdefg";  
Str.Delete(str1, 3, 3); (* str1 now = "abcg" *)  
  
str2 := "abcdefg";  
Str.Delete(str2, 3, 7); (* str2 now = "abc" *)  
  
str3 := "abcdefg";  
Str.Delete(str3, 7, 3); (* str3 does not change *)
```

FindSubStr — Find all matches

```
PROCEDURE FindSubStr(
    Source, Pattern: ARRAY OF CHAR;
    VAR pos ARRAY OF PosLen
): BOOLEAN;
```

The FindSubStr procedure is an extension of the Match procedure. FindSubStr builds a table of the item in Source that matches the wildcard pattern given by Pattern. The table consists of an ARRAY OF PosLen. The PosLen record contains two fields Pos and Len which indicate, for each wildcard, where each pattern match occurred (Pos) and the length of the matching substring (Len).

FindSubStr returns TRUE if a match was found and FALSE when there are no matches to the pattern. For example:

```
VAR
    Table : ARRAY [1..10] OF Str.PosLen;
    Matched : BOOLEAN;
    Match := Str.FindSubstr("Good evening!", "e?e*g", Table);
```

Match will be TRUE after this last statement and Table will contain two entries:

Index	Pos	Len	Matched with
1	6	1	"v"
2	8	3	"nin"

FixRealToStr — Convert Real to String without exponent

```
PROCEDURE FixRealToStr(
    V: LONGREAL;
    Precision: CARDINAL;
    VAR S: ARRAY OF CHAR;
    VAR OK: BOOLEAN);
```

RealToStr produces a string representation of the LONGREAL value V in S. No exponent is produced by this conversion function. The Precision parameter denotes the number of digits you want after the decimal point.

OK will be set to FALSE if either S is not large enough to hold the result, or if ABS(V) is larger than 1.08E18. For example:

```
Str.FixRealToStr(0.12345, 7, str1, Done);
    (* str1 = "0.1234500" *)
Str.FixRealToStr(123.4567, 5, str1, Done);
    (* str1 = "123.45670" *)
Str.FixRealToStr(123.4567, 2, str1, Done);
    (* str1 = "123.46" *)
```

Insert — Insert characters into string

```
PROCEDURE Insert(
    VAR R: ARRAY OF CHAR;
    S: ARRAY OF CHAR;
    P: CARDINAL);
```

This procedure inserts the string S into the string R starting at position P. If P is greater than Length(R), then S is inserted at position Length(R). For example:

```
str1 := "abcdef";
Str.Insert(str1,"aabbcc",3);
(* str1 now = "abcaabbccdef" *)
```

IntToStr — Convert Integer to String

```
PROCEDURE IntToStr(
    V: LONGINT;
    VAR S: ARRAY OF CHAR;
    Base: CARDINAL;
    VAR OK: BOOLEAN);
```

Converts the LONGINT value V into a string representation in the specified Base. OK is set to FALSE if S is too short to hold the resulting string. For example:

```
VAR
    str1 : ARRAY [0..20] OF CHAR;
    Done : BOOLEAN;
...
Str.IntToStr(-1234567,str1,8,Done);
(* str1 = "-4553207" *)
Str.IntToStr(-1234567,str1,10,Done);
(* str1 = "-1234567" *)
Str.IntToStr(-1234567,str1,16,Done);
(* str1 = "-12D687" *)
```

Item — Extract sub-field

```
PROCEDURE Item(
    VAR R : ARRAY OF CHAR;
    S : ARRAY OF CHAR;
    T : CHARSET;
    N : CARDINAL );
```

The Item procedure treats the string S as a series of non-blank substrings separated by elements from T. It returns the Nth of these substrings (counting from 0) in the array R. For example:

```
str1 := "item0,item1,$item2 *item3* item4";
...
Str.Item(str2,str1,CHARSET{' ', ','},2);
(* str2 is assigned the string '$item2' *)
...
Str.Item(str3,str1,CHARSET{'$', ' '},0);
(* str3 is assigned the string 'item0,item1' *)
```

ItemS — Extract subfield

```
PROCEDURE ItemS(
    VAR R: ARRAY OF CHAR;
    S: ARRAY OF CHAR;
    T: ARRAY OF CHAR;
    N: CARDINAL);
```

This procedure is identical in function to Item. The only difference is that the delimiting characters (T) are here passed as an array rather than as a CHARSET. For example:

```
TYPE
    SmallArray = ARRAY [0..3] OF CHAR;
CONST
    Delims = SmallArray( CHR(10), CHR(13), CHR(26), ' ' );
VAR
    str1, str2 : ARRAY [0..40] OF CHAR;
...
str1 := 'aa bb cc dd';
Str.ItemS(str2,str1,Delims,3);
(* str2 now has the value 'dd' *)
```

Length — Length of string

PROCEDURE Length(S: ARRAY OF CHAR): CARDINAL;

This procedure returns the actual number of characters in a string, not including the zero terminator, if present. For example:

```
VAR
  i : CARDINAL;
...
i := Str.Length('Mother'); (* i = 6 *)
```

Match — Wildcard string match

PROCEDURE Match(
 Source,
 Pattern: ARRAY OF CHAR
): BOOLEAN;

The Match procedure returns TRUE if the string Source matches the wildcard pattern in the string Pattern. The Pattern string uses '?' and '*' as wildcard characters:

?	matches any single character at that position in the string
*	matches any sequence of characters (including none at all)

All other characters are treated literally and must match exactly, except that case is ignored. For example:

```
VAR
  IsMatched : BOOLEAN;
...
IsMatched := Str.Match("Automatic","*M?t*i*");
(* IsMatched is set TRUE *)

IsMatched := Str.Match("automatic","m?t*i*");
(* IsMatched is set FALSE *)
```

NextPos — Locate next substring

```
PROCEDURE Pos(
    S, P: ARRAY OF CHAR;
    Place: CARDINAL): CARDINAL;
```

NextPos is identical to Pos in that it returns the starting position of the substring P within S or MAX(CARDINAL) if not found. The only difference is that the search begins at Place within P and not at the start of P. For example:

```
VAR
    i : CARDINAL;
...
i := Str.NextPos("zabcdabxy", "ab", 3);
(* i = 5 *)
i := Str.NextPos("zabcdabxy", "ab", 0);
(* i = 1 *)
(*
    this is identical to
    Str.Pos("zabcdabxy", "ab")
*)
```

Pos — Locate substring

```
PROCEDURE Pos(
    S, P: ARRAY OF CHAR): CARDINAL;
```

Returns the position (with 0 as the first position) of the first occurrence of the substring P in the string S. If P cannot be found in S then the procedure returns MAX(CARDINAL). For example:

```
VAR
    i : CARDINAL;
...
i := Str.Pos('example', 'amp'); (* i = 2 *)
i := Str.Pos('source string', 'not in it');
(* i = 65535 *)
```

Prepend — Prepend string

```
PROCEDURE Prepend(
    VAR S1: ARRAY OF CHAR;
    S2: ARRAY OF CHAR);
```

This procedure is the complement to Append which adds one string on to the end of another. Prepend prefixes the string S2 to the string S1. For example:

```
str1 := "world";
Str.Prepend(str1, "Hello, ");
(* str1 now = "Hello, world" *)
```

RCharPos — Find character in string from right

```
PROCEDURE RCharPos(  
    S: ARRAY OF CHAR;  
    C: CHAR): CARDINAL;
```

RCharPos returns position of first occurrence from the right of character C in string S. If the character is not found it returns MAX(CARDINAL). The procedure CharPos is similar but searches S from the beginning. For example:

```
i := RCharPos("abcabc",'c');(* i = 5 *)
```

RealToStr — Convert Real to String

```
PROCEDURE RealToStr(  
    V: LONGREAL;  
    Precision: CARDINAL;  
    Eng: BOOLEAN;  
    VAR S: ARRAY OF CHAR;  
    VAR OK: BOOLEAN);
```

RealToStr produces a string representation of the LONGREAL value V in S. The Precision parameter denotes the number of digits you want in the mantissa and it must lie in the range 1 to 17, inclusive. If Precision is out of this range then its value will be adjusted to the nearest boundary of the range.

If Eng is TRUE then engineering notation is used, i.e., the exponent is represented as a multiple of three. If it is FALSE then the mantissa is represented with one significant digit before the decimal point. For example:

```
Str.RealToStr(123.45,7,FALSE,str1,Done);  
    (* str1 = "1.234500E+2" *)  
Str.RealToStr(0.12345,7,FALSE,str1,Done);  
    (* str1 = "1.234500E-1" *)  
Str.RealToStr(0.12345,7,TRUE,str1,DONE);  
    (* str1 = "123.4500E-3" *)
```

Slice — Extract substring

PROCEDURE Slice(

VAR R: ARRAY OF CHAR;
S: ARRAY OF CHAR;
P, L: CARDINAL);

R is assigned a slice of the string S. This slice goes from position P to position $P + L - 1$, inclusive. If $P + L - 1$ is greater than the length of S, then R is assigned the slice from P to Length(S). If P is greater than Length(S) or L is zero, then R is assigned the null string (i.e., a string of length zero). For example:

```
Str.Slice(str1,'overnight',2,4);  
(* str1 = 'ern' *)  
Str.Slice(str1,'overnight',2,12);  
(* str1 = 'ernight' *)  
Str.Slice(str1,'overnight',12,2);  
(* str1 = null string *)
```

StrToC — Convert string to C

PROCEDURE StrToC(

S: ARRAY OF CHAR;
VAR D: ARRAY OF CHAR): BOOLEAN

This procedure converts a Modula-2 string S to a zero terminated string in D. The resulting string D is suitable for passing to a C or operating system function.

StrToCard — Convert String to Cardinal

```
PROCEDURE StrToCard(
    S: ARRAY OF CHAR;
    Base: CARDINAL;
    VAR OK: BOOLEAN): LONGCARD;
```

StrToCard takes the string S and attempts to convert it to a LONGCARD that will be returned by the procedure. Base represents the numeric base of the digits currently in string form. OK will be set to FALSE if the string does not represent a LONGCARD value. For example:

