TMT Pascal

Multi-target Edition

Version 3.50 (Build 2.50)

Supplied Units

Developer’s Guide

2000

Contents

1 The Comp Unit 21

1.1 Comp Unit Types and Overloaded Operators 21

1.2 Comp Unit Procedures and Functions 22

add_cc function 22

add_cr function 22

2 The CRT Unit 23

2.1 CRT Unit Constants and Variables 23

Color constants 23

TextMode constants 24

CheckBreak variable 24

CheckEOF variable 24

DirectVideo variable 24

CheckSnow variable 25

LastMode variable 25

TextAttr variable 25

WindMax variable 25

WindMin variable 26

2.2 CRT Unit Procedures and Functions 26

AssignCrt procedure 26

ClrEOL procedure 27

ClrScr procedure 27

Delay procedure 27

DelLine procedure 27

GetCharXY function 28

GotoXY procedure 28

HideCursor procedure 28

HighVideo procedure 28

InsLine procedure 29

KeyPressed function 29

LowVideo procedure 29

NormVideo procedure 29

NoSound procedure 30

ReadKey function 30

SetScreenSize procedure....30
ShowCursor procedure....30
Sound procedure....30
TextBackGround procedure....31
TextColor procedure....31
TextMode procedure....31
WhereX function....32
WhereY function....32
Window procedure....32
WriteAttr procedure....32

3 The Debug Unit....34

4 The DOS Unit....35

4.1 Dos Unit Constants and Variables....35
Flag constants....35
File-mode constants....35
File-attribute constants....36
DataTime type....36
DosError variable....36
Registers type....37
SearchRec type....37

4.2 Dos Unit Procedures and Functions....38
DiskFree function....38
DiskSize function....38
DosExitCode function....39
DosVersion function....39
EnvCount function....39
EnvStr function....39
Exec procedure....40
FExpand function....40
FindFirst procedure....40
FindNext procedure....41
FSearch function....42
FSplit procedure....42
GetCBreak procedure....43
GetDate procedure....43
GetEnv function....44
GetFAttr procedure....44
GetFTime procedure....44

GetIntVec procedure 45
GetTime procedure 45
GetVerify procedure 46
Intr procedure 46
Keep procedure 47
MsDos procedure 47
PackTime procedure 47
SetCBreak procedure 48
SetDate procedure 48
SetFAttr procedure 48
SetFTime procedure 49
SetIntVec procedure 49
SetTime procedure 50
SetVerify procedure 51
SwapVectors procedure 51
UnpackTime procedure 51

5 The DPMI Unit 52

5.1 DPMI Unit Types 52
TDescriptor type 52
TRmRegs type 52

5.2 DPMI Unit Procedures and Functions 53
AllocateDescriptors function 53
AllocDosMemoryBlock function 53
AllocRealModeCallBack function 53
AllocateSpecificDescriptor function 54
CallRealModeFar procedure 54
CallRealModeIRet procedure 54
ClearRmRegs procedure 54
CreateCodeAlias function 54
CreateCodeDescriptor function 55
CreateDataAlias function 55
CreateDataDescriptor function 55
DosMemoryAlloc function 55
DosMemoryFree function 56
FarGetByte function 56
FarGetDWord function 56
FarGetWord function 57
FarPutByte procedure 57
FarPutDWord procedure 57

FarPutWord procedure 57
FreeDescriptor function 57
FreeDosMemoryBlock function 58
FreePhysicalMap function 58
FreeRealModeCallBack function 58
GetCS function 58
GetDisableInterruptState function 58
GetDPMIIntVec function 59
GetDPMIVer function 59
GetDS function 59
GetEnableInterruptState function 59
GetExceptionHandler function 60
GetFreeMemoryInfo function 60
GetInterruptState function 60
GetRealModeIntVec function 60
GetSegmentBaseAddress function 61
GetSelectorAccessRights function 61
MapPhysicalToLinear function 61
RealModeInt function 61
ResizeDosMemoryBlock function 62
SegmentToDescriptor function 62
SelectorInc function 62
SetDPMIIntVec function 62
SetExceptionHandler function 63
SetRealModeIntVec function 63
SetSelectorAccessRights function 63
SetSelectorBaseAddress function 63
SetSelectorLimit function 64

6 The ErrCodes Unit 65

7 The Graph Unit 66

7.1 Graph Unit Introduction 66

7.2 Graph Unit Types, Constants and Variables 68
DrawBorder variable 68
Bar3D constants 68
BitBlt operators 68
Clipping constants 68
Color constants 68
Fill pattern constants 69

FillSettingsType 69
GraphModeType........ 69
GraphWndProc........ 70
Graphic result constants........ 71
IgnoreBreak variable 71
IgnoreCloseMessage variable........ 71
Justification constants........ 71
LineSettingsType........ 72
PaletteType........ 72
PointType 72
RGBType........ 73
SVGA mode constants........ 73
TextSettingsType........ 73
Text-Style constants 73
ViewPortType........ 73
VbeInfoType........ 74
VbeModeInfoType 74

7.3 Graph Unit Procedures and Functions........ 75
AnalizeRGBColor procedure 75
Arc procedure........ 75
Bar procedure 75
Bar3D procedure 76
Circle procedure 76
ClearDevice procedure 76
ClearPage procedure........ 77
ClearViewPort procedure 77
CliRetrace procedure........ 77
CliHRetrace procedure 77
CloseGraph procedure........ 77
DetectSVGAMode procedure........ 78
DrawEllipse procedure 78
DrawHLine procedure........ 79
DrawPoly procedure........ 79
Ellipse procedure........ 79
ExpandFill procedure 79
FillCircle procedure........ 79
FillEllipse procedure........ 80
FillPoly procedure 80
FillTriangle procedure........ 80

FlipImageOX procedure 81
FlipImageOY procedure 81
FlipToMemory procedure 82
FlipToScreen procedure 82
FloodFill procedure 82
GetActivePage function 82
GetAspectRatio procedure 82
GetBytesPerScanLine function 83
GetColor function 83
GetDefaultPalette procedure 83
GetFillColor function 83
GetFillPattern procedure 83
GetFillSettings procedure 84
GetGraphBufSize function 84
GetGraphMode function 84
GetHTextel procedure 84
GetImage procedure 85
GetLfbAddress function 85
GetLineSettings procedure 85
GetLogicalPage procedure 85
GetMaxColor function 85
GetMaxPage function 86
GetMaxX function 86
GetMaxY function 86
GetOemProductName function 86
GetOemProductRev function 87
GetOemString function 87
GetOemVendorName function 87
GetPageDC function 87
GetPageSize function 88
GetPalette procedure 88
GetPixel function 88
GetRGBPalette procedure 88
GetScreenHeight function 89
GetScreenWidth function 89
GetTextSettings procedure 89
GetTranspSettings procedure 89
GetVbeCapabilities function 89
GetVbeInfo procedure 90

GetVbeModeInfo procedure 90
GetVbeModesList procedure 90
GetVbeVersion function 90
GetViewSettings procedure 91
GetVisualPage function 91
GetWindowHandle function 91
GetWriteMode function 91
GetX function 91
GetY function 92
GraphResult function 92
GraphDefaults procedure 92
GraphErrorMsg function 92
HRetrace procedure 92
ImageSize function 93
InvertImage procedure 93
IsLfbUsed function 93
Line procedure 93
LineRel procedure 94
LineTo procedure 94
MoveRel procedure 94
MoveTo procedure 95
OutCharXY procedure 95
OutText procedure 95
OutTextXY procedure 96
PutHTextel procedure 96
PutImage procedure 96
PutPixel procedure 96
PutSprite procedure 96
Rectangle procedure 97
ReleasePageDC procedure 97
RestoreCrtMode procedure 97
Retrace procedure 97
RGBColor procedure 98
SetActivePage procedure 98
SetAllPalette procedure 98
SetAspectRatio procedure 98
SetBkColor procedure 99
SetColor procedure 99
SetCustomFont procedure 99
SetFillColor procedure 99

SetFillPattern procedure....100
SetFillStyle procedure....100
SetGraphBufSize procedure....100
SetGraphMode procedure....101
SetLineStyle procedure....101
SetLogicalPage procedure....101
SetNormalMode procedure....103
SetPalette procedure....103
SetRGBPalette procedure....103
SetScreenStart procedure....103
SetSVGAMode procedure....104
SetTextJustify procedure....104
SetTextStyle procedure....104
SetSplineLineSteps procedure....104
SetTranspMode procedure....105
SetViewPort procedure....105
SetVirtualMode procedure....106
SetVisualPage procedure....106
SetWriteMode procedure....106
Spline procedure....107
Stretch procedure....107
TextHeight function....107
TextWidth function....107
TotalVbeMemory function....108
TotalVbeModes function....108
Triangle procedure....108

8 The Keyboard Unit....109

8.1 Keyboard Unit Constants....109

8.2 Keyboard Unit Procedures and Functions....110
AsciiToScan....110
GetKey....110
FlushKeyboard....110
MultikeysDone....110
MultikeysInit....111
ScanToAscii....111
TestAlt....111
TestCapsLock....111
TestCtrl....111

TestNumLock ... 112
TestScrollLock ... 112
TestShift ... 112
TestKey ... 112

9 The Math Unit ... 113

9.1 Math Unit Constants ... 113

9.2 Math Unit Procedures and Functions ... 113
ArcCos function ... 113
ArcCosH function ... 113
ArcCotan function ... 114
ArcCotanH function ... 114
ArcCsc function ... 114
ArcCscH function ... 114
ArcSec function ... 114
ArcSecH function ... 114
ArcSin function ... 114
ArcSinH function ... 115
ArcTan2 function ... 115
ArcTanH function ... 115
Ceil function ... 115
CelsToFahr function ... 115
ChgSign function ... 116
CmToInch function ... 116
CopySign function ... 116
CosH function ... 116
Cotan function ... 117
Csc function ... 117
CscH function ... 117
Cterm function ... 117
CycleToRad function ... 117
DeltaPercent function ... 118
DegToRad function ... 118
Evaluate procedure ... 118
FahrToCels function ... 119
Floor function ... 119
FMod function ... 119
Fv function ... 119
GalToLitre function ... 119
GradToRad function ... 120

Hypot function 120
InchToCm function 120
KgToLb function 120
LbToKg function 121
LitreToGal function 121
Log10 function 121
Log2 function 121
LogN function 122
LRotL and LRotR functions 122
Max function 122
Min function 122
Modf function 123
Npv function 123
Percent function 123
Pmt function 123
Power function 124
Pv function 124
RadToCycle function 124
RadToDeg function 124
RadToGrad function 124
Rate function 125
Sec function 125
SecH function 125
Sgn function 125
SinH function 126
Sln function 126
Syd function 126
Tan function 126
TanH function 127
Term function 127

10 The MMedia Unit 128

10.1 TMMedia Object Fields 128
TMMedia.AutoRewind 128
TMMedia.MCErrorProc 128
TMMedia.Notify 129
TMMedia.PlayFullScreen 129
TMMedia.PlayRepeat 129
TMMedia.StopAtClose 129
TMMedia.Wait 129

10.2 TMMedia Object Methods 130
TMMedia.CDMediaIsPresent 130
TMMedia.Close 130
TMMedia.Create 130
TMMedia.ErrorCode 130
TMMedia.GetDevice 131
TMMedia.GetDeviceCaps 131
TMMedia.GetDeviceID 131
TMMedia.GetFirstAudioTrack 132
TMMedia.GetLength 132
TMMedia.GetPos 132
TMMedia.GetStatus 132
TMMedia.GetTimeFormat 133
TMMedia.GetTrackFormat 133
TMMedia.GetTrackLength 133
TMMedia.GetTracksCount 134
TMMedia.GetTrackPos 134
TMMedia.GotoFirstAudioTrack 134
TMMedia.Next 134
TMMedia.Open 135
TMMedia.Pause 135
TMMedia.Play 135
TMMedia.Previous 135
TMMedia.Rec 136
TMMedia.ResetEndPos 136
TMMedia.ResetStartPos 136
TMMedia.Resume 136
TMMedia.Save 137
TMMedia.SetDevice 137
TMMedia.SetDisplayRect 137
TMMedia.SetDisplayWindow 138
TMMedia.SetDoorClosed 138
TMMedia.SetDoorOpen 138
TMMedia.SetEndPos 138
TMMedia.SetPos 138
TMMedia.SetStartPos 139
TMMedia.SetTimeFormat 139
TMMedia.Step 139
TMMedia.Stop 140
TMMedia.Rewind 140

11 The Mouse Unit............141

11.1 Mouse Unit Procedures and Functions............141

ClearMouseHandler procedure............141

DoneMouse procedure............141

GetButtonCount function............142

GetMouseInfo procedure............142

GetMouseX function............142

GetMouseY function............142

HideMouse procedure............142

InitMouse procedure............143

LeftButtonPressed function............143

MickyToText function............143

MiddleButtonPressed function............143

RightButtonPressed function............144

SetMouseHandler procedure............144

SetMousePos procedure............145

SetMouseRange procedure............145

ShowMouse procedure............146

TextToMicky function............146

12 OS/2 API Interface Units............147

13 The Printer Unit............148

14 The Strings Unit............149

14.1 Strings Unit Overloaded Operators............149

14.2 Strings Unit Procedures and Functions............149

Align function............149

AnsiCompareStr function............149

AnsiLowerCase function............150

AnsiCompareText function............150

AnsiStrComp function............150

AnsiStrIComp function............151

AnsiStrLComp function............151

AnsiStrLIComp function............151

AnsiStrLower function............152

AnsiStrUpper function............152

AnsiUpperCase function............152

AppendPathDelimiter function............153

Bin function............153

Dup_CI function 153
Dup_SI function 153
Fix function 153
FloatToStr function 154
Fls function 154
Flt function 154
Hex function 154
HexVal function 154
IntToBin function 155
IntToHex function 155
IntToStr function 155
IsDelimiter function 155
IsPathDelimiter function 155
IsValidIdent function 156
LastDelimiter function 156
LowerCase function 156
QuotedStr function 156
StrAppend procedure 157
StrAppendC procedure 157
StrCat function 157
StrComp function 157
StrCopy function 157
StrDispose procedure 157
StrECopy function 158
StrEnd function 158
StrLCat function 158
StrIComp function 158
StrLComp function 158
StrLCopy function 159
StrLen function 159
StrLIComp function 159
StrLower function 159
StrMove function 160
StrNew function 160
StrPas function 160
StrPCopy function 160
StrPos function 160
StrRScan function 160
StrScan function 161
StrToInt function 161

StrToIntDef function....161
StrUpper function....161
Trim Function....161
TrimLeft Function....162
TrimRight Function....162
Uns function....162
UpperCase function....162
Whl function....162

15 The System Unit....163

15.1 System Unit Constants and Variables....163
_environ variable....163
_psp variable....163
_zero variable....163
CmdLine variable....164
CmdShow variable....164
ErrorAddr variable....164
ExeName variable....165
ExeSize variable....165
ExitCode variable....165
ExitProc variable....165
FarPointer type....166
FileMode variable....166
HeapHandle variable....166
hInstance variable....167
InOutRes variable....167
Input variable....167
IsConsole variable....167
IsLibrary variable....167
LongRec type....168
MaxCardinal constant....168
MaxDWord constant....168
MaxInt constant....168
MaxLongint constant....168
MaxWord constant....169
Output variable....169
RandSeed variable....169
StdErrorHandle variable....169
StdInputHandle variable....169
StdOutputHandle variable....170

Test8086 variable 170
Test8087 variable 170
15.2 System Unit Procedures and Functions 171
Abs function 171
Addr function..... 171
Append procedure 171
ArcTan function..... 172
Assign procedure 172
Assigned function 172
BlockRead procedure 173
BlockWrite procedure..... 173
Break procedure..... 173
ChDir procedure 174
Chr function..... 174
Close procedure 174
Concat function 175
Continue procedure 175
Copy function 175
Cos function..... 176
Dec procedure..... 176
Delete procedure..... 176
Dispose procedure 176
Eof function 177
Eoln function 177
Erase procedure 177
Exit procedure 178
Exclude procedure 178
Exp function 178
FilePos function..... 178
FileSize function..... 179
FillChar procedure..... 179
Flush procedure 179
Frac function..... 179
FreeMem procedure..... 180
GetDir procedure 180
GetMem procedure 180
Halt procedure 181
Hi function..... 181
High function..... 182

Inc procedure 182
Include procedure..........182
Insert procedure 182
Int function..........183
IOResult function..........183
Length function..........183
Ln function..........183
Lo function..........184
LoCase function..........184
Low function..........184
MaxAvail function..........184
MemAvail function..........184
MkDir procedure..........185
Move procedure..........185
New procedure..........185
Odd function 186
Ofs function 186
Ord function..........186
ParamCount function 186
ParamStr function 186
Pi function..........187
Pos function 187
Pred function..........187
Ptr function 188
Random function..........188
Randomize procedure 188
Read procedure 188
ReadLn procedure..........189
Rename procedure 190
Reset procedure..........190
ReWrite procedure..........190
RmDir procedure 191
Round function..........191
RunError procedure 191
Seek procedure..........191
SeekEof procedure..........192
SeekEoln procedure 192
SetString procedure..........192
SetTextBuf procedure 192

Sin function 193
SizeOf function 193
Sqr function 193
Sqrt function 193
Str procedure 194
Succ function 194
Swap function 194
Trunc function 194
Truncate procedure 195
UpCase function 195
Val procedure 195
Write procedure 196
WriteLn procedure 196

16 The Use32 Unit 198

17 Win32 API Interface Units 199

18 The WinCRT Unit 200

18.1 WinCRT Unit Constants and Variables 200
AutoTracking variable 200
CloseOnExit variable 200
Color constants 201
CurOrg variable 201
Cursor variable 201
InactiveTitle variable 202
ScreenSize variable 202
ScrollCrtWindow variable 202
WindowOrg variable 202
WindowSize variable 202
WindowTitle variable 203

18.2 WinCRT Unit Procedures and Functions 203
CursorTo procedure 203
DoneWinCRT procedure 203
InitWinCRT procedure 204
ReadBuf procedure 204
ScrollTo procedure 204
TrackCursor procedure 204
WriteBuf procedure 204
WriteChar procedure 205

19 The WinDos Unit 206

19.1 WinDos Unit Constants and Variables........206
TDataTime type........206
TRegisters type........206
TSearchRec type........207
19.2 WinDos Unit Procedures and Functions........208
CreateDir procedure........208
FileExpand function........208
FileSearch function........208
FileSplit procedure........208
GetArgCount function........209
GetArgStr function........209
GetCurDir function........209
GetEnvVar function........209
RemoveDir procedure........210
SetCurDir procedure........210
20 The ZenTimer Unit........211
20.1 ZenTimer Unit Procedures and Functions........211
CPU_getCPUIdFeatures function........211
CPU_getProcessorSpeed function........211
CPU_getProcessorType function........212
CPU_haveMMX function........213
CPU_have3DNow function........213
LZDelay procedure........214
LZTimerCount function........214
LZTimerLap function........215
LZTiemerOff procedure........215
LZTiemerOn procedure........215
LZTimerResolution function........216
ULZDelay procedure........216
ULZElapsedTime function........216
ULZReadTime function........216
ULZTimerCount function........217
ULZTimerLap function........217
ULZTimerOff procedure........217
ULZTimerOn procedure........217
ULZTimerResolution function........218
ZTimerInit procedure........218

About this guide

This document describes all constants, types, variables, functions and procedures as they are declared in the units that come with TMT Pascal.
Functions and procedures have their own subsections, and for each function or procedure we have the following topics:

Declaration The exact declaration of the function.
Remarks Necessary remarks.
See Also: Cross references to other related functions or commands.

The chapters are ordered alphabetically.

Chapter 1

The Comp Unit

Targets: MS-DOS, OS/2, Win32

The Comp unit provides procedures and functions to manipulate complex numbers, including standard arithmetic operations, relational operators and extended trigonometric functions.