```
VAR
    Lcrd : LONGCARD;
    Done : BOOLEAN;
...
Lcrd := Str.StrToCard("4553207",8,Done);
(* Lcrd = -1234567 *)
Lcrd := Str.StrToCard("1234567",10,Done);
(* Lcrd = -1234567sz *)
Lcrd := Str.StrToCard("12D687",16,Done);
(* Lcrd = -1234567 *)
```

StrToInt — Convert String to Integer

```
PROCEDURE StrToInt(
    S: ARRAY OF CHAR;
    Base: CARDINAL;
    VAR OK: BOOLEAN): LONGINT;
```

StrToInt takes the string S and attempts to convert it to a LONGINT that will be returned by the procedure. Base represents the numeric base of the digits currently in string form. OK will be set to FALSE if the string does not represent a LONGINT value. For example:

```
VAR
    Lint : LONGINT;
    Done : BOOLEAN;
...
Lint := Str.StrToInt("-4553207",8,Done);
(* Lint = -1234567 *)
Lint := Str.StrToInt("-1234567",10,Done);
(* Lint = -1234567 *)
Lint := Str.StrToInt("-12D687",16,Done);
(* Lint = -1234567 *)
```

StrToPas — Convert string to Pascal

```
PROCEDURE StrToPas (
    S: ARRAY OF CHAR;
    VAR D: ARRAY OF CHAR): BOOLEAN
```

This procedure converts a Modula-2 in S string to a Pascal dynamic string in D. The resulting string in D is suitable only for passing to Pascal functions.

StrToReal — Convert String to Real

```
PROCEDURE StrToReal(
    S: ARRAY OF CHAR;
    VAR OK: BOOLEAN): LONGREAL;
```

StrToReal takes a string, S, representing a real value and attempts to convert it to the corresponding LONGREAL value. The syntax for a valid REAL number is given in Chapter 2. OK will be set to FALSE if the string does not represent a real value. For example:

```
VAR
    Lreal : LONGREAL;
    Done : BOOLEAN;
...
Lreal := Str.StrToReal("455.3207",Done);
(* Lreal = 4.553207E+02 *)
Lreal := Str.StrToReal("123456e-7",Done);
(* Lreal = 1.23456E-2 *)
Lreal := Str.StrToReal("-0.1234567",Done);
(* Lreal = -1.234567E-1 *)
```

Subst — Replace substring

```
PROCEDURE Subst(
    VAR S1: ARRAY OF CHAR;
    Target: ARRAY OF CHAR;
    New: ARRAY OF CHAR);
```

The Subst procedure searches string S1 and replaces the first occurrence of Target that it finds with New. If Target is not found no substitution takes place. For example:

```
str1 := "Say hello to me";
Str.Subst(str1,"hello","goodbye");
(* str1 now = "Say goodbye to me" *)
```

MODULE SYSTEM

Introduction

The SYSTEM module defines a large number of system dependent types and constants that are, in fact, built-in to the TopSpeed Modula-2 compiler. This section only describes the extra features designed to aid machine-specific programming on the 80x86 family of microprocessors.

The Registers Record

The SYSTEM module defines a record type, Registers, that is used to hold values corresponding to the actual machine registers of the processor. Note that these are not the registers themselves but a convenient structure to pass values to operating system functions and return values.

```

TYPE
  Registers = RECORD
    CASE : BOOLEAN OF
      | TRUE :
        AX,BX,CX,DX,BP,SI,DI,DS,ES: CARDINAL;
        Flags: BITSET;
      | FALSE :
        AL,AH,BL,BH,CL,CH,DL,DH: SHORTCARD;
    END;
  END;
```

Coroutines

A coroutine is a sequential process that can be suspended by transferring execution to another coroutine. Each coroutine “remembers” its previous state so that it can resume, where it left off, when control is transferred back to it.

As the IBM PC/AT and compatibles are single-processor machines, that cannot carry out truly concurrent processes. Coroutines, however, allow them to simulate this behavior.

Coroutines are implemented in TopSpeed Modula-2 by using the TRANSFER and IOTRANSFER procedures defined in this module. These are used together with the module priority feature of Modula-2 to allow interrupt driven processes to be written.

Under OS/2 only interrupt 8 (the timer interrupt) can be used in this way. Under DOS, all the interrupts are available.

Each coroutine, or process, is initiated using the NEWPROCESS procedure. Once the process has been initiated, control can be passed to it by using the

TRANSFER procedure. The process will then do some work and, probably, use TRANSFER to pass control to some other process. At a later time, the original process will receive control again, in exactly the same state as it was before it gave up control.

Further details can be found in the TRANSFER and IOTRANSFER procedure descriptions, below.

Low-level Procedures

The SYSTEM module also provides a number of procedures that allow you carry out hardware specific, low-level actions. These should be used with caution and with the full awareness that they are not transferable between different operating systems and hardware configurations.

The procedures In, InW, Out and OutW described below, must reside in a segment with I/O privilege when they are used in a program running under OS/2. This is achieved with the aid of the iopl pragma:

```
(*# save, call( iopl => yes ) *)  
  
... (* code requiring I/O privilege *)  
  
(*# restore *)
```

SYSTEM Reference

The following are the individual procedures defined within the SYSTEM module.

DI — Disable interrupts

PROCEDURE DI();

This procedure disables interrupts, preventing the calling process from being interrupted. It is useful when data is being global shared by several interrupt driven processes.

This procedure is not available under OS/2.

EI — Enable interrupts

PROCEDURE EI();

This procedure allows interrupts to occur again. It is used in conjunction with DI to enclose those parts of a process that must not be interrupted, such as procedures that access shared global data.

This procedure is not available under OS/2.

GetFlags — Read processor flags

PROCEDURE GetFlags() : CARDINAL;

This procedure returns the 80x86 flag register.

In — Read byte value from port

PROCEDURE In(p : CARDINAL) : SHORTCARD;

This procedure reads a byte value from the I/O port p and returns it.

InW — Read word value from port

PROCEDURE InW(p : CARDINAL) : CARDINAL;

This procedure reads a word value from the I/O port p and returns it.

IOTRANSFER — Transfer to I/O co-process

PROCEDURE IOTRANSFER(
 VAR P1, P2: FarADDRESS;
 I: CARDINAL);

The IOTRANSFER procedure is used to associate the current process with an processor interrupt. The interrupt number is specified in I. A call to IOTRANSFER causes the current process to be suspended and its process reference is placed in P1. Then control is passed to the process specified by P2.

P2 must be the result of a previous call to NEWPROCESS or TRANSFER.

At some later time, when the CPU receives notification of an interrupt associated with the interrupt vector I, the currently active process will be suspended and its reference placed in P2. The process that called IOTRANSFER is then reactivated and its reference placed in P1. In this way, the caller of IOTRANSFER can respond to asynchronous demands for its services that are driven only by the occurrence of the required interrupt. For example:

```
VAR  
  ProcRef1,  
  ProcRef2: FarADDRESS;  
...  
  IOTRANSFER(ProcRef1,ProcRef2,0BH);
```

This causes the current process to be associated with interrupt vector 0BH (IRQ 3). The current process is suspended and its reference placed in ProcRef1. Then the process associated with ProcRef2 is activated.

If, at some later time, interrupt 0BH occurs, then whatever process is then active will be suspended. Its state will be saved and its reference placed in ProcRef2.

The process ProcRef1 is then reactivated to carry out the service of the interrupt.

NEWPROCESS — Start new process

```
PROCEDURE NEWPROCESS(
    P: PROC;
    A: FarADDRESS;
    S: CARDINAL;
    VAR P1: FarADDRESS);
```

This procedure is used to initiate new processes (coroutines). P is the name of the procedure that is the entry point to the process. A is a pointer to the workspace for the process. This work space is used for local variables and to store the state of the process when is suspended. S is the size of this workspace in bytes, and should be at least 1Kb.

The procedure returns a reference to the procedure in the variable P1, this reference should be used with TRANSFER and IOTRANSFER.

Note that NEWPROCESS only prepares a process for execution, it does not start it running, a call to TRANSFER is required to start a procedure executing.

Out — Write byte to value port

```
PROCEDURE Out( p : CARDINAL; v : SHORTCARD );
```

This procedure writes a byte value, v, to the I/O port p.

OutW — Write word to value port

```
PROCEDURE OutW( p : CARDINAL; v : CARDINAL );
```

This procedure writes a word value, v, to the I/O port p.

SetFlags — Set processor flags

```
PROCEDURE SetFlags(F: CARDINAL);
```

This procedure loads the 80x86 flag register with the value specified by F. This procedure should be used sparingly, if at all.

TRANSFER — Transfer to co-process

PROCEDURE TRANSFER(VAR P1, P2 : FarADDRESS);

This procedure transfers control from one process to another. The current process is suspended and its reference is assigned to P1. Then the processed referenced by P2 is resumed.

P2 must be the result of a previous call to NEWPROCESS or TRANSFER. The process associated with P1 may be reactivated at some later time by using P1 in another call to TRANSFER.

This type of transfer is know as a synchronous transfer as both the calling (P1) and the called (P2) process stay in step though their calls to TRANSFER. This is in contrast to IOTRANSFER where the process may be resumed as a result of an external event unassociated with the logical flow of the program. For example:

```

TYPE
    MainProcess,
    Process1,
    Process2: FarADDRESS;

    PROCEDURE Action1;
    BEGIN
        ...
        SYSTEM.TRANSFER(Process1,Process2);
    (* Transfer to Process2 (Action2) *)
        ...
    END Action1;
    PROCEDURE Action2;
        ...
        SYSTEM.TRANSFER(Process2,Process1);
    (* Transfer to Process1 (Action1) *)
        ...
    END Action2;
BEGIN
    ...
    SYSTEM.NEWPROCESS(
        Action1,
        Workspace1,
        1024,
        Process1);
    SYSTEM.NEWPROCESS(
        Action2,
        Workspace2,
        1024,
        Process2);
    SYSTEM.TRANSFER(
        MainProcess,
        Process1);
    ...
END;
```

MODULE Window

Introduction

The Window module provides a powerful set of procedures for defining and manipulating screen windows. A window is a rectangular area of the screen into which screen output is placed, as if it were a small screen. The screen can contain many overlapping windows, each receiving output from various parts of your program. The module automatically takes care of the overlaps to ensure that the covering windows don't display the contents of those underneath them.

Terminology

There are a number of terms and phrases used throughout the window module that require some explanation. Rather than define them several times, they are collected together here.

current window	The window that is currently in use. This window need not be the top window, indeed it need not be visible at all. Each process (if you are using the Process module) has its own current window.
top window	The window that is regarded as being "in front of" all the other windows on the screen. It is the only window that is guaranteed to be totally visible.
coordinates	Coordinates are given as (column,row) or (x,y) throughout this section.
absolute coordinates	Coordinates that apply to the entire screen. These are numbered from (0,0) which is the top left-hand corner of the screen.
relative coordinates	Relative coordinates are given relative to the top left-hand corner of the window and start from (1,1).
palette	A set of colors attached to a window. These are an array of records that defined the foreground and background colors. Each color is referred to by a number that is the index of the record in the array.
frame	The border of a window. This may be missing. The characters that make up the border can be any character from the IBM character set you choose. Two standard borders (double and single lines) are defined.
wrapping/clipping	Describes the behaviors of a window when a text string extends beyond its edges. If wrapping is ON, then the extra text is shifted onto the next line and the

window scrolls downwards, if required. If wrapping is OFF, the window simply “throws away” all the excess characters.

Constants and Types

There are a number of general, fairly self-explanatory types and constants used by the Window module:

```

CONST
    ScreenWidth = 80;
    (* Maximum number of columns *)
    ScreenDepth = 25;
    (* Maximum number of rows *)
    CGASnow = FALSE;
    (* Set To TRUE for CGA Snow Check *)

TYPE
    WinType;
    RelCoord = CARDINAL;
    (* Relative coordinates *)
    AbsCoord = CARDINAL;
    (* Absolute coordinates *)
    Color = ((* IBM color set *)
        Black, Blue,
        Green, Cyan,
        Red, Magenta,
        Brown, LightGray,
        DarkGray, LightBlue,
        LightGreen, LightCyan,
        LightRed, LightMagenta,
        Yellow, White );
    TitleStr = ARRAY[0..ScreenWidth-1] OF CHAR;
    TitleMode = (NoTitle, LeftUpperTitle,
        CenterUpperTitle,
        RightUpperTitle,
        LeftLowerTitle,
        CenterLowerTitle,
        RightLowerTitle );

```

Types of Window Procedures

The window procedures can be divided into four classes:

- Window manipulation procedures. These procedures allow you to define and maintain the windows themselves. They also allow you to associate palettes with windows.
- Positioning procedures. These procedures allow you to change the current output position, determine the current position, and so on.
- Contents manipulation procedures. These procedures supplement the IO module to allow you to display information in a window.

- Multi-thread procedure. This single procedure is needed when using the Window module with the Process module.

Window Manipulation Procedures

The window manipulation procedures are:

Open	Define a new window.
PaletteOpen	Open a palette window.
SetPalette	Change a window's palette.
PaletteColor	Get palette.
SetPaletteColor	Change current palette color.
SetTitle	Set the title of a window.
SetFrame	Set the frame of a window.
Use	Change the current window.
PutOnTop	Make a window the top window.
PutBeneath	Reorder the windows on the screen.
Hide	Remove a window from the screen (but don't close it).
Top	Determine the current top window.
Change	Alter a window's size and position.
Close	Close a window for good.
Used	Determined the current window.
PaletteColorUsed	See if a palette color is being used.
Info	Get information on a window.

A window is defined using the Open (or PaletteOpen) procedure. This procedure takes a description of a window, defined with WinDef and returns an opaque type called WinType. You use this WinType value to refer to a window in much the same way as a file handle refers to a file. For this reason, we will call WinType the window handle. The WinDef record is defined as follows:

```
WinDef = RECORD
  X1,Y1,
  X2,Y2: AbsCoord;
  (* outer co-ordinates of opposite corners *)
  Foreground,
  Background: Color;
  (* not used if Palette *)
  CursorOn: BOOLEAN;
  (* if cursor active   *)
```

```

WrapOn: BOOLEAN;
(* if EOL wrap enabled *)
Hidden: BOOLEAN;
(* if window on view *)
rameOn: BOOLEAN;
(* if frame *)
FrameDef: FrameStr;
(* only used if frame *)
FrameFore,
FrameBack: Color;
(* only used if frame and
   not Palette Window *)
END;

```

The FrameStr is a special array describing the frame (and there are two defined frame strings):

```

TYPE
  FrameStr = ARRAY[0..8] OF CHAR;
            (* Characters for frame *)
            (* 0   1   2   *)
            (* 3   4   *)
            (* 5   6   7   *)
CONST
  SingleFrame  = FrameStr('Ú-¿³ Å-Û');
  DoubleFrame  = FrameStr('ÉÍ»ºÉÍ¼');

```

The initialization part of the Window module opens a default window, with no frame, that covers the whole screen. It can be referenced using the window handle FullScreen and has the following definition:

```

CONST
  FullScreenDef = WinDef(
    0,0, ScreenWidth-1,ScreenDepth-1,
    White,Black,TRUE,TRUE,FALSE,FALSE,
    '   ',Black,Black);
VAR
  FullScreen : WinType;

```

Palette windows use some extra definitions:

```

CONST
  PaletteSize = 10;
  PaletteMax  = PaletteSize-1;
  NormalPaletteColor = 0;
  FramePaletteColor  = 1;
TYPE
  PaletteRange  = SHORTCARD [0..PaletteMax];
  PaletteColorDef = RECORD
    Fore,
    Back   : Color
  END;
  PaletteDef = ARRAY PaletteRange OF PaletteColorDef;

```

PaletteColorDef defines a record containing a foreground color and a background color. The palette is a fixed size array containing a number of these. When using

palette windows, the frame is always associated with palette color 1 and the normal color is associated with palette color 0.

The advantage of a palette window is that it is very easy to change the displayed colors; simply change the palette. Your program does not have to remember which color combinations are used where.

Positioning Procedures

The positioning procedures are used to handle window coordinates:

ObscuredAt	See if a window is obscured a particular point.
At	See what window is at a particular position.
GotoXY	Move cursor.
WhereX	Get current column.
WhereY	Get current row.
ConvertCoords	Convert relative coordinates to absolute.

Contents Manipulation Procedures

The IO module is used for most and output. The Window module uses the redirection features of the IO module to alter the destination of the output procedures defined in IO, so for the majority of items you simply need to import the IO module into your application along with Window. There are, however, some extra procedures in Window to extend the facilities offered by IO:

InsLine	Insert a line at current row.
DelLine	Delete the line at the current row.
ClrEol	Clear to end of line.
TextColor	Change the current text color.
TextBackground	Change the current text background.
DirectWrite	Write directly to the window.
RdBufferLn	Read a line from window buffer.
WrBufferLn	Write a line to window buffer.
SetWrap	Toggle end of line wrapping.

Clear	Clear the current window.
CursorOn	Switch the cursor on.
CursorOff	Switch the cursor off.
SnapShot	Capture screen contents to window.

The RdBufferLn and WrBufferLn procedures allow you to manipulate both the text and color information of the window directly, which can be useful for special effects. Both these procedures use the following type definition:

```
TYPE
  BufferPtr = POINTER TO
    ARRAY [0..(ScreenWidth * ScreenDepth - 1)]
      OF CARDINAL;
```

Each CARDINAL represents the character and attribute bytes of the screen position. The values for the attribute bytes can be found on the last page of the ASCII table in the TopSpeed Environment.

The SnapShot procedure allows you to capture the contents of an area of the screen in a window. This can be used to duplicate the Cut & Paste facility of the TopSpeed Environment or to implement screen painting utilities.

You should not use the TextBackground and TextColor procedures with palette windows.

Multi-thread Procedure

The Window module, by default, is not able to function correctly from within processes running under the time-sliced scheduler of the Process module. The SetProcessLocks procedure is used to define the Lock and Unlock procedures that the Window module requires in order to function in a multi-thread program.

Example of Use

There is not enough space to demonstrate the use of all the procedures in the Window module, but the following example demonstrates its major features. The example program with the Process module shows how to use the IO and Process modules with the Window module. The smaller example below, opens three normal windows and randomly changes their size.

```
MODULE wDemo1;
  IMPORT Window, IO, Lib;
```

CONST

```

wd1 = Window.WinDef(
    5, 5, 10, 10,
    Window.Yellow,
    Window.Blue, FALSE,
    FALSE, FALSE,
    TRUE, Window.DoubleFrame,
    Window.White,
    Window.Red);

wd2 = Window.WinDef(
    12, 6, 35, 15,
    Window.Black,
    Window.Cyan,
    FALSE, FALSE,
    FALSE, TRUE,
    Window.SingleFrame,
    Window.LightBlue,
    Window.Brown);

wd3 = Window.WinDef(
    25, 9, 50, 19,
    Window.LightGray,
    Window.Green,
    FALSE, FALSE,
    FALSE, TRUE,
    Window.DoubleFrame,
    Window.Green,
    Window.Magenta);

```

VAR

```
w1, w2, w3: Window.WinType;
```

PROCEDURE NewSize(

```

    VAR x1, y1,
        x2, y2 : Window.RelCoord);

```

VAR

```
Width, Height, Corr: CARDINAL;
```

BEGIN

```
Width := x2 - x1;
```

```
Height := y2 - y1;
```

```
IF (Lib.RANDOM(10) < 5) THEN
```

```
    x1 := Lib.RANDOM(Window.ScreenWidth - 1);
```

```
    y1 := Lib.RANDOM(Window.ScreenDepth - 1);
```

```
END;
```

```
IF (Lib.RANDOM(10) < 5) THEN
```

```
    Width := Lib.RANDOM(Window.ScreenWidth - 2);
```

```
    Height := Lib.RANDOM(Window.ScreenDepth - 2);
```

```
END;
```

```
x2 := x1 + Width;
```

```
y2 := y1 + Height;
```

```
IF (x2 > Window.ScreenWidth) THEN
```

```
    Corr := x2 - Window.ScreenWidth;
```

```
    DEC(x1,Corr);      DEC(x2,Corr);
```

```
END;
```

```
    IF (y2 > Window.ScreenDepth) THEN
        Corr := y2 - Window.ScreenDepth;
        DEC(y1,Corr);      DEC(y2,Corr);
    END;
END NewSize;