1.1 Comp Unit Types and Overloaded Operators

Following types are defined in the Comp unit:

```
type
  CReal = Extended;

type Complex = record
  re, im: CReal
end;

type
  TComplex = Complex;
```

The Comp unit overloads the following operators

```
overload  +:= = add_cc
overload  +:= = add_cr
overload  +:= = add_cc
overload  -:= = sub_cr
overload  -:= = sub_rc
overload  -:= = sub_rc
overload  *:= = mul_cr
overload  *:= = mul_rc
overload  *:= = mul_rc
overload  /:= = dvi_cr
overload  /:= = div_rc
overload  /:= = div_rc

overload  +:= = addab_cc
overload  +:= = addab_cr
overload  +:= = subab_cc
overload  +:= = subab_cr
overload  +:= = mulab_cc
overload  +:= = mulab_cr
overload  +:= = divab_cc
overload  +:= = divab_cr
overload  =   = eq_cc
overload  =   = eq_cr
overload  =   = eq_rc
overload  <>  = ne_cc
overload  <>  = ne_cr
overload  <>  = ne_rc
```

1.2 Comp Unit Procedures and Functions

add_cc function

The add_cc function is used by the Comp unit.

Declaration:

```
function add_cc(const a: Complex; const b: Complex): Complex;
```

add_cr function

Returns a sum of two arguments of Complex type.

Declaration:

```
function add_cr(const a: Complex; const b: CReal): Complex;
```

Chapter 2

The CRT Unit

Targets: MS-DOS, OS/2, Win32 console

The CRT unit contains routines to control the keyboard and monitor. This unit, with over 20 functions and procedures, allows for a powerful interface to be developed between the program and the user. To the end user, a program's appearance and interface are essential aspects that cannot be ignored. With CRT, programming window displays and keyboard handling is very easy. In the initialization code for CRT, output to the monitor or screen is redirected from the console standard input and output files to the direct screen I/O. Redirection is possible only if the Input and Output file are assigned and reopened.

The CRT units allows one to create OS/2 and Win32 console applications easily.

Only a few differences remain between MS-DOS, OS/2 and Win32 implementations of CRT unit. These differences are described in this chapter.

2.1 CRT Unit Constants and Variables

Color constants

Use these color constants with **TextColor** and **TextBackGround** procedures.

Dark colors (foreground & background):	
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
LightGray	7

Light colors (foreground only) :	
DarkGray	8
LightBlue	9
LightGreen	10
LightCyan	11
LightRed	12
LightMagenta	13
Yellow	14
White	15

For flashing (blinking) text foreground, Blink = 128.

TextMode constants

Use these constants with **TextMode** procedure.

Value	Meaning
BW40	$00;
CO40	$01;
BW80	$02;
CO80	$03;
Mono	$07;
Font8x8	$FF;

CheckBreak variable

Controls user termination of an application using the CRT window.

Declaration:

```
const CheckBreak: Boolean = TRUE;
```

Remarks:

When *CheckBreak* is True, the user can terminate the application at any time by pressing Ctrl-Break or Ctrl+C.

Application cannot be terminated if *CheckBreak* is False.

CheckEOF variable

Controls the end-of-file character checking in the CRT window.

Declaration:

```
const CheckEOF: Boolean = FALSE;
```

DirectVideo variable

Enables and disables direct memory access for Write and WriteLn statements that output to the screen.

Declaration:

```
const DirectVideo: Boolean = TRUE;
```

MS-DOS target:

If *DirectVideo* is TRUE, Write and WriteLn procedures will store characters directly in the video memory, instead of using the BIOS to display them.

Win32 target:

If *DirectVideo* is TRUE, CRT uses simplified code for faster Write and WriteLn procedures execution.

CheckSnow variable

Enables and disables «snow-checking» when storing characters directly in video memory

Declaration:

```
const CheckSnow: Boolean = FALSE;
```

Win32 and OS/2 targets:

The value of this variable is ignored.

LastMode variable

Each time **TextMode** is called, the current video mode is stored in *LastMode*.

Declaration:

```
var LastMode: Word;
```

Remarks:

Also, *LastMode* is initialized at program startup to the then-active video mode.

TextAttr variable

Stores currently selected text attributes.

Declaration:

var TextAttr: Byte := LigthGray;

Remarks:

The text attributes are normally set through calls to **TextColor** and **TextBackGround**.

Win32 target:

A direct change of the *TextAttr* variable will have no effect. Use *TextColor*, *TextBackground, HighVideo* and *LowVideo* procedures instead.

WindMax variable

Stores the screen coordinates of the current window.

Declaration:

```
var WindMax: Word;
```

Remarks:

These variables are set by calls to the **Window** procedure. **WindMin** defines the upper left corner. *WindMax* defines the lower right corner.

The X coordinate is stored in the low byte, and the Y coordinate is stored in the high byte.

For example, `Lo(`*`WindMin`*`)` produces the X coordinate of the left edge, and `Hi(`*`WindMax`*`)` produces the Y coordinate of the bottom edge.

The screen's upper left corner corresponds to (X,Y) = (0,0), but for coordinates sent to **Window** and **GotoXY**, the upper left corner is at (1,1).

WindMin variable

Stores the screen coordinates of the current window.

Declaration:

```
var WindMin: Word;
```

Remarks:

These variables are set by calls to the **Window** procedure. *WindMin* defines the upper left corner. **WindMax** defines the lower right corner.

The X coordinate is stored in the low byte, and the Y coordinate is stored in the high byte.

For example, *Lo(WindMin)* produces the X coordinate of the left edge, and *Hi(WindMax)* produces the Y coordinate of the bottom edge.

The screen's upper left corner corresponds to (X,Y) = (0,0), but for coordinates sent to **Window** and **GotoXY**, the upper left corner is at (1,1).

2.2 CRT Unit Procedures and Functions

AssignCrt procedure

Associates a text file with the CRT window.

Declaration:

```
procedure AssignCrt(var f: Text);
```

Remarks:

AssignCrt works exactly like the Assign standard procedure; however, no file name is specified. Instead, the text file is associated with the CRT.

This allows for faster output (and input) than would normally be possible using the standard output (or input) procedure.

ClrEOL procedure

Associates a text file with the CRT window.

Clears all characters from the cursor position to the end of the line without moving the cursor from it’s initial position.

Declaration:

```
procedure ClrEOL;
```

Remarks:

All character positions are set to blanks with the currently defined text attributes. Thus, if *TextBackground* is not black, the current cursor position to the right edge becomes the background color.

ClrEol is window-relative.

ClrScr procedure

Clears the active windows and returns the cursor to the upper-left corner.

Declaration:

```
procedure ClrScr;
```

Remarks:

Sets all character positions to blanks with the currently defined text attributes. Thus, if TextBackground is not black, the entire screen becomes the background color. This also applies to characters cleared by **ClrEOL**, **InsLine** and **DelLine**, and to empty lines created by scrolling.

ClrScr is window-relative.

Delay procedure

Delays a specified number of milliseconds.

Declaration:

```
procedure Delay(Ms: Word);
```

Remarks:

Ms specifies the time, in milliseconds of the delay.

Delay is only an approximation, and therefore, the delay period will not last for the exact number of *Ms* milliseconds.

DelLine procedure

Deletes the line containing the cursor.

Declaration:

```
procedure DelLine;
```

Remarks:

The line containing the cursor is deleted, and all lines below are shifted by one line. A new line is then added at the bottom.

All character positions are set to blanks using the currently defined text attributes. Thus, if the TextBackground is not black, the new line becomes the background color.

GetCharXY function

Reads a character from the screen.

Declaration:

```
function GetCharXY(X, Y: Longint): Char;
```

GotoXY procedure

Shifts the cursor over to the given coordinates within the virtual screen.

Declaration:

```
procedure GotoXY(X, Y: Byte);
```

Remarks:

The upper-left corner of the virtual screen corresponds to (1, 1).

HideCursor procedure

Hides the text cursor.

Declaration:

```
procedure HideCursor;
```

See also: **ShowCursor**

HighVideo procedure

Selects high-intensity characters.

Declaration:

```
procedure HighVideo;
```

Remarks:

HighVideo sets the high intensity bit of **TextAttr**’s fore-ground color, thus mapping colors 0-7 onto colors 8-15.

InsLine procedure

Inserts an empty line at the cursor position.

Declaration:

```
procedure InsLine;
```

Remarks:

All lines below the inserted line are moved down one line, and the bottom line is scrolled off of the screen.

All character positions are set to blanks with the currently defined text attributes; therefore, if the TextBackground is not black, the new line will become the background color.

KeyPressed function

Returns True if a key has been pressed on the keyboard.

Declaration:

```
function KeyPressed: Boolean;
```

Remarks:

The key code can be read using the **ReadKey** function.

LowVideo procedure

Selects low intensity characters.

Declaration:

```
procedure LowVideo;
```

Remarks:

LowVideo clears the high-intensity bit of **TextAttr**’s foreground color, thus mapping colors 8 to 15 onto colors 0 to 7.

NormVideo procedure

Selects the original text attribute read from the cursor location at startup.

Declaration:

```
procedure NormVideo;
```

Remarks:

NormVideo restores **TextAttr** to the value it had when the program was started.

NoSound procedure

Turns off the computer’s internal speaker.

Declaration:

```
procedure NoSound;
```

ReadKey function

Reads a character or an extended scan code from the keyboard.

Declaration:

```
function ReadKey: Char;
```

Remarks:

The character is not echoed to the screen.

SetScreenSize procedure

Defines the custom size in characters of the text screen.

Declaration:

```
procedure SetScreenSize(Cols, Rows: DWord);
```

ShowCursor procedure

Shows the text cursor.

Declaration:

```
procedure ShowCursor;
```

See also: **HideCursor**

Sound procedure

Starts the internal speaker.

Declaration:

```
procedure Sound(Hz: DWord);
```

Remarks:

Hz specifies the frequency of the emitted sound in hertz. The sound continues until explicitly turned off by a call to **NoSound**.

Windows'95, Windows'98:

The *Sound* function ignores the `Hz` parameters. On computers with a sound card, the function plays the default sound event. On computers without a sound card, the function plays the standard system beep.

TextBackGround procedure

Selects the background color.

Declaration:

```
procedure TextBackground(Color: Byte);
```

Remarks:

Color is an integer expression in the range 0..7, corresponding to one of the first eight text color constants. *TextBackground* sets bits 4-6 of **TextAttr** to *Color*.

The background of all characters subsequently written will be in the specified color.

See also: **Color Constants**

TextColor procedure

Selects the foreground character color.

Declaration:

```
procedure TextColor(Color: Byte);
```

Remarks:

Color is an integer expression in the range 0..15, corresponding to one of the text color constants defined in CRT.

MS-DOS target:

TextColor sets bits 0-3 to *Color*. If *Color* is greater than 15, the blink bit (bit 7) is also set; otherwise, it is cleared.

See also: **Color Constants**

TextMode procedure

Selects a specific text mode.

Declaration:

```
procedure TextMode(Mode: Integer);
```

Remarks:

When *TextMode* is called, the current window is reset to the entire screen, **DirectVideo** is set to True, **CheckSnow** is set to True if a color mode was selected, the current text attribute is reset to normal corresponding to a call to **NormVideo**, and the current video is stored in

LastMode. In addition, **LastMode** is initialized at program startup to the then-active video mode.

Specifying **TextMode(LastMode)** causes the last active text mode to be re-selected. This is useful when you want to return to text mode after using a graphics package, such as Graph unit.

See also: **TextMode Constants**

WhereX function

Returns the CP's X coordinate of the current cursor location.

Declaration:

```
function WhereX: Byte;
```

WhereY function

Returns the CP's Y coordinate of the current cursor location.

Declaration:

```
function WhereY: Byte;
```

Window procedure

Defines a text window on the screen.

Declaration:

```
procedure Window(X1, Y1, X2, Y2: Byte);
```

Remarks:

X1 and *Y1* are the coordinates of the upper left corner of the window, and *X2* and *Y2* are the coordinates of the lower right corner. The upper left corner of the screen corresponds to (1, 1). The minimum size of a text window is one column by one line. If the coordinates are invalid in any way, the call to **Window** is ignored.

The default window is (1, 1, 80, 25) in 25-line mode, and (1, 1, 80, 43) in 43-line mode, corresponding to the entire screen.

All screen coordinates (except the window coordinates themselves) are relative to the current window. For instance, `GotoXY(1, 1)` will always position the cursor in the upper left hand corner of the current window.

Many CRT procedures and functions are window-relative, including **ClrEOL**, **ClrScr**, **DelLine**, **GotoXY**, **InsLine**, **WhereX**, **WhereY, Write** and **Writeln**.

WindMin and **WindMax** store the current window definition. A call to the **Window** procedure always moves the cursor to (1, 1).

WriteAttr procedure

Writes given string *S* in attributes/characters format to the screen at (*X*,*Y*).

Declaration:

```
procedure WriteAttr(X, Y: Longint; Var S; Len: Longint);
```

The Microsoft® Win32® and IBM® OS/2® application programming interfaces (API) provide consoles that manage input and output (I/O) for character-mode applications (applications that do not provide their own graphical user interface).

Chapter 3

The Debug Unit

Targets: MS-DOS, OS/2, Win32

This module prints out the error code and the call stack in case of a run-time error. The stack is printed as follows:

```
RunError #201 (range check error)
Calls stack:
SYSTEM.BOUND_ERROR [chk_fun.inp(21) at 0000000A]
TEST.ASSN [TEST.PAS(61) at 00000015]
TEST.TEST [TEST.PAS(82) at 0000001D]
```

To use Debug, simply list it in the uses clause of the main program. Using the Debug UNIT increases the .EXE module size

⚠ A call of a procedure with a NIL address is currently diagnosed as an arithmetic overflow, or (under the PMODE extender) causes a GP Fault or other trap.

Chapter 4

The DOS Unit

Targets: MS-DOS, OS/2, Win32

The Dos unit allows easy access to most of the functions provided by the MS-DOS operating system from a 32-bit protected mode application. Also the Dos unit emulates MS-DOS functions under OS/2 and Win32 using the standard API, provided by the OS/2 and Win32 operating systems. Operations such as find file, disk size or status, time and date, get environment strings and more are provided by the Dos unit. In total, over 50 functions and procedures are available. For more information about MS-DOS consult your DOS operating system reference manual.

4.1 Dos Unit Constants and Variables

Flag constants

The Flag constants (fXXXX) test individual flag bits in the Flags register after a call to **Intr** or **MsDos**.

Constant	Value
fCarry	$0001
fParity	$0004
fAuxiliary	$0010
fZero	$0040
fSign	$0080
fOverflow	$0800

File-mode constants

File-handling procedures use fmXXXX constants when opening and closing disk files.

The Mode fields of TFileRec and TTextRec will contain one of these values:

Constant	Value
fmClosed	$D7B0
fmInput	$D7B1
fmOutput	$D7B2
fmInOut	$D7B3

File-attribute constants

These constants test, set, and clear file-attribute bits in connection with the **GetFAttr**, **SetFAttr**, **FindFirst**, and **FindNext** procedures.

These constants are additive. The faAnyFile constant is the sum of all attributes.

Constant	Value
ReadOnly	$01
Hidden	$02
SysFile	$04
VolumeID	$08
Directory	$10
Archive	$20
AnyFile	$3F

Win32 Constant	Value
faReadOnly	WINDOWS.FILE_ATTRIBUTE_READONLY
faHidden	WINDOWS.FILE_ATTRIBUTE_HIDDEN
faSysFile	WINDOWS.FILE_ATTRIBUTE_SYSTEM
faVolumeID	$08
faDirectory	WINDOWS.FILE_ATTRIBUTE_DIRECTORY
faArchive	WINDOWS.FILE_ATTRIBUTE_ARCHIVE
faAnyFile	WINDOWS.FILE_ATTRIBUTE_NORMAL

DataTime type

The **UnpackTime** and **PackTime** procedures use variables of type *DateTime* to examine and construct 4-byte, packed date-and-time values for the **GetFTime**, **SetFTime**, **FindFirst**, and **FindNext** procedures:

Declaration:

```
type
  DateTime = record
    Year, Month, Day, Hour,
    Min, Sec: Word;
  end;
```

DosError variable

DosError is used by many of the routines in the Dos unit to report errors.

Declaration:

```
var DosError: Integer;
```

Remarks:

The values stored in DosError are operating system dependent error codes.

Registers type

The **Intr** and **MsDos** procedures use variables of type *Registers* to specify the input register contents and examine the output register contents of a software interrupt.

Declaration:

```
type Registers =
  record
  case Integer of
    1: (edi, esi, ebp, _res, ebx, edx, ecx, eax: Longint;
        flags, es, ds, fs, gs, ip, cs, sp, ss: Word);
    2: (_dmy2: array [0..15] of byte; bl, bh, b1, b2, dl,
        dh, d1, d2, cl, ch, c1, c2, al, ah: Byte);
    3: (di, i1, si, i2, bp, i3, i4, i5, bx, b3, dx, d3, cx,
        c3, ax: Word);
  end;
```

SearchRec type

The **FindFirst** and **FindNext** procedures use variables of type *SearchRec* to scan directories:

Declaration:

MS-DOS target:

```
type
  SearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : string[12];
end;
```

OS/2 target:

```
type
  SearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : string;
end;
```

Win32 target:

```
type
  SearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : TFileName;
    ExcludeAttr: Longint;
    FindHandle: THandle;
    FindData: TWin32FindData;
  end;
```

Information for each file found by FindFirst or FindNext is reported back in a *SearchRec*.

Field	Meaning
Attr	File’s attributes
Time	File’s packed date and time
Size	File’s size, in bytes
Name	File’s name

The *Fill* field is reserved by DOS and should never be modified.

4.2 Dos Unit Procedures and Functions

DiskFree function

Returns the number of free bytes on a specified disk drive.

Declaration:

```
function DiskFree(Drive: Byte): Longint;
```

Remarks:

Drive is the drive to check where A: = 1, B: = 2, and so on. If *Drive* is zero then the current drive is used.

DiskFree is useful in determining whether there is enough disk space for disk output. A message can be displayed if there is not enough disk space available.

Example:

```
uses Dos;
begin
  if (DiskFree(0) < 100000) and IsFixed(0) then
  begin
    WriteLn('Insufficient Disk Space');
    Halt(1);
  end;
  ...
end.
```

See also: **DiskSize**

DiskSize function

Returns the total size, in bytes, of a specified disk drive.

Declaration:

```
function DiskSize(Drive: Byte): Longint;
```

Drive is the drive to check where A: = 1, B: = 2, and so on. If *Drive* is zero then the current drive is used.

Example:

```
size := DiskSize(0);  // size of current drive
size := DiskSize(3);  // size of drive C:
```

See also: **DiskFree**

DosExitCode function

Returns the exit code of a subprocess.

Declaration:

```
function DosExitCode: Word;
```

See also: **Exec**

DosVersion function

Returns the OS version number.

Declaration:

```
function DosVersion: Word;
```

EnvCount function

Returns the number of strings contained in the system environment.

Declaration:

```
function EnvCount: Integer;
```

See also: **EnvStr, GetEnv**

EnvStr function

Returns a specified environment string.

Declaration:

```
function EnvStr(Index: Integer): string;
```

Remarks:

Index is the number of the environment variable, for instance, the first variable is one, second is two, and so on. An invalid index returns an empty string.

EnvStr returns a string in the form of (VarName=String). If the order of system environment variables is unknown then use **GetEnv** to retrieve a variable by name.

Example:

```
uses Dos;
var
  i : Integer;
begin
  for i := 1 to EnvCount do
    WriteLn(EnvStr(i));
end.
```

See also: **EnvCount, GetEnv**

Exec procedure

Executes a specified program with a specified command line.

Declaration:

```
procedure Exec(Path, CmdLine: string);
```

Remarks:

Path is the drive, directory, and program name to execute. *CmdLine* contains the command line arguments.

Exec transfers control to the program specified by Path. Memory allocation is not modified by *Exec*. Upon completion of *Exec* use **DosExitCode** to determine the exit code of the program. *Exec* also sets the value of *DosError* if an error occurred.