PROCEDURE MoveWindow(w: Window.WinType);
VAR
    wd : Window.WinDef;
BEGIN
    Window.Info(w,wd);
    NewSize(wd.X1,wd.Y1,wd.X2,wd.Y2);
    Window.Change(w,wd.X1,wd.Y1,wd.X2,wd.Y2);
    END MoveWindow;

BEGIN
    Lib.RANDOMIZE;
    w1 := Window.Open(wd1);
    w2 := Window.Open(wd2);
    w3 := Window.Open(wd3);
    WHILE NOT IO.KeyPressed() DO
        MoveWindow(w1);
        MoveWindow(w2);
        MoveWindow(w3);
        CASE Lib.RANDOM(5) OF
            | 0 : Window.PutOnTop(w1);
            | 1 : Window.PutOnTop(w2);
            | 2 : Window.PutOnTop(w3);
        END;
    END;
    Window.Close(w1);
    Window.Close(w2);
    Window.Close(w3);
END wDemo1.
```

Window Reference

The following are the individual procedures defined within the Window module.

At — Get window at position

PROCEDURE At(X, Y : AbsCoord) : WinType;

This procedure returns the window handle of the window that is displayed at the position (X,Y). If there is no window at that position, it returns NIL.

Change — Alter window's size and position

PROCEDURE Change(
 W: WinType;
 X1, Y1, X2, Y2: AbsCoord);

The Change procedure is used to alter the dimension and/or position of a window. The window need not be on the screen. The outer bounds of the window W are change to (X1,Y1) to (X2,Y2). If the new window is smaller than the old one, then data is lost from the window. If it is larger then the old size, blanks are added to the display.

Clear — Clear window

PROCEDURE Clear;

This procedure clears the current window in the current background color. All data disappears from the window. The title and frame are not affected by this procedure.

ClrEol — Clear to end of line

PROCEDURE ClrEol;

The ClrEol procedure clears the current line of the current window from the current X coordinate to the right edge of the window. The color used is the current background color.

Close — Close window

```
PROCEDURE Close( VAR W : WinType );
```

This procedure closes the window W, which need not be visible or on the top. If it is visible, the window is removed from the screen. All buffers associated with W are freed and returned to the memory pool. Finally, W is set to NIL to ensure that further access to this window is impossible.

ConvertCoords — Convert window coordinates to screen coordinates

```
PROCEDURE ConvertCoords(  
    W: WinType;  
    X, Y: RelCoord;  
    VAR XO, YO: AbsCoord);
```

This procedure is used to convert the window relative coordinates (X,Y) into the absolute screen coordinates (XO,YO). This is useful if you want to create a new window in a particular position with regard to W. The resulting absolute coordinates are calculated from (X,Y) assuming that they are a relative position within window W.

CursorOn — Turn cursor on/off

CursorOff

```
PROCEDURE CursorOff;  
PROCEDURE CursorOn;
```

These two procedures turn the cursor on and off in the current window. Each window can have a different cursor state and as your program moves between them the cursor is automatically set to the required state.

DelLine — Delete line

```
PROCEDURE DelLine;
```

This procedure deletes the line at the current row of the current window. If necessary, the lines below it are scrolled up and the bottom line of the window is cleared to the current background color.

DirectWrite — Write directly to screen

```
PROCEDURE DirectWrite(  
    X, Y: RelCoord;  
    A: ADDRESS;  
    Len: CARDINAL);
```

This procedure is used to write text directly to the screen. (X,Y) is the start position for the text. A is a pointer to an array of characters that is the string you want printing. Len is the number of characters in the string.

You can use this procedure to achieve different effects, such as multiple titles to a window, since it will allow you to write to the relative coordinates (0,0) (the top line of the window). No check is made for end-of-line wrap or special characters in the string, all characters are displayed.

GotoXY — Set new position in window

```
PROCEDURE GotoXY( X, Y : RelCoord );
```

This procedure moves the current cursor position in the current window to (X,Y). Any further screen output or action will then take place at this new cursor position. If the position is outside the area of the window, the position will be adjusted to the nearest boundary.

Hide — Hide a window

```
PROCEDURE Hide( W : WinType );
```

This procedure removes a window from the screen, if it is visible. It does not prevent output to the window but the effects of any changes to the window will only become visible when the window is redisplayed using the PutOnTop procedure.

As a result of hiding a window, any windows, or parts of windows, obscured by it are uncovered.

Info — Get information on window

```
PROCEDURE Info( W : WinType; VAR WD : WinDef );
```

The Info procedure copies the current definition of the window W into the WinDef record WD. This allows you to base the definition of new windows on the current configuration of another window.

InsLine — Insert blank line

PROCEDURE InsLine;

This procedure inserts a line at the current row of the current window. The current line is scrolled down and lines may be lost from the bottom line of the window. The new blank line is cleared in the current background color.

ObscuredAt — Is window obscured at this position?

PROCEDURE ObscuredAt(
 W: WinType;
 X, Y: RelCoord): BOOLEAN;

This procedure return TRUE if the window W is on the screen and obscured from view at the position (X,Y) by one or more window. Otherwise it returns FALSE.

This procedure is useful for determining of your next screen output will be visible to the user of your program.

Open — Open a new window

PaletteOpen

PROCEDURE Open(
 WD: WinDef): WinType;
PROCEDURE PaletteOpen(
 WD: WinDef;
 Pal: PaletteDef): WinType;

Both these procedures define a new window and return a window handle. The Open procedure opens a normal window while the PaletteOpen procedure opens a palette window.

The new window becomes the current window, whether it is visible or not, and all positioning and output procedures will automatically use this window.

The window will only be displayed on the screen if the Hidden field of WD is FALSE.

The Open procedure uses the colors defined in WD when drawing the window. The frame, if used, will be of colored using FrameFore and

FrameBack. The main part of the window will be cleared using the defined Background color.

The PaletteOpen procedure uses entry Pal[NormalPaletteColor].Back to clear the main part of the window, and colors the frame, if present, using Pal[FramePaletteColor].

A new window has no title, and the SetTitle procedure must be used to give it one, if required. The cursor is initially placed at relative coordinate (1,1), i.e., the top left-hand corner of the window.

A window opened with Hidden set to FALSE, can be displayed on the screen using the PutOnTop procedure.

PaletteColor — Get current palette color

PROCEDURE PaletteColor(): PaletteRange;

This procedure returns the current color from the current palette of the current window. Note this is simply the index of the color record in the palette array, it does not refer to the actual colors used.

PaletteColorUsed — Is palette color used in window?

PROCEDURE PaletteColorUsed(
 W: WinType;
 pc: PaletteRange): BOOLEAN;

This procedure returns TRUE if the palette color number, pc, is currently being used anywhere in the window W. The window does not have to be visible.

PutBeneath — Put one window beneath another

PROCEDURE PutBeneath(W : WinType; WA: WinType);

PutBeneath places the window W beneath the window WA. If both are visible on the screen then W may be all or partially obscured by WA.

PutOnTop — Display window on top of others

```
PROCEDURE PutOnTop( W : WinType );
```

This procedure makes W the top window. If W is hidden, it is displayed on the screen. This will include any output made to the window since it was obscured. It also makes W the current window and all subsequent output will appear in this window until the current window is changed.

RdBufferLn — Read line directly from window buffer

```
PROCEDURE RdBufferLn(
    W: WinType;
    X, Y: RelCoord;
    Dest: ADDRESS;
    Len: CARDINAL);
```

This procedure reads one line from the window buffer of the window W. This line includes not only the character information but also the attribute (color) information about each character position.

(X,Y) is the start position in the window. Dest is a pointer to an array of Len CARDINALs or other WORD-sized objects. The procedure reads the character and attribute information from Len character positions into the array. The WrBufferLn procedure enables you to change information in the buffer. For example:

```
VAR
    Buff : ARRAY [0..5] OF
        RECORD
            Chr : SHORTCARD;
            Atr : SHORTCARD;
        END;
    i : CARDINAL;
...
Window.RdBufferLn(aWindow,1,1,ADR(Buff),6);
FOR i := 0 TO 5 DO
    Buff[i].Atr := Window.Red;
END;
Window.WrBufferLn(aWindow,1,1,ADR(Buff),6);
...
(*
    changes the color of the first six
    characters of the window 'aWindow'
    to Red on Black
*)
```

SetFrame — Set new frame for window

```
PROCEDURE SetFrame(  
    W: WinType;  
    Frame: FrameStr;  
    Fore, Back: Color);
```

This procedure changes the frame of a window. W is the window handle of the window to alter. This window does not have to be visible or on top. The Frame parameter defines the new frame string. The color parameters (ignored for palette windows) specify the colors to be used for the frame. If necessary, the frame and title, if present, will be redrawn. The title of a window always appears in the same color as the frame.

SetPalette — Set palette for palette window

```
PROCEDURE SetPalette(  
    W: WinType;  
    Pal: PaletteDef);
```

This procedure is used to change the entire palette of a palette window. If the window, W, is visible, the display is updated to reflect the new palette immediately. Otherwise the new palette will be used when the window becomes visible.

SetPaletteColor — Set new palette color

```
PROCEDURE SetPaletteColor(pc: PaletteRange);
```

This procedure selects the palette color number to be used for future window output. This procedure must be used for palette windows instead of TextColor and TextBackground. The colors are set to those specified in element pc of the current palette array for the current window.

SetProcessLocks — Multi-process support

```
PROCEDURE SetProcessLocks(  
    LockProc,  
    UnlockProc: LockProc);
```

This procedure is used to assign the correct procedures to implement process locks when using the Window module with the Process module. The type LockProc is defined as:

```
TYPE  
    LockProc = PROCEDURE;
```

The usual procedures to assign are the Lock and Unlock procedures in the Process module. For example:

```
IMPORT Process, Window;  
...  
Window.SetProcessLocks(  
    Process.Lock,  
    Process.Unlock);
```

SetTitle — Set title for window

```
PROCEDURE SetTitle(  
    W: WinType;  
    Title: ARRAY OF CHAR;  
    Mode: TitleMode);
```

The SetTitle procedure adds a new title or changes an old title to a window. Title is the string to use as the title. The parameter W is the window whose title you want to change and the Mode parameter is one of the following:

```
TYPE  
    TitleMode = (  
        NoTitle, (* Remove title *)  
        LeftUpperTitle,  
        CenterUpperTitle,  
        RightUpperTitle,  
        LeftLowerTitle,  
        CenterLowerTitle,  
        RightLowerTitle  
    );
```

SetWrap — Toggle end of line wrap

PROCEDURE SetWrap(on: BOOLEAN);

Sets the end of line wrapping mode on or off for the current window. If on is TRUE then end of line wrap is set ON, otherwise it is set OFF.

SnapShot — Capture data from screen

PROCEDURE SnapShot;

This procedure updates the window buffer of the current window from the screen. The current window cannot be a palette window. The current window must also be hidden (by using Hide or opening the window with Hidden set to TRUE). A call to SnapShot will then “capture” the current screen contents in the window buffer of the current window.

TextBackground — Set text background color

PROCEDURE TextBackground(c : Color);

This procedure sets the current background color of the current window to c. This procedure should not be used with palette windows. All subsequent output to the current window will have c as the background color. It does not affect existing text.

TextColor — Set text color

PROCEDURE TextColor(c : Color);

The procedure sets the current text color of the current window to c. This procedure should not be used with palette windows. All subsequent output to the current window will have c as the text color. It does not affect existing text.

Top — Get top windows

PROCEDURE Top() : WinType;

This procedure returns the window handle of the top window.

Use — Switch to another window

PROCEDURE Use(W : WinType);

This procedure makes W the current output window for the current process. In a multi-thread program, each process has its own current window. See the Process module for an example of its use.

Use does not make the window visible if it is hidden, you must use PutOnTop to do that.

Used — Get current window

PROCEDURE Used() : WinType;

This procedure returns the window handle of the current window for the current process. If no window has been assigned with Use, then the window handle of the top window is returned.

WhereX — Determine current position in window

WhereY

PROCEDURE WhereX() : RelCoord;
PROCEDURE WhereY() : RelCoord;

These two procedures are used to return the current position of the cursor in the current window. WhereX returns the current relative x-coordinate (column) of the cursor. WhereY returns the current y-coordinate (row) of the cursor.

You can convert these values to absolute screen coordinates using the ConvertCoords procedure.

WrBufferLn — Write line directly to window buffer

```

PROCEDURE WrBufferLn(
    W: WinType;
    X, Y: RelCoord;
    Src: ADDRESS;
    Len: CARDINAL);

```

This procedure updates a single line of the window buffer of the window W with the data contained in Src. This line includes not only the character information but also the attribute (color) information about each character position.

(X,Y) is the start position in the window. Src is a pointer to an array of Len CARDINALs or other WORD-sized objects. The procedure writes the character and attribute information from Len character positions into the window buffer. The RdBufferLn procedure enables you to read existing information in the buffer. For example:

```

VAR
    Buff : ARRAY [0..5] OF
        RECORD
            Chr : SHORTCARD;
            Atr : SHORTCARD;
        END;
    i : CARDINAL;
...
Window.RdBufferLn(aWindow,1,1,ADR(Buff),6);
FOR i := 0 TO 5 DO
    Buff[i].Atr := Window.Red;
END;
Window.WrBufferLn(aWindow,1,1,ADR(Buff),6);
...
(*
    changes the color of the first six
    characters of the window 'aWindow'
    to Red on Black
*)

```

APPENDIX A

RUN-TIME ERROR CODES;

This appendix describes the meanings of the error codes you may encounter when using the TopSpeed Modula-2 Compiler.

File Format

The file ERRORINF.\$\$\$ has the following format:

SSSS:0000 CCCC

where S = code segment, O = IP value and C = Hex error code.

General Errors

10H	Memory allocation initialization (Out Of Memory).
11H	I/O initialization (Out Of Memory).
12H	Invalid use of near heap in DLL.
21H	Dos Re-entered.
22H	NullProc Called.
D1H	Graphics Call, Not Supported Under OS/2.
D2H	OSFatalError, Operating System Error.
FFH	User Break.

Window Module

30H	Invalid window.
31H	Use memory allocation.
32H	SetTitle memory allocation.
33H	Initialization error (Out Of Memory).

DOS Process Module

40H	Too many processes.
41H	New process memory allocation.
42H	Signal memory allocation.
43H	Scheduler memory allocation.
44H	Initialization error.
45H	Stop process error.
48H	StopProcess - Corrupt Process.
49H	TRANSFER - Corrupt Process.
4AH	ExecCmd, Memory Failure.
4BH	ExecCmd, Command Line Too Long.
4CH	ExecCmd, Command Processor Failed To Load.

OS/2 Process Module

50H	StartScheduler error.
51H	StopScheduler error.
52H	StartProcess error.
53H	SEND error.
54H	WAIT error.
55H	Notify error.
56H	Init error.
57H	Delay error.

Memory Management

89H	NearMakeHeap, Invalid Argument.
8AH	NearHeapAllocate, Invalid Argument.
8BH	NearHeapDeallocate, Invalid Argument.
8CH	NearHeapAvail, Invalid Argument.
8DH	NearHeapTotalAvail, Invalid Argument.
8EH	NearHeapChangeSize, Invalid Argument.
8FH	NearHeapChangeAlloc, Invalid Argument.
90H	FarHeapAllocate, Out Of Memory.
91H	FarHeapDeallocate, Invalid Argument.
92H	FarHeapDeallocate, Heap Corrupt.
93H	FarHeapChangeAlloc, Invalid Argument.
94H	FarAllocate, Out Of Memory.
95H	NearMakeHeap, Size Too Small.
96H	NearHeapAllocate, Out Of Memory.
97H	NearHeapDeallocate, Invalid Argument.
98H	NearAllocate, Out Of Memory.
99H	Far Heap Call, Not Supported Under OS/2.
9AH	ShtHeap.Initialize, Heap too small.
9BH	ShtHeap.Increase, Not Enough Space.
9CH	ShtHeap.Allocate, Size Error.
9DH	ShtHeap.Allocate, Out Of Memory.
9EH	ShtHeap.Free, Invalid Pointer.
9FH	ShtHeap.Test, Heap Corruption.

File I/O

A0H	Close, Invalid Handle.
A2H	Open, File Access Error.
A3H	OpenRead, File Access Error.
A4H	Append, File Access Error.
A5H	Create, File Access Error.

A6H	WrBin
A7H	RdBin
A8H	FIO Internal Error, Write Error Flushing File.
A9H	GetPos, Write Seek Error.
AAH	Seek, Write Seek Error.
ACH	Truncate, File Write Error.
AEH	Erase, File Name Error.
AFH	Rename, File Name Error.
B0H	ChDir, Directory Name Error.
B1H	MkDir, Directory Name Error.
B2H	RmDir, Directory Name Error.
B3H	GetDir, Drive Number Error.
B4H	ReadFirstEntry
B5H	ReadNextEntry
BBH	Assign Buffer, Stream Table Full.
BCH	FIO Directory Call, Not Supported Under OS/2.
BEH	IO.RedirectInput.
BFH	IO.RedirectOutput.

Process Module

C0H	StartScheduler, System Error.
C1H	StartProcess, System Error.
C2H	StartProcess, SetPriority - System Error.
C3H	StartProcess, SuspendThread - System Error.
C4H	SEND, Semaphore System Error.
C5H	WAIT, Semaphore System Error.
C6H	Notify, Semaphore System Error.
C7H	Init, Semaphore System Error.
C8H	Delay, Sleep System Error.
C9H	Launch, System Error.
CAH	NEWPROCESS, Too Many Threads.
FFH	User Break.
CBH	IOTRANSFER, Invalid Argument.

Index

Symbols

(Not equal) 26
 & (Logical AND) 26
 << (Left shift) 42
 <> (Not equal) 26
 >> (Right shift) 42
 ~ (Logical NOT) 26

A

ABS 31, 51
 accessing the printer 115
 ACos
 MATHLIB 218
 Actual parameters 49
 AddAddr
 Lib 191
 AddExtension
 FIOR 140
 Addition operator 42
 ADDRESS 31
 ADR 31, 41, 51
 Alias declaration 31
 aliasing in class declarations 72
 ALLOCATE 52
 Storage 241
 Allocate
 ShtHeap 237
 AllocatePages
 LIM 211
 AND 25, 26, 41, 42
 Append
 FIO 116
 Str 255
 AppendHandle
 FIO 117
 AppendStream
 FIO 117
 Arc
 Graph 156
 Arithmetic operators
 in Modula 2 42
 ARRAY 25, 28, 34, 47, 49
 ARRAY OF BYTE 50
 ARRAY OF LONGWORD 50
 ARRAY OF WORD 50
 Array types 34
 ASin

MATHLIB 218
 AssignBuffer
 FIO 117
 Assignment compatibility 37
 Assignment statement 43
 At
 Window 282
 ATan
 MATHLIB 218
 ATan2
 MATHLIB 218
 Available
 Storage 242
 Awaited
 Process 233

B

base class 61
 base class initialization 68
 BcdToLong
 MATHLIB 216
 BEGIN 25, 47
 Binary procedures 48, 55
 BiosIO
 KBFlags 105
 KeyPressed 104
 RdChar 104
 RdKey 104
 BITSET 31
 BOOLEAN 31, 33, 41, 42
 BY 25, 45
 BYTE 31, 37, 50

C

Calling procedures 47, 49
 CAP 31, 51
 Caps
 Str 255
 CARDINAL 31, 32, 33, 37, 40
 CardToStr
 Str 256
 CASE 25, 35, 44
 CGASnow
 Window 275
 Change
 Window 282
 ChangeExtension
 FIOR 140
 CHAR 31, 33
 Character literal 27

- CharPos
 - Str 256
- ChDir
 - FIO 118
- checked guard operator 74
- CHR 31, 51
- Circle
 - Graph 157
- CLASS 25
- class compatibility rules 73
- class declarations 63
- class implementation 63
- class initialization block 66
- class interface 63
- class name clash 71
- class name visibility 71
- class scope 62, 64
- class syntax 63, 67, 74, 76
- class variables 66
- Clear
 - Window 282
- ClearScreen
 - Graph 158
- Close
 - FIO 118
 - Window 283
- ClrEol
 - Window 282
- Comments 27
- Compare
 - Lib 192
 - Str 257
- Comparison operators 41
- Comparisons 41
- Concat
 - Str 257
- CONST 25, 31, 39
- constants
 - MsMouse 221
 - ShtHeap 236
 - Window 275
- Constants in Modula 2 38
- Conventions
 - Key combinations 21
 - Typographic 21
- ConvertCoords
 - Window 283
- Copy
 - Str 258
- coroutines
 - SYSTEM 268
- Cos

- MATHLIB 218
- CosH
 - MATHLIB 219
- Cpuld
 - Lib 193
- Create
 - FIO 119
 - FIOR 141
- Cube
 - Graph 158
- Cursor
 - MsMouse 223
- CursorOff
 - Window 283
- CursorOn
 - Window 283

D

- Data hiding 30
- DEALLOCATE 53
 - Storage 242
- DeallocatePages
 - LIM 211
- DEC 31, 52
- DecAddr
 - Lib 191
- Decimal literal 26
- Declarations in Modula 2 30
- DEFINITION 25, 54
- Definition part 54
- Delay
 - Lib 193
 - Process 233
- Delete
 - Str 258
- Delimiters 25, 26
- DellLine
 - Window 283
- Dereferencing 40
- derived class 61
- Designators in Modula 2 40
- DI
 - SYSTEM 270
- DirectWrite
 - Window 284
- DisableBreakCheck
 - Lib 194
- DisableExceptionHandling
 - FloatExc 145
- Disc
 - Graph 157

DisplayCursor
 Graph 159
 DISPOSE 31, 52
 DIV 25, 41, 42
 Division operators 42
 DO 25, 44, 45, 46
 Dos
 Lib 194
 DriverSize
 MsMouse 223

E

EI
 SYSTEM 270
 Ellipse
 Graph 159
 ELSE 25, 35, 43, 44
 ELSIF 25
 EnableBreakCheck
 Lib 194
 EnableExceptionHandling
 FloatExc 145
 Encapsulation 59
 END 25, 47
 EndOfRd
 IO 182
 Eng
 FIO 109
 Enumeration types 33
 Environment
 Lib 195
 EnvironmentFind
 Lib 195
 EOF
 FIO 109
 Erase
 FIO 119
 FIOR 141
 ERRORINF.\$\$\$ 108
 EXCL 31, 52
 Exec
 Lib 196
 ExecCmd
 Lib 197
 Execute
 Lib 197
 Exists
 FIO 120
 EXIT 25, 45
 Exp
 MATHLIB 215

ExpandPath
 FIOR 141
 EXPORT 25, 57
 Expressions in Modula 2 41

F

FALSE 31, 42
 Far procedures 51
 FarADDRESS 31, 40
 FarADR 31, 51
 FarAvailable
 Storage 242
 FarDEALLOCATE
 Storage 243
 FarHeapAllocate
 Storage 244
 FarHeapAvail
 Storage 245
 FarHeapChangeAlloc
 Storage 247
 FarHeapChangeSize
 Storage 248
 FarHeapDeallocate
 Storage 250
 FarHeapTotalAvail
 Storage 251
 FarMakeHeap
 Storage 252
 FarNIL 31, 39
 FatalError
 Lib 197
 Field names 35
 Field selection 40
 FieldOfs 51
 file buffers 107
 file handles 106
 file positioning 106
 Fill
 Lib 198
 FindNewPath
 FIOR 142
 FindPath
 FIOR 142
 FindSubStr
 Str 259
 FIO
 accessing the printer 115
 Append 116
 AppendHandle 117
 AppendStream 117
 ChDir 118

Close 118
 Create 119
 definitions 107
 Eng 109
 EOF 109
 Erase 119
 Exists 120
 file buffers 107
 file handles 106
 file positioning 106
 Flush 120
 formatted I/O 108
 GetCurrentDate 120
 GetDir 121
 GetDrive 121
 GetFileDate 121
 GetFileStamp 122
 GetPos 122
 GetStreamPointer 122
 I/O error handling 108
 IOresult 123
 Mkdir 123
 Open 124
 OpenRead 125
 predefined file handles 108
 random file processing 112
 Rdbin 126
 RdBool 127
 RdCard 127
 RdChar 127
 RdHex 127
 RdInt 127
 RdItem 126
 RdLngCard 127
 RdLngHex 127
 RdLngInt 127
 RdLngReal 127
 RdReal 127
 RdShtCard 127
 RdShtHex 127
 RdShtInt 127
 RdStr 128
 ReadFirstEntry 129
 ReadNextEntry 131
 Rename 131