Exec does not execute programs that require File Control Blocks (FCBs).

Example:

```
begin
  Exec('PROGRAM.EXE','');
  WriteLn(Hi(DosExitCode),'.', Lo(DosExitCode),'.',DosError);
end.
```

See also: **DosExitCode**

FExpand function

Expands a file name into a fully-qualified file name.

Declaration:

```
function FExpand(Path: PathStr): PathStr;
```

FindFirst procedure

Searches the specified directory for the matching file.

Declaration:

```
procedure FindFirst(Path: PChar; Attr: Word; var F:
TSearchRec);
```

Remarks:

Path is the drive and directory to search in and the file name to search for. Wildcards are allowed, for instance, ‘MYFILE??.*’.

Attr contains the file attributes to include in the search in addition to all normal files.

FindFirst is used in conjunction with **FindNext**. Use **FindNext** to locate any addition files matching the search criteria. All errors are reported in *DosError* , which is a variable defined in the Dos unit.

Example:

```
program DirList;
uses Dos;
var
  TotalDirCnt: Longint;
procedure List(Path : String);
var
  DirSearchRec: SearchRec;
begin
  if (Path[Length(Path)] <> '\') then Path := Path + '\';
  FindFirst(Path + '*.*', AnyFile, DirSearchRec);
  while DosError = 0 do
  begin
    if (DirSearchRec.Name <> '.') and (DirSearchRec.Name <>
'..') and
       ((DirSearchRec.Attr and Directory) <> 0)
    then
    begin
      Inc(TotalDirCnt);
      Writeln(Path + DirSearchRec.Name);
      List(Path + DirSearchRec.Name);
    end;
    FindNext(DirSearchRec) ;
  end;
end;

begin
  TotalDirCnt := 0;
  List('C:\');
  Writeln;
  Writeln('Total number of directories = ', TotalDirCnt);
end.
```

See also: **Fsearch**

FindNext procedure

Finds the next entry that matches the name and attributes specified in an earlier call to **FindFirst**.

Declaration:

```
procedure FindNext(var F: TSearchRec);
```

Remarks:

FindNext is used in conjunction with **FindFirst**. Use *FindNext* to locate any addition files matching the search criteria defined by a prior call to **FindFirst**. *F* must be the same variable that was passed to **FindFirst**. All errors are reported in *DosError*, which is a variable defined in the Dos unit.

Example:

```
uses Dos;
var
  DirRec : SearchRec;
begin
  FindFirst('C:/*.*',AnyFile,DirRec);
  while DosError = 0 do
    WriteLn(DirRec.Name);
    FindNext(DirRec);
  end;
end.
```

See also: **Fsearch**

FSearch function

Searches for a file.

Declaration:

```
function FSearch(Path: PathStr; DirList: string): PathStr;
```

Remarks:

Path is of type PathStr which is defined in the Dos unit. *DirList* is a list of the directories to include in the search each delimited with a semicolon (;).

FSearch returns the directory and file name if the file has been located. If *Path* is not found then an empty string is returned. *FSearch* always begins with the current directory and then checks the directories listed in *DirList* in the order that they appear.

Example:

```
uses Dos,Strings;
var
  DosPath,
  TreePath: String;
  File    : PathStr;
begin
  DosPath := GetPath;
  File    := 'TREE.COM';
  TreePath:= FSearch(File, DosPath);
  if Empty(TreePath) then Halt(1);
  TreePath := AppendBKSlash(TreePath);
  Exec(TreePath + File,'');
end.
```

See also: **FindFirst, Fexpand, Fsplit**

FSplit procedure

Splits a file name into its three components.

Declaration:

```
procedure FSplit(Path: PathStr; var Dir: DirStr; var Name:
NameStr; var Ext: ExtStr);
```

Remarks:

Use this procedure to break down a file specification into three parts: path, file name, and file extension. *Path* is of type *PathStr*, which is defined in the Dos unit. *Dir* returns the path or directory part of *Path*. *Name* returns the actual file name without extension. *Ext* returns the file extension preceded by a period (.).

It is possible that one or more of the components be returned empty. This occurs if *Path* contains no such component. For instance, if there is no path, *Dir* is empty.

Example:

```
uses Dos;
var
  Fi    : File;
  Direc : DirStr;
  Fname : NameStr;
  Exten : ExtStr;
begin
  FSplit(ParamStr(1), Direc, Fname, Exten);
  if Fname = '' then Halt(1);
  Assign(Fi, ParamStr(1));
  Erase(Fi);
end.
```

GetCBreak procedure

Returns the state of Ctrl-Break checking in DOS.

Declaration:

```
procedure GetCBreak(var Break: Boolean);
```

Remarks:

State returns True if Ctrl-Break is enabled, otherwise False.

With Ctrl-Break checking enabled, all I/O calls (console, printer, and communications) are checked. To set Ctrl-Break either on or off, use **SetCBreak**.

Win32 target:

GetCBreak does not change the *Break* variable, since this service is not provided by the Windows API.

GetDate procedure

Returns the current date set in the operating system.

Declaration:

```
procedure GetDate(var Year, Month, Day, DayOfWeek: Word);
```

GetEnv function

Returns the value of a specified environment variable.

Declaration:

```
function GetEnv(VarName: string): string;
```

Remarks:

VarName is the name of the variable to retrieve. If *VarName* does not exist as an environment variable then an empty string is returned.

GetEnv returns the string assigned to the environment variable.

Example:

```
s := GetEnv(‘COMSPEC’);  // Returns COMSPEC
```

GetFAttr procedure

Returns the attributes of a file.

Declaration:

```
procedure GetFAttr(var F; var Attr: Word);
```

Remarks:

F is a file variable, either typed, untyped, or text file that is not open. *Attr* contains the file attributes.

The file associated with *F* must be closed. *Attr* should be examined by ANDing it with the file attribute constants, which are defined in the Dos unit.

Errors are reported in *DosError* , a variable defined in the Dos unit. For more information about file attributes consult your DOS reference manual.

Example:

```
uses Dos;
var
  Fi  : Text;
  Attr: Word;
begin
  Assign(Fi,'THEFILE.DOC');
  GetFAttr(Fi, Attr);
  if (Attr and ReadOnly) <> 0 then SetFAttr(Fi,(Attr xor
ReadOnly));
end.
```

See also: **SetFAttr**

GetFTime procedure

Returns the date and time a file was last written.

Declaration:

```
procedure GetFTime(var F; var Time: Longint);
```

Remarks:

The file associated with *F* must be open. *Time* can be unpacked with *UnPackTime*. Errors are reported in *DosError*, a variable defined in the Dos unit.

Example:

```
uses Dos;
var
  Fi  : Text;
  Time: Longint;
  DT  : DateTime;
begin
  Assign(Fi, 'DATA.BAK');
  Reset(Fi);
  GetFTime(Fi, Time);
  UnPackTime(Time, DT);
  WriteLn('Year:   ', DT.Year);
  WriteLn('Month:  ', DT.Month);
  WriteLn('Date:   ', DT.Day);
  Close(Fi);
end.
```

See also: **SetFTime**

GetIntVec procedure

Returns the address stored in a specified interrupt vector.

Declaration:

```
procedure GetIntVec(IntNo: Byte; var Vector: Pointer);
procedure GetIntVec(IntNo: Byte; var Vector: FarPointer);
```

Remarks:

Note that *GetIntVec* returns the address of a protected mode interrupt. To get the vector of a real mode interrupt use *GetRealIntVec*. Interrupts may occur while in protected mode or while in real mode.

See also: **SetIntVec, FarPointer**

GetTime procedure

Returns the current time set in the operating system.

Declaration:

```
procedure GetTime(var Hour, Minute, Second, Sec100: Word);
```

Remarks:

Ranges of the values returned are *Hour* [0..23], *Minute* [0..59], *Second* [0..59], and *Sec100* (hundredths of seconds) [0..99].

Example:

```
var
  Hour, Minute, Second, Sec100: Word;
```

```
begin
  GetTime(Hour, Minute, Second,Sec100);
  WriteLn('Current time is: ', Hour, ':', Minute, ':', Second,
':', Sec100);
end.
```

See also: **SetTime**

GetVerify procedure

Returns the state of the verify flag in DOS.

Declaration:

```
procedure GetVerify(var State: Boolean);
```

Remarks:

State is True if the DOS verify flag is enabled, otherwise false.

With the DOS verify flag enabled, all output to disk is verified to ensure data integrity. Otherwise, all output is not verified. To set the status of the DOS verify flag, use **SetVerify**.

Win32 target:

GetVerify does not change the *State* variable, since this service is not provided by the Windows API.

Intr procedure

Executes a specified software interrupt.

Declaration:

```
procedure Intr(IntNo: Byte; var Regs: TRegisters);
```

Remarks:

Before calling *Intr*, load *Regs* with the appropriate parameters needed for the interrupt. *Regs* returns the values of the registers after the interrupt call. Calls that depend on ESP and SS cannot be executed. For more information about software interrupt calls consult your BIOS and DOS reference manual.

Note that all segment resgisters (DS,ES,FS,GS) must contain valid segment descriptors or be set to zero prior to calling *Intr*. All interrupt calls that require an offset must be passed a 32 bit offset.

Example:

```
uses Dos;
function GetVideoMode: Byte;
var
  Regs: Registers;
begin
  Regs.AX := $0F00;
  Regs.DS := DSeg;  Regs.ES := 0;
  Regs.FS := 0;     Regs.GS := 0;
  Intr($10, Regs);
  GetVideoMode := Regs.Al;
end.
```

Keep procedure

This procedure is only a stub procedure and **always** causes a run-error message.

Declaration:

```
procedure Keep(ExitCode: Word);
```

MsDos procedure

Executes a DOS function call.

Declaration:

```
procedure MsDos(var Regs: TRegisters);
```

Remarks:

Load *Regs* with the proper parameters before calling MS-DOS. Regs returns the values of the registers after the interrupt. Notice that *TRegisters* type allows access to 32 bit registers. Calls to DOS that depend on ESP and SS cannot be executed. To an interrupt other than $21, use **Intr**. For more information about DOS interrupt calls consult your DOS reference manual.

Note that all segment registers (DS, ES, FS, GS) must contain valid segment descriptors or be set to zero prior to using MS-DOS.

Example:

```
uses Dos;
procedure DispString(DispStr: String);
var
  Regs    : Registers;
begin
  DispStr := DispStr + #0;
  Regs.AX := $0900;
  Regs.EDX:= DWord(@DispStr) + 1;
  Regs.DS := DSeg;
  Regs.ES := 0;
  Regs.FS := 0;
  Regs.GS := 0;
  MsDos(Regs);
end.
```

PackTime procedure

Converts a *DateTime* record.

Declaration:

```
procedure PackTime(var T: DateTime; var Time: Longint);
```

Remarks:

PackTime can be used in conjunction with **SetFTime**. To unpack a four byte packed date time into a *DateTime* record, use **UnpackTime**.

SetCBreak procedure

Sets the state of Ctrl-Break checking.

Declaration:

```
procedure SetCBreak(Break: Boolean);
```

Remarks:

With Ctrl-Break checking enabled, all I/O calls (console, printer, and communications) are checked. To get Ctrl-Break status, use **GetCBreak**.

Win32 target:

SetCBreak does nothing, since this service is not provided by the Windows API.

SetDate procedure

Sets the current date

`procedure SetDate(Year, Month, Day: Word);` in the operating system.

Declaration:

Remarks:

Invalid dates are ignored by the operating system. To get the operating system date use **GetDate**.

Windows NT:

The *`SetDate`* function fails if the calling process does not have the SE_SYSTEMTIME_NAME privilege. This privilege is disabled by default. Use the **AdjustTokenPrivileges** function to enable this privilege and again to disable it after the time has been set.

SetFAttr procedure

Sets the attributes of a file.

Declaration:

```
procedure SetFAttr(var F; Attr: Word);
```

Remarks:

The file associated with *F* must be closed. *Attr* should be formed by ORing it with the file attribute constants, which are defined in the Dos unit.

Errors are reported in *DosError*, a variable defined in the Dos unit. For more information about file attributes consult your DOS reference manual.

Win32 target:

The file handle must have been created with GENERIC_WRITE access.

Example:

```
uses Dos;
var
  Fi  : File;
  Attr: Word;
begin
  Assign(Fi, 'SECRET.DOC');
  Attr := Hidden or ReadOnly;
  SetFAttr(Fi, Attr);
  WriteLn('SECRET.DOC is now hidden.');
end.
```

See also: **GetFAttr**

SetFTime procedure

Sets the date and time a file was last written.

Declaration:

```
procedure SetFTime(var F; Time: Longint);
```

The file associated with *F* must be open. A packed date and time stamp can be created with **PackTime**. Errors are reported in *DosError*, a variable defined in the Dos unit. For more information about a file’s packed date and time stamp consult your DOS reference manual.

Win32 target:

The file handle must have been created with GENERIC_WRITE access.

Example:

```
uses Dos;
var
  Fi  : Text;
  DT  : DateTime;
  Time: Longint;
begin
  Assign(Fi, 'FUTURE.DOC');
  Reset(Fi);
  with DT do begin
    Year  := 2010;
    Month := 3;
    Day   := 31;
    Hour  := 2;
    Min   := 45;
    Sec   := 22;
  end;
  PackTime(DT, Time);
  SetFTime(Fi, Time);
  Close(Fi);
end.
```

SetIntVec procedure

Sets a specified interrupt vector to a specified address.

Declaration:

```
procedure SetIntVec(IntNo: Byte; Vector: Pointer);
procedure SetIntVec(IntNo: Byte; Vector: FarPointer);
```

Remarks:

Interrupts may occur while in protected mode or while in real mode.

Example:

```
program Timer;
uses Dos, Crt;
var  Int1CSave: FarPointer;
     Time     : LongInt;
// timer handler
procedure TimerHandler(eip,eax,ecx,edx,ebx,esp,ebp,esi,edi:
Dword; gs,fs,es: Word); interrupt;
var StoreX, StoreY: Word;
begin
  Inc(time);
  Store X:= WhereX;
  Store Y:= WhereY;
  GotoXY(1,1);
  Write(time);
  GotoXY(StoreX, StoreY);
  Port[$20] := $20;
end;
// main program
begin
  ClrScr;
  Time := 0;
  GetIntVec($1C, Int1CSave);
  SetIntVec($1C, @TimerHandler);
  Writeln;
  Writeln('Type something and press "ENTER" to exit');
  Readln;
  SetIntVec($1C, Int1CSave);
end.
```

See also: **GetIntVec, FarPointer**

SetTime procedure

Sets the current time in the operating system.

Declaration:

```
procedure SetTime(Hour, Minute, Second, Sec100: Word);
```

Remarks:

Invalid values are ignored. To get the current operating system time use **GetTime**.

Windows NT:

The `SetTime` function fails if the calling process does not have the SE_SYSTEMTIME_NAME privilege. This privilege is disabled by default. Use the **AdjustTokenPrivileges** function to enable this privilege and again to disable it after the time has been set.

SetVerify procedure

Sets the state of the verify flag in DOS.

Declaration:

```
procedure SetVerify(Verify: Boolean);
```

Win32 target:

This function has no effect (ignored), since this service is not provided by the Windows API.

See also: **GetVerify**

SwapVectors procedure

The *SwapVectors* function does nothing and is provided for compatibility with Borland Pascal.

Declaration:

```
procedure SwapVectors;
```

UnpackTime procedure

Converts a Longint to a record.

Declaration:

```
procedure UnpackTime(Time: Longint; var DataTime: TDateTime);
```

Remarks:

UnPackTime can be used in conjunction with **GetTime**, **FindFirst**, and **FindNext**. These routines return a file’s four byte packed date and time stamp. To pack a *DateTime* record, use **PackTime**.

Example:

```
uses Dos;
var
  Fi  : Text;
  Time: Longint;
  DT  : DateTime;
begin
  Assign(Fi,'USER.DOC');
  Reset(Fi);
  GetFTime(Fi,Time);
  UnPackTime(Time,DT);
  WriteLn('Year:   ',DT.Year);
  WriteLn('Month:  ',DT.Month);
  WriteLn('Date:   ',DT.Day);
  Close(Fi);
end.
```

Chapter 5

The DPMI Unit

Targets: MS-DOS only

The DOS Protected Mode Interface (DPMI) was defined to allow DOS programs to access the extended memory of PC architecture computers while maintaining system protection. DPMI defines a specific subset of DOS and BIOS calls that can be made by protected mode DOS programs. It also defines a new interface via software interrupt 31h that protected mode programs use to allocate memory, modify descriptors, call real mode software, etc. Any operating system that currently supports virtual DOS sessions should be capable of supporting DPMI without affecting system security. Some DPMI implementations can execute multiple protected mode programs in independent virtual machines. Thus, DPMI applications can behave exactly like any other standard DOS program and can, for example, run in the background or in a window (if the environment supports these features). Programs that run in protected mode also gain all the benefits of virtual memory and can run in a 32-bit flat model if desired. The DPMI unit is intended for simplified access to DPMI functions from a Pascal program.

5.1 DPMI Unit Types

TDescriptor type

Declaration:

```
type
  TDescriptor = record
  SegmentLimit: Word;
  BaseAddressL: Word;
  BaseAddressH: Byte;
  FlagsL:       Byte;
  FlagsH:       Byte;
  BaseAddressU: Byte;
end;
```

TRmRegs type

Declaration:

```
type
  TRmRegs = record
   case integer of
```

```
     1: (edi,esi,ebp,_res,ebx,edx,ecx,eax: Longint;
         flags,es,ds,fs,gs,ip,cs,sp,ss: Word);
     2: (_dmy2: array [0..15] of byte);
     3: (bl,bh,b1,b2,dl,dh,d1,d2,cl,di,i1,si,i2,bp,i3,i4,i5,bx,
         b3,dx,d3,cx,c3,ax: Word);
end;
```

5.2 DPMI Unit Procedures and Functions

AllocateDescriptors function

This function is used to allocate one or more descriptors from the task's Local Descriptor Table (LDT). The descriptor(s) allocated must be initialized by the application.

Declaration:

```
function AllocateDescriptors(NumberOfDescriptors: Word): Word;
```

Remarks:

Returns base selector if successful or zero if failed.

See also: **FreeDescriptor**

AllocDosMemoryBlock function

This function will allocate a block of memory from the DOS free memory pool. It returns both the real mode segment and one or more descriptors that can be used by protected mode applications to access the block.

Declaration:

```
function AllocDOSmemoryBlock(SizeInBytes: DWord): DWord;
```

Remarks:

Returns the paragraph-segment value in its high-order word and a selector in its low-order word if successful. Otherwise returns zero.

See also: **FreeDosMemoryBlock, ResizeDosMemoryBlock**

AllocRealModeCallBack function

This function is used to obtain a unique real mode SEG:OFFSET that will transfer control from a real mode to a protected mode procedure.

Declaration:

```
function AllocateRealModeCallBack(HandlerAddr, RegsAddr:
Pointer; var HndSeg: Word; var HndOfs: DWord): Boolean;
```

Remarks:

Returns True if successful.

See also: **FreeRealModeCallBack**

AllocateSpecificDescriptor function

This function is used to allocate one specific LTD descriptor.

Declaration:

```
function AllocateSpecificDescriptor(Selector: Word): Boolean;
```

Remarks:

Returns True if successful.

See also: **FreeDescriptor**

CallRealModeFar procedure

This function calls a real mode procedure. The called procedure must execute a far return when it completes.

Declaration:

```
function CallRealModeFar(var Regs: TRmRegs): Boolean;
```

Remarks:

Returns True if successful.

See also: **TRmRegs type**

CallRealModeIRet procedure

This function calls a real mode procedure. The called procedure must execute an **iret** when it completes.

Declaration:

```
function CallRealModeIRet(var Regs: TRmRegs): Boolean;
```

Remarks:

Returns True if successful.

See also: **TRmRegs type**

ClearRmRegs procedure

This procedure clears (fills with zero) the Real Mode registers structure. You must do it before you call any function, which uses it!