 Rmdir 131
 Seek 132
 Separators 109
 sequential file processing 109
 SetDrive 132
 SetFileDate 132
 ShareMode 109, 116, 119, 124, 125
 Size 132
 ThreadEOF 132
 ThreadOK 133
 Truncate 133
 WrBin 134
 WrBool 135
 WrCard 135
 WrChar 135
 WrCharRep 134
 WrFixLngReal 135
 WrFixReal 135
 WrHex 135
 WrInt 135
 WrLngCard 135
 WrLngHex 135
 WrLngInt 135
 WrLngReal 135
 WrReal 135
 WrShtCard 135
 WrShtHex 135
 WrShtInt 135
 WrStr 138
 WrStrAdj 138
 FIO.AssignBuffer 117
 FIOR 139
 AddExtension 140
 ChangeExtension 140
 Create 141
 Erase 141
 ExpandPath 141
 FindNewPath 142
 FindPath 142
 IOresult 142
 IsExtension 143
 MakePath 143
 Open 143
 ReadRedirectionFile 144
 RemoveExtension 140
 SplitPath 143
 FIXREAL 31
 FixRealToStr
 Str 259
 FLOAT 31, 51
 FloatExc 145
 DisableExceptionHandling 145
 EnableExceptionHandling 145
 InstallHandler 146
 FloodFill
 Graph 160
 FOR 45
 Formal notation for syntax descriptions 28
 Formal type 47

formatted I/O 108
 FormIO 147
 WrF 148
 WrF1 148
 WrF2 148
 WrF3 149
 WrF4 149
 WrF5 149
 FORWARD 25, 47
 FrameStr
 Window 277
 Free
 ShtHeap 237
 FreePages
 LIM 212
 FROM 25, 56
 FullScreen
 Window 277
 Function procedures 47

G

Generic tokens 25, 26
 GetBkColor
 Graph 161
 GetCurrentDate
 FIO 120
 GetDate
 Lib 198
 GetDir
 FIO 121
 GetDrive
 FIO 121
 GetFileDate
 FIO 121
 GetFileStamp
 FIO 122
 GetFillMask
 Graph 161
 GetFlags
 SYSTEM 270
 GetImage
 Graph 162
 GetLineStyle
 Graph 162
 GetMotion
 MsMouse 223
 GetPage
 MsMouse 224
 GetPageFrame
 LIM 212
 GetPos
 FIO 122
 GetPress
 MsMouse 224
 GetRelease
 MsMouse 224
 GetSensitivity
 MsMouse 225
 GetStatus
 LIM 212
 MsMouse 225
 GetStreamPointer
 FIO 122
 GetTextColor
 Graph 162
 GetTextPosition
 Graph 163
 GetTime
 Lib 198
 GetVideoConfig
 Graph 163
 Global variables 55
 GOTO 25, 46
 GotoXY
 Window 284
 Graph 150
 adapters 152
 Arc 156
 Circle 157
 ClearScreen 158
 colors 154
 Cube 158
 Disc 157
 DisplayCursor 159
 Ellipse 159
 FillMask 150
 FloodFill 160
 GetBkColor 161
 GetFillMask 161
 GetImage 162
 GetLineStyle 162
 GetTextColor 162
 GetTextPosition 163
 GetVideoConfig 163
 GraphMode 163
 HLine 164
 ImageSize 164
 InitCGA 164
 InitEGA 164
 InitGraph 164
 InitHerc 164
 InitVGA 164
 Line 165

- Line Style 150
- OutText 165
- palettes 154
- Pie 166
- Plot 166
- Point 167
- Polygon 167
- PutImage 168
- Rectangle 169
- RemapAllPalette 170
- RemapPalette 170
- screen modes 152
- SelectPalette 171
- SetActivePage 172
- SetBkColor 161
- SetClipRgn 173
- SetFillMask 174
- SetLineStyle 175
- SetTextColor 175
- SetTextPosition 176
- SetTextWindow 176
- SetVideoMode 177
- SetVisualPage 172
- StackFill 160
- TextMode 177
- TrueCircle 157
- TrueDisc 157
- VideoConfig record 151
- GraphMode
 - Graph 163

H

- HALT 31, 53
- Hardware record
 - MsMouse 220
- HashString
 - Lib 199
- HeapAllocate
 - Storage 243
- HeapAvail
 - Storage 245
- HeapChangeAlloc
 - Storage 246
- HeapChangeSize
 - Storage 247
- HeapDeallocate
 - Storage 249
- HeapTotalAvail
 - Storage 250
- Hex literal 27
- Hexadecimal 27

- Hide
 - Window 284
- HIGH 31, 51
- HLine
 - Graph 164
- HSort
 - Lib 201

I

- I/O error handling 108
- Identifiers 26, 27
- Identifiers, predefined 31
- IF 25, 43
- ImageSize
 - Graph 164
- IMPLEMENTATION 25
- IMPLEMENTATION' 54
- Implementation part 54
- IMPORT 25, 56
- Importing 56
- IN 25, 41, 42
- In
 - SYSTEM 270
- INC 31, 53
- IncAddr
 - Lib 191
- INCL 31, 53
- Increase
 - ShtHeap 237
- Indexing 40
- Info
 - Window 284
- inheritance 68
- Init
 - Process 233
- InitCGA
 - Graph 164
- InitEGA
 - Graph 164
- InitGraph
 - Graph 164
- InitHerc
 - Graph 164
- Initialization code 55
- Initialize
 - ShtHeap 237
- InitVGA
 - Graph 164
- INLINE 25, 48, 55
- Inline procedures 55
- Insert

- Str 260
- InsLine
 - Window 285
- InstallHandler
 - FloatExc 146
- instance 60
- INTEGER 31, 32, 33, 37
- Intr
 - Lib 199
- IntToStr
 - Str 260
- invoking a base class method 70
- invoking methods 67
- InW
 - SYSTEM 271
- IO 179
 - EndOfRd 182
 - KeyPressed 182
 - RdBool 185
 - RdCard 185
 - RdChar 185
 - RdHex 185
 - RdInt 185
 - RdItem 183
 - RdKey 184
 - RdLn 184
 - RdLngCard 185
 - RdLngHex 185
 - RdLngInt 185
 - RdLngReal 185
 - RdReal 185
 - RdShtCard 185
 - RdShtHex 185
 - RdShtInt 185
 - RdStr 185
 - RedirectInput 186
 - RedirectOutput 186
 - ThreadOK 186
 - WrCharRep 186
 - WrStr 189
 - WrStrAdj 189
- IOresult
 - FIO 123
 - FIOR 142
- IOTRANSFER
 - SYSTEM 271
- IS 25
- IS operator 76
- IsExtension
 - FIOR 143
- Item
 - Str 261

- ItemS
 - Str 261

K

- KBFlags
 - BiosIO 105
- KeyPressed
 - BiosIO 104
 - IO 182
- Keywords 25

L

- LABEL 25, 46
- Largest
 - ShtHeap 238
- Left shift 42
- Length
 - Str 262
- Lib
 - AddAddr 191
 - Compare 192
 - Cpuld 193
 - DecAddr 191
 - Delay 193
 - DisableBreakCheck 194
 - Dos 194
 - EnableBreakCheck 194
 - Environment 195
 - EnvironmentFind 195
 - Exec 196
 - ExecCmd 197
 - Execute 197
 - FatalError 197
 - Fill 198
 - GetDate 198
 - GetTime 198
 - HashString 199
 - HSort 201
 - IncAddr 191
 - Intr 199
 - LongJump 206
 - MathError 199
 - Move 200
 - NoSound 200
 - ParamCount 200
 - ParamStr 201
 - RAND 203
 - RANDOM 203
 - RANDOMIZE 203
 - ScanL 204

- ScanNeL 204
- ScanNeR 205
- ScanR 205
- SetDate 198
- SetJump 206
- SetReturnCode 207
- SetTime 198
- Sound 208
- SubAddr 191
- SysErrno 208
- Terminate 208
- UserBreak 209
- WordFill 209
- WordMove 210
- WrDosError 210
- LightPen
 - MsMouse 225
- LIM 211
 - AllocatePages 211
 - DeallocatePages 211
 - FreePages 212
 - GetPageFrame 212
 - GetStatus 212
 - MapPage 213
 - variables 211
- Line
 - Graph 165
- Local modules 57
- Local variables 48
- Lock
 - Process 233
- Log
 - MATHLIB 215
- Log10
 - MATHLIB 215
- LONGCARD 31, 32, 37
- LONGINT 31, 32, 37
- LongJump
 - Lib 206
- LONGREAL 31, 32
- LongToBcd
 - MATHLIB 216
- LONGWORD 31, 37, 50
- LOOP 25, 45

M

- Main module 54
- MakeHeap
 - Storage 251
- MakePath
 - FIOR 143

- MapPage
 - LIM 213
- Match
 - Str 262
- MathError 214
 - Lib 199
 - MATHLIB 216
- MATHLIB
 - ACos 218
 - ASin 218
 - ATan 218
 - Atan2 218
 - BcdToLong 216
 - Cos 218
 - CosH 219
 - Exp 215
 - Log 215
 - Log10 215
 - LongToBcd 216
 - MathError 214, 216
 - Mod 216
 - Pow 217
 - Rexp 217
 - Sin 218
 - SinH 219
 - Sqrt 219
 - Tan 218
 - TanH 219
- MAX 31, 51
- messages 60
- methods 60
- MIN 31, 52
- MkDir
 - FIO 123
- MOD 25, 41, 42
- Mod
 - MATHLIB 216
- Modula 2
 - comments 27
 - declarations 30
 - delimiters 25, 26
 - generic tokens 25, 26
 - keywords 25
 - language syntax 22
 - separators 25, 27
 - tokens 25
 - types 30
 - white-space 27
- MODULE 25, 54
- Modules 54
- Move
 - Lib 200

MsData record
 MsMouse 220
 MsGraphcur record
 MsMouse 220
 MsMotion record
 MsMouse 220
 MsMouse 220
 constants 221
 Cursor 223
 DriverSize 223
 GetMotion 223
 GetPage 224
 GetPress 224
 GetRelease 224
 GetSensitivity 225
 GetStatus 225
 Hardware record 220
 LightPen 225
 MsData record 220
 MsGraphcur record 220
 MsMotion record 220
 MsRange record 220
 MsSense record 221
 MsTextCur record 221
 Reset 225
 RestoreDriver 226
 SaveDriver 226
 SetDouble 226
 SetGraphCursor 226
 SetInterrupt 226
 SetMickeys 227
 SetPage 224
 SetPosition 227
 SetRange 227
 SetSensitivity 225
 SetTextCursor 227
 Software record 221
 UpdateScreen 227
 MsRange record
 MsMouse 220
 MsSense record
 MsMouse 221
 MsTextcur record
 MsMouse 221
 MTA 31
 multiple inheritance 70
 multiple threads
 Window 279
 Multiplication operators 42

N

Named constants 39
 near and far variants
 Storage 239
 NearADDRESS 31
 NearADR 31, 52
 NearALLOCATE
 Storage 241
 NearAvailable
 Storage 242
 NearDEALLOCATE
 Storage 243
 NearHeapAllocate
 Storage 244
 NearHeapAvail
 Storage 245
 NearHeapChangeAlloc
 Storage 246
 NearHeapChangeSize
 Storage 248
 NearHeapDeallocate
 Storage 249
 NearHeapTotalAvail
 Storage 250
 NearMakeHeap
 Storage 251
 NearNIL 31, 39
 NEW 31, 53
 NEWPROCESS
 SYSTEM 272
 NextPos
 Str 263
 NIL 31, 39
 NoSound
 Lib 200
 NOT 25, 26, 41, 42
 Notify
 Process 234
 NULLPROC 31

O

object 60
 ObscuredAt
 Window 285
 Octal literal 26
 ODD 31, 52
 OF 25
 Offsets 24
 OfS 31, 52
 Opaque types 54

- Open
 - FIO 124
 - FIOR 143
 - Window 285
- Open array parameters 47, 48, 49
- OpenRead
 - FIO 125
- Operands 41
- Operators
 - in Modula 2 41
- OR 25, 41, 42
- ORD 31, 52
- Ordinal types 33
- Out
 - SYSTEM 272
- OutText
 - Graph 165
- OutW
 - SYSTEM 272
- overriding methods 69

P

- PaletteColor
 - Window 286
- PaletteColorDef
 - Window 277
- PaletteColorUsed
 - Window 286
- PaletteOpen
 - Window 285
- ParamCount
 - Lib 200
- Parameters 47
- ParamStr
 - Lib 201
- Pie
 - Graph 166
- Plot
 - Graph 166
- Point
 - Graph 167