Declaration:

```
procedure ClearRmRegs(var Regs: TRmRegs);
```

See also: **RealModeInt, TRmRegs type**

CreateCodeAlias function

This function will create a code descriptor that has the same base and limit as the specified code segment descriptor.

Declaration:

```
function CreateCodeAlias(Selector: Word): Word;
```

Remarks:

Returns alias descriptor if successful.

CreateCodeDescriptor function

This function is used to allocate one code descriptor from the task's Local Descriptor Table (LDT) with specified *Base* and *Limit*.

Declaration:

```
function CreateCodeDescriptor(Base, Limit: DWord): Word;
```

Remarks:

Returns code selector if successful. Otherwise returns zero.

See also: **FreeDescriptor**

CreateDataAlias function

This function will create a code descriptor that has the same base and limit as the specified code segment descriptor.

Declaration:

```
function CreateDataAlias(Selector: Word): Word;
```

Remarks:

Returns alias if selector is successful. Otherwise returns zero.

See also: **FreeDescriptor**

CreateDataDescriptor function

This function is used to allocate one data descriptor from the task's Local Descriptor Table (LDT) with specified *Base* and *Limit*

Declaration:

```
function CreateDataDescriptor(Base, Limit: DWord): Word;
```

Remarks:

Returns data selector if successful, or zero if not.

See also: **FreeDescriptor**

DosMemoryAlloc function

This function will allocate a block of memory from the DOS free memory pool. It returns real mode only segment of allocated DOS memory block.

Declaration:

```
function DOSMemoryAlloc(SizeInBytes: DWord): Word;
```

Remarks:

Returns the segment value if successful. Otherwise returns zero.

To get access to allocated DOS memory block, multiply the returned paragraph-segment by 16.

Example:

```
function MkDOSPointer (Segment: Word): Pointer;
begin
  Result := DWord(Segment)*16;
end.
```

This function makes a protected mode pointer on a given segment of allocated DOS memory block.

See also: **DosMemoryFree**

DosMemoryFree function

This function frees memory that was allocated through the **DosMemoryAlloc** function.

Declaration:

```
function DOSMemoryFree(Segment: Word): Boolean;
```

Remarks:

Returns True if successful.

DOSMemoryAlloc and *DOSMemoryFree* functions use Int 21h.

See also: **DosMemoryAlloc**

FarGetByte function

Returns the byte value from a specified offset of a specified protected mode segment (selector).

Declaration:

```
function FarGetByte(Seg: Word; Offs: DWord): Byte;
```

Returns the word value from a specified offset of a specified protected mode segment (selector).

See also: **FarGetDWord, FarGetWord, FarPutByte, FarPutDWord, FarPutWord**

FarGetDWord function

Returns the dword value from a specified offset of a specified protected mode segment (selector).

Declaration:

```
function FarGetDWord(Seg: Word; Offs: DWord): DWord;
```

See also: **FarGetByte, FarGetWord, FarPutByte, FarPutDWord, FarPutWord**

FarGetWord function

Returns the word value from a specified offset of a specified protected mode segment (selector).

Declaration:

```
function FarGetWord(Seg: Word; Offs: DWord): Word;
```

See also: **FarGetByte, FarGetDWord, FarPutByte, FarPutDWord, FarPutWord**

FarPutByte procedure

Assigns the byte value to a specified offset of a specified protected mode segment (selector).

Declaration:

```
procedure FarPutByte(Seg: Word; Offs: DWord; Value: Byte);
```

See also: **FarGetByte, FarGetDWord, FarGetWord, FarPutDWord, FarPutWord**

FarPutDWord procedure

Assigns the dword value to a specified offset of a specified protected mode segment (selector).

Declaration:

```
procedure FarPutDWord(Seg: Word; Offs: DWord; Value: DWord);
```

See also: **FarGetByte, FarGetDWord, FarGetWord, FarPutByte, FarPutWord**

FarPutWord procedure

Assigns the word value to a specified offset of a specified protected mode segment (selector).

Declaration:

```
procedure FarPutWord(Seg: Word; Offs: DWord; Value: Word);
```

See also: **FarGetByte, FarGetDWord, FarGetWord, FarPutByte, FarPutDWord**

FreeDescriptor function

This function is used to free descriptors that were allocated through the **AllocateDescriptors** function.

Declaration:

```
function FreeDescriptor(Selector: Word): Boolean;
```

Remarks:

Returns True if successful.

See also: **AllocateDescriptors, AllocateSpecificDescriptor, CreateCodeDescriptor, CreateDataDescriptor**

FreeDosMemoryBlock function

This function frees memory that was allocated through the **AllocDosMemoryBlock** function.

Declaration:

```
function FreeDOSMemoryBlock(Selector: Word): Boolean;
```

Remarks:

Returns True if successful.

See also: **AllocDosMemoryBlock, ResizeDosMemoryBlock**

FreePhysicalMap function

This function frees the physical mapping that was allocated through the **MapPhysicalToLinear** function.

Declaration:

```
function FreePhysicalMap(LinearAddr: DWord): Boolean;
```

Remarks:

Returns True if successful.

See also: **MapPhysicalToLinear**

FreeRealModeCallBack function

This function frees a real mode call-back address that was allocated through the allocate real mode call-back address service.

Declaration:

```
function FreeRealModeCallBack(HndSeg: Word; HndOfs: DWord):
Boolean;
```

Remarks:

Returns True if successful.

See also: **AllocRealModeCallBack**

GetCS function

Returns current code segment.

Declaration:

```
function GetCS: Word;
```

See also: **GetDS**

GetDisableInterruptState function

This function will disable the virtual interrupt flag and return the previous state of the virtual interrupt flag.

Declaration:

```
function GetDisableInterruptState: Boolean;
```

See also: **GetEnableInterruptState, GetInterruptState**

GetDPMIIntVec function

This function returns the selector and offest of the current protected mode interrupt handler for the specified interrupt number.

Declaration:

```
function GetDPMIIntVec(IntNo: Byte; var Sel: Word; var Offs:
DWord): Boolean;
```

Remarks:

Returns True if successful.

See also: **SetDPMIIntVec**

GetDPMIVer function

Returns the version of DPMI services supported.

Declaration:

```
function GetDPMIVer: Word;
```

Remarks:

Returns version of DPMI service if successful. Otherwise returns zero.

GetDS function

Returns current code segment.

Declaration:

```
function GetDS: Word;
```

See also: **GetCS**

GetEnableInterruptState function

This function will enable the virtual interrupt flag and return the previous state of the virtual interrupt flag.

Description:

```
function GetEnableInterruptState: Boolean;
```

See also: **GetDisableInterruptState, GetInterruptState**

GetExceptionHandler function

This function returns the pointer to the current protected mode exception handler for the specified exception number.

Declaration:

```
function GetExceptionHandler(ExpFault: Byte; var Sel: Word; var
Offs: DWord): Boolean;
```

Remarks:

Returns True if successful.

See also: **SetExceptionHandler**

GetFreeMemoryInfo function

This function is provided so that protected mode applications can determine how much memory is available. Under DPMI implementations that support virtual memory, it is important to consider issues such as the amount of available physical memory.

Declaration:

```
function GetFreeMemoryInfo (BufferPtr: Pointer): Boolean;
```

Remarks:

Returns True if successful.

GetInterruptState function

This function will disable the return state of the virtual interrupt flag.

Declaration:

```
function GetInterruptState: Boolean;
```

See also: **GetDisableInterruptState, GetEnableInterruptState**

GetRealModeIntVec function

This function returns the value of the current task’s real mode interrupt vector for the specified interrupt.

Declaration:

```
function GetRealModeIntVec(IntNo: Byte; var RSeg,ROfs: Word):
Boolean;
```

Remarks:

Returns segment and offset of real mode interrupt handler.

See also: **SetRealModeIntVec**

GetSegmentBaseAddress function

This function returns the 32-bit linear base address of the specified segment.

Declaration:

```
function GetSegmentBaseAddress(Selsctor: Word): DWord;
```

Remarks:

Returns 32-bit linear base address of segment if successful. Otherwise returns zero.

See also: **GetSelectorAccessRights, SetSelectorAccessRights, SetSelectorBaseAddress**

GetSelectorAccessRights function

This function returns access rights and type fields of a descriptor.

Declaration:

```
function GetSelectorAccessRights(Selector: Word): Word;
```

Remarks:

Returns access rights if successful. Otherwise returns zero.

See also: **GetSegmentBaseAddress, SetSelectorAccessRights, SetSelectorBaseAddress**

MapPhysicalToLinear function

This function can be used by device drivers to convert a physical address into a linear address. The linear address can then be used to access the device memory.

Declaration:

```
function MapPhysicalToLinear (PhysAddr, SizeInBytes: DWord):
DWord;
```

Remarks:

Returns a pointer a to linear address that can be used to access the physical memory. Otherwise returns **nil**.

See also: **FreePhysicalMap**

RealModeInt function

This function simulates an interrupt in real mode. It will invoke the CS:IP specified by the real mode interrupt vector and the handler must return by executing an **iret**.

Declaration:

```
function RealModeInt(IntNo: Byte; var Regs: TRmRegs): Boolean;
```

Remarks:

Returns True if successful.

See also: **TRmRegs type, ClearRmRegs, GetRealModeIntVec, SetRealModeIntVec**

ResizeDosMemoryBlock function

This function is used to grow or shrink a memory block that was allocated through the **AllocDosMemoryBlock** function.

Declaration:

```
function ResizeDOSmemoryBlock(Selector: Word; NewSize: DWord):
Boolean;
```

Remarks:

Returns True if successful.

See also: **AllocDosMemoryBlock, FreeDosMemoryBlock**

SegmentToDescriptor function

This function is used to convert real mode segments into descriptors that are addressable by protected mode programs.

Declaration:

```
function SegmentToDescriptor(Segment: Word): Word;
```

Remarks:

Returns selector mapped to real mode segment if successful, or zero.

SelectorInc function

Some functions such as allocate LDT descriptors and allocate DOS memory can return more than one descriptor. You must call this function to determine the value that must be added to a selector to access the next descriptor in an array.