- POINTER 36
- Pointer constructor 40
- Pointer types 36
- Polygon
 - Graph 167
- polymorphism 61
- Pos
 - Str 263
- Pow
 - MATHLIB 217

- Precedence of operators 41
- predefined file handles 108
- Predefined identifiers 31
- Predefined procedures 51
- Prepend
 - Str 263
- Priority 54
- PROC 31, 51
- PROCEDURE 25, 47, 50
- Procedure body 47
- Procedure types 47
- Procedures 47
- Procedures types 50
- Process
 - Awaited 233
 - Delay 233
 - Init 233
 - Lock 233
 - Notify 234
 - SEND 234
 - StartProcess 234
 - StartScheduler 234
 - StopProcess 235
 - StopScheduler 235
 - Unlock 235
 - WAIT 235
- Productions, syntax 28
- Proper procedures 47
- PutBeneath
 - Window 286
- PutImage
 - Graph 168
- PutOnTop
 - Window 287

Q

- QUALIFIED 25, 57
- Qualified names 56

R

- RAND
 - lib 203
- RANDOM
 - Lib 203
- random file processing 112
- RANDOMIZE
 - Lib 203
- RCharPos
 - Str 264
- RdBin

- FIO 126
- RdBool
 - FIO 127
 - IO 185
- RdBufferLn
 - Window 287
- RdCard
 - FIO 127
 - IO 185
- RdChar
 - BiosIO 104
 - FIO 127
 - IO 185
- RdHex
 - FIO 127
 - IO 185
- RdInt
 - FIO 127
 - IO 185
- RdItem
 - FIO 126
 - IO 183
- RdKey
 - BiosIO 104
 - IO 184
- RdLn
 - IO 184
- RdLngCard
 - FIO 127
 - IO 185
- RdLngHex
 - FIO 127
 - IO 185
- RdLngInt
 - FIO 127
 - IO 185
- RdLngReal
 - FIO 127
 - IO 185
- RdReal
 - FIO 127
 - IO 185
- RdShtCard
 - FIO 127
 - IO 185
- RdShtHex
 - FIO 127
 - IO 185
- RdShtInt
 - FIO 127
 - IO 185
- RdStr
 - FIO 128
 - IO 185
- ReadFirstEntry
 - FIO 129
- ReadNextEntry
 - FIO 131
- ReadRedirectionFile
 - FIOR 144
- REAL 31, 32
- Real literal 27
- RealToStr
 - Str 264
- RECORD 25, 35
- Record types 35
- Rectangle
 - Graph 169
- RedirectInput
 - IO 186
- redirection 139
- RedirectOutput
 - IO 186
- Registers record
 - SYSTEM 268
- RemapAllPalette
 - Graph 170
- RemapPalette
 - Graph 170
- RemoveExtension
 - FIOR 140
- Rename
 - FIO 131
- REPEAT 25, 44
- Reset
 - MsMouse 225
- RestoreDriver
 - MsMouse 226
- RETURN 25, 48
- Returning from procedures 47
- Rexp
 - MATHLIB 217
- Right shift 42
- Rmdir
 - FIO 131

S

- SaveDriver
 - MsMouse 226
- ScanL
 - Lib 204
- ScanNeL
 - Lib 204

- ScanNeR
 - Lib 205
- ScanR
 - Lib 205
- Scope 30
- ScreenDepth
 - Window 275
- ScreenWidth
 - Window 275
- Seek
 - FIO 132
- Seg 31, 52
- SegAllocate
 - Storage 252
- SegDeallocate
 - Storage 252
- Segment/Offset pairs 24
- Segmented architecture 24
- Segments 24
- Selectors 24
- SelectPalette
 - Graph 171
- self 65
- self scope 66
- SEND
 - Process 234
- Separators 25, 27
 - FIO 109
- sequential file processing 109
- Server modules 54, 56
- SET 25, 34
- Set difference 42
- Set intersection 42
- Set symmetric difference 42
- Set types 34
- Set union 42
- Set values 39
- SetActivePage
 - Graph 172
- SetBkColor
 - Graph 161
- SetClipRgn
 - Graph 173
- SetDate
 - Lib 198
- SetDouble
 - MsMouse 226
- SetDrive
 - FIO 132
- SetFileDate
 - FIO 132
- SetFillMask
 - Graph 174
- SetFlags
 - SYSTEM 272
- SetFrame
 - Window 288
- SetGraphCursor
 - MsMouse 226
- SetInterrupt
 - MsMouse 226
- SetJump
 - Lib 206
- SetLineStyle
 - Graph 175
- SetMickeys
 - MsMouse 227
- SetPage
 - MsMouse 224
- SetPalette
 - Window 288
- SetPaletteColor
 - Window 288
- SetPosition
 - MsMouse 227
- SetProcessLocks
 - Window 289
- SetRange
 - MsMouse 227
- SetReturnCode
 - Lib 207
- SetSensitivity
 - MsMouse 225
- SetTextColor
 - Graph 175
- SetTextCursor
 - MsMouse 227
- SetTextPosition
 - Graph 176
- SetTextWindow
 - Graph 176
- SetTime
 - Lib 198
- SetTitle
 - Window 289
- SetVideoMode
 - Graph 177
- SetVisualPage
 - Graph 172
- SetWrap
 - Window 290
- ShareMode
 - FIO 109, 116, 119, 124, 125
- SHORTADDR 31

- SHORTCARD 31, 32, 37
- SHORTINT 31, 32, 37
- ShtHeap 236
 - Allocate 237
 - constants 236
 - Free 237
 - Increase 237
 - Initialize 237
 - Largest 238
 - Test 238
 - Total 238
- Sign operators 41
- SinH
 - MATHLIB 219
- SIZE 31, 52
- Size
 - FIO 132
- Slice
 - Str 265
- SnapShot
 - Window 290
- Software record
 - MsMouse 221
- Sound
 - Lib 208
- SplitPath
 - FIOR 143
- Sqrt
 - MATHLIB 219
- StackFill
 - Graph 160
- StartProcess
 - Process 234
- StartScheduler
 - Process 234
- Statements 43
 - assignment 43
 - CASE statement 44
 - FOR statement 45
 - GOTO statement 46
 - IF statement 43
 - LOOP statement 45
 - REPEAT statement 44
 - WHILE statement 44
 - WITH statement 46
- static methods 79
- StopProcess
 - Process 235
- StopScheduler
 - Process 235
- Storage 239
 - ALLOCATE 241
 - Available 242
 - DEALLOCATE 242
 - FarAvailable 242
 - FarDEALLOCATE 243
 - FarHeapAllocate 244
 - FarHeapAvail 245
 - FarHeapChangeAlloc 247
 - FarHeapChangeSize 248
 - FarHeapDeallocate 250
 - FarHeapTotalAvail 251
 - FarMakeHeap 252
 - HeapAllocate 243
 - HeapAvail 245
 - HeapChangeAlloc 246
 - HeapChangeSize 247
 - HeapDeallocate 249
 - HeapTotalAvail 250
 - MakeHeap 251
 - near and far variants 239
 - NearALLOCATE 241
 - NearAvailable 242
 - NearDEALLOCATE 243
 - NearHeapAllocate 244
 - NearHeapAvail 245
 - NearHeapChangeAlloc 246
 - NearHeapChangeSize 248
 - NearHeapDeallocate 249
 - NearHeapTotalAvail 250
 - NearMakeHeap 251
 - SegAllocate 252
 - SegDeallocate 252
 - variables 239
- Str 253
 - Append 255
 - Caps 255
 - CardToStr 256
 - CharPos 256
 - Compare 257
 - Concat 257
 - conversion procedures 254
 - Copy 258
 - Delete 258
 - FindSubStr 259
 - FixRealToStr 259
 - Insert 260
 - IntToStr 260
 - Item 261
 - ItemS 261
 - Length 262
 - Match 262
 - NextPos 263
 - Pos 263

- Prepend 263
- RCharPos 264
- RealToStr 264
- Slice 265
- StrToC 265
- StrToCard 266
- StrToInt 266
- StrToPas 267
- StrToReal 267
- Subst 267
- String concatenation 42
- String literal 27
- Strings 49
- strings 253
- StrToC
 - Str 265
- StrToCard
 - Str 266
- StrToInt
 - Str 266
- StrToPas
 - Str 267
- StrToReal
 - Str 267
- SubAddr
 - Lib 191
- Subrange types 33
- Subst
 - Str 267
- Subtraction operator 42
- Syntactical constructs 28
- Syntax 28
- SysErrno
 - Lib 208
- SYSTEM 54, 268
 - coroutines 268
 - DI 270
 - EI 270
 - GetFlags 270
 - In 270
 - InW 271
 - IOTRANSFER 271
 - NEWPROCESS 272
 - Out 272
 - OutW 272
 - Registers record 268
 - SetFlags 272
 - TRANSFER 273

T

- Tag identifier 35

- Tag type 35
- Tan
 - MATHLIB 218
- TanH
 - MATHLIB 219
- TEMPREAL 31
- Terminate
 - Lib 208
- Test
 - ShtHeap 238
- TextBackGround
 - Window 290
- TextColor
 - Window 290
- TextMode
 - Graph 177
- THEN 25, 43
- ThreadEOF
 - FIO 132
- ThreadOK
 - FIO 133
 - IO 186
- TO 25, 45
- Tokens
 - in Modula 2 25
- Top
 - Window 290
- Total
 - ShtHeap 238
- TRANSFER
 - SYSTEM 273
- TRUE 31, 42
- TrueCircle
 - Graph 157
- TrueDisc
 - Graph 157
- TRUNC 31, 52
- Truncate
 - FIO 133
- TS.RED 139
- TYPE 25, 32, 54
- Type compatibility 37
- Type transfer functions 52
- types
 - Window 275
- Types in Modula 2 30
- Typographic Conventions 21
- Typographical conventions
 - in syntax 28

U

Unlock
 Process 235
 UNTIL 25, 44
 UpdateScreen
 MsMouse 227
 Use
 Window 291
 UserBreak
 Lib 209

V

VAL 31, 52
 Value parameters 47, 48
 VAR 25, 38, 47, 49, 50
 Variable (VAR) parameters 47, 48, 49
 variables
 LIM 211
 Storage 239
 Variant record types 35
 VIRTUAL 25
 virtual methods 78
 virtual pointers 37
 VSIZE 52

W

WAIT
 Process 235
 WhereX
 Window 291
 WhereY
 Window 291
 WHILE 25, 44
 White-space 27
 WinDef record
 Window 276
 Window 274
 At 282
 CGASnow 275
 Change 282
 Clear 282
 Close 283
 ClrEol 282
 constants 275
 ConvertCoords 283
 CursorOff 283
 CursorOn 283
 DelLine 283
 DirectWrite 284

FrameStr 277
 FullScreen 277
 GotoXY 284
 Hide 284
 Info 284
 InsLine 285
 multiple threads 279
 ObscuredAt 285
 Open 285
 PaletteColor 286
 PaletteColorDef 277
 PaletteColorUsed 286
 PaletteOpen 285
 PutBeneath 286
 PutOnTop 287
 RdBufferLn 287
 ScreenDepth 275
 ScreenWidth 275
 SetFrame 288
 SetPalette 288
 SetPaletteColor 288
 SetProcessLocks 289
 SetTitle 289
 SetWrap 290
 Snapshot 290
 TextBackGround 290
 TextColor 290
 Top 290
 types 275
 Use 291
 Used 291
 WhereX 291
 WhereY 291
 WinDef record 276
 Wintype 275
 WrBufferLn 292
 WinType
 Window 275
 Wirth, Niklaus 24
 WITH 25, 30, 46
 WORD 31, 37, 50
 WordFill
 Lib 209
 WordMove
 Lib 210
 WrBin
 FIO 134
 WrBool
 FIO 135
 WrBufferLn
 Window 292
 WrCard

FIO 135
WrChar
FIO 135
WrCharRep
FIO 134
IO 186
WrDosError
Lib 210
WrF
FormIO 148
WrF1
FormIO 148
WrF2
FormIO 148
WrF3
FormIO 149
WrF4
FormIO 149
WrF5
FormIO 149
WrFixLngReal
FIO 135
WrFixReal
FIO 135
WrHex
FIO 135
WrInt
FIO 135
WrLngCard
FIO 135
WrLngHex
FIO 135
WrLngInt
FIO 135
WrLngReal
FIO 135
WrReal
FIO 135
WrShtCard
FIO 135
WrShtHex
FIO 135
WrShtInt
FIO 135
WrStr
FIO 138
IO 189
WrStrAdj
FIO 138
IO 189