Declaration:

function SelectorInc: Word;

Remarks:

If successful, returns the value to add to get to the next selector. Otherwise returns zero.

SetDPMIIntVec function

This function sets the selector and offest of the protected mode interrupt handler for the specified interrupt number.

Declaration:

```
function SetDPMIIntVec(IntNo: Byte; Sel: Word; Offs: DWord):
Boolean;
```

Remarks:

Returns True if successful.

See also: **GetDPMIIntVec**

SetExceptionHandler function

This function allows protected mode applications to intercept processor exceptions that are not handled by the DPMI environment. Programs may wish to handle exceptions such as not present segment faults which would otherwise generate a fatal error.

Declaration:

```
function SetExceptionHandler(ExpFault: Byte; Sel: Word; Offs:
DWord): Boolean;
```

Remarks:

Returns True is successful.

See also: **GetExceptionHandler**

SetRealModeIntVec function

This function sets the value of the current task's real mode interrupt vector for the specified interrupt.

Declaration:

```
function SetRealModeIntVec(IntNo: Byte; RSeg, ROfs: Word):
Boolean;
```

Remarks:

Returns True if successful.

See also: **GetRealModeIntVec, RealModeInt**

SetSelectorAccessRights function

This function allows a protected mode program to modify the access rights and type fields of a descriptor.

Declaration:

```
function SetSelectorAccessRights(Selector,Rights: Word):
Boolean;
```

Remarks:

Returns True if successful.

See also: **GetSegmentBaseAddress, GetSelectorAccessRights, SetSelectorBaseAddress, SetSelectorLimit**

SetSelectorBaseAddress function

This function changes the 32-bit linear base address of the specified selector.

Declaration:

```
function SetSelectorBaseAddress(Selector: Word; Base: DWord):
Boolean;
```

Remarks:

Returns True if successful.

See also: **SetSelectorAccessRights, GetSegmentBaseAddress, GetSelectorAccessRights, SetSelectorLimit**

SetSelectorLimit function

This function sets the limit for the specified segment.

Declaration:

```
function SetSelectorLimit(Selector: Word; Limit: DWord): Word;
```

Remarks:

Returns selector if successful. Otherwise returns zero.

See also: **GetSegmentBaseAddress, GetSelectorAccessRights, SetSelectorAccessRights, SetSelectorBaseAddress**

Chapter 6

The ErrCodes Unit

Targets: MS-DOS, OS/2, Win32

Contains constants for error codes, given by *RunError*, and by the *Error_msg* function that deciphers the error code.

Declaration:

```
Function Error_msg (Err: Word): string;
```

You should use *Error_msg* the function in your own error handling procedures.

See also: **Run-time Error Codes**

Chapter 7

The Graph Unit

Targets: MS-DOS, Win32

7.1 Graph Unit Introduction

The Graph unit for TMT Pascal is as compatible with the Borland Graphics Library as possible. Since BGI is by now an obsolete interface, we added a number of enhancements. They are described in detail below. This graphics library for TMT Pascal allows easy porting of programs written for Borland Pascal (with minimal changes to the source).

Features

- OS independent interface. You need not make any changes in the sources to compile them as MS-DOS or Win32 applications.
- Real 32-bit accelerated graphics;
- Mostly compatible with Borland's GRAPH;
- Supports the following graphic modes:
  - 256-colored VGA/MCGA (13h BIOS) mode;
  - all SVGA 256 color (PaletteColor) modes;
  - all SVGA 32k/64k color (HiColor) modes;
  - all SVGA 16M/16M+A color (TrueColor) modes;
- VESA VBE 1.2/2.0 features (32-bits PM interface, etc);
- Microsoft DirectDraw 5.0 features in Win32 applications.
- Banked and LFB (Linear Flat frame Buffer) SVGA modes;
- Logical pages and hardware scrolling (MS-DOS applications only);
- No 64K limit on sprite size;
- Does not use BGI drivers;
- Uses a flat memory model for greater performance.
- Advanced sprite engine with transparent BLT;
- A virtual graphics mode for DOUBLE and TRIPLE buffering is available.

System Requirements

- VGA compatible video card required (VESA VBE 1.2 recommended, VESA VBE 2.0 is best) for MS-DOS and Microsoft DirectX 5.0 or higher for Windows'95/98/2000 or Windows NT applications.
- CPU Intel 80386 or higher compatible;
- PMODE, PMODEW or WDOSX compatible DOS extender for MS-DOS 32-bit protected mode applications.

Notices on Win32 target

The Graph unit can be used to emulate MS-DOS SVGA graphics in a Windows 32-bit GUI and console applications using the Microsoft DirectDraw 5.0 or later. Moreover the Graph unit for Win32 works in conjunction with the CRT, Keyboard and Mouse units in a same way, as in MS-DOS applications. So you may recompile most of your old MS-DOS programs as Win32 GUI applications without any changes in your sources. For example, you may compile any source from your TMTPL\SAMPLES\COMMON\GRAPH subdirectory for both Win32 GUI and MS-DOS targets.

Compatibility with the Graph unit from Borland Pascal

TMT Graph unit will partly replicate the Graph unit from Borland Pascal. However there are some differences. Listed below are the names of all procedures and functions from the Graph unit from Borland Pascal 7 and an indication of their status in the TMT Graph unit.

```
+Arc
+Bar
+Bar3D
+Circle
*ClearDevice
*ClearViewPort
*CloseGraph
-DetectGraph
*Drawpoly
+Ellipse
*FillEllipse
*FillPoly
*FloodFill
-GetArcCoords
*GetAspectRatio
*GetBkColor
+GetColor
+GetDefaultPalette
-GetDriverName
+GetFillPattern
-GetFillSettings
*GetGraphMode
+GetImage
+GetLineSettings
+GetMaxColor
-GetMaxMode
+GetMaxX
+GetMaxY
-GetModeName
-GetModeRange
+GetPalette
+GetPaletteSize
+GetPixel
*GetTextSettings
+GetViewSettings
+GetX
+GetY
*GraphDefaults
+GraphErrorMsg
+GraphResult
+ImageSize
-InitGraph
-InstallUserDriver
-InstallUserFont
+Line
+LineRel
+LineTo
+MoveRel
+MoveTo
+OutText
+OutTextXY
-PieSlice
*PutImage
+PutPixel
*Rectangle
-RegisterBGIdriver
-RegisterBGIfont
*RestoreCrtMode
-Sector
+SetActivePage
+SetAllPalette
*SetAspectRatio
*SetBkColor
+SetColor
*SetFillPattern
*SetFillStyle
+SetGraphBufSize
*SetGraphMode
+SetLineStyle
+SetPalette
+SetRGBPalette
+SetTextJustify
*SetTextStyle
-SetUserCharSize
*SetViewPort
+SetVisualPage
*SetWriteMode
+TextHeight
+TextWidth
```

Definitions:
+ procedure/function supported and functionally equivalent to Borland;
* procedure/function supported, but somewhat different from Borland;
- procedure/function not supported.

Note that TMT Graph provides many procedures and functions, which are not supported by Borland’s Graph unit.

7.2 Graph Unit Types, Constants and Variables

DrawBorder variable

You can enable and disable a border drawing for *Ellipse, FillCircle* and *FillTriangle* procedures.

Declaration:

```
var DrawBorder: Boolean
```

Remarks:

If DrawBorder = True, border drawing enabled (default).

If DrawBorder = False, border drawing disabled.

Bar3D constants

These constants are used to specify whether a 3D graph bar has a top. (See **Bar3D**)

Constant	Value
TopOn	True
TopOff	False

BitBlt operators

Use these operators for images you place on the screen with *PutImage, PutSprite* and *PutHTextel.*

Constant	Value	Meaning
NormalPut	0	MOV
CopyPut	0	MOV
XORPut	1	XOR
OrPut	2	OR
AndPut	3	AND

Clipping constants

Constant	Value
ClipOn	True
ClipOff	False

Color constants

The graph unit defines the following color constants:

```
clBlack:        DWord =0;
clBlue:         DWord =1;
clGreen:        DWord =2;
clCyan:         DWord =3;
```

```
clRed:          DWord =4;
clMagenta:      DWord =5;
clBrown:        DWord =6;
clLightGray:    DWord =7;
clDarkGray:     DWord =8;
clLightBlue:    DWord =9;
clLightGreen:   DWord =10;
clLightCyan:    DWord =11;
clLightRed:     DWord =12;
clLightMagenta: DWord =13;
clYellow:       DWord =14;
clWhite:        DWord =15;
```

Remarks:
All assigned values above are listed for 256-colored modes only. In HiColor and TrueColor SVGA modes the values will automatically be toned up by the Graph unit to a concrete graphic mode; i.e. the constants actually are constant-variables, which initialize whenever the current color mode changes. You may use color variables to display the same colors in different graphic modes (256-colored, HiColor or TrueColor).

Fill pattern constants

Constant	**Value**	**Meaning**
EmptyFill	0	Uses background color
SolidFill	1	Uses draw color
LineFill	2	--- fill
LtSlashFill	3	/// fill
SlashFill	4	/// thick fill
BkSlashFill	5	\thick fill
LtBkSlashFill	6	\fill
HatchFill	7	Light hatch fill
XHatchFill	8	Heavy cross hatch
InterleaveFill	9	Interleaving line
WideDotFill	10	Widely spaced dot
CloseDotFill	11	Closely spaced dot
UserFill	12	User-defined fill

FillSettingsType

Declaration:

```
FillSettingsType = record
  Pattern: DWord;
  Color  : DWord;
end;
```

GraphModeType

Declaration:

```
GraphModeType = record
 VideoMode    : Word;
 HaveLFB      : Boolean;
 BitsPerPixel : Byte;
 XResolution  : Word;
 YResolution  : Word;
end;
```

GraphWndProc

Targets: Win32 only

A function variable that processes the system messages sent to the graphical window.

Declaration:

```
var GraphWndProc: ^function(Window: HWND; Mess, WParam, LParam:
LongInt): LongInt := nil;
```

Remarks:

Assign your own callback function to the *GraphWndProc* variable to process any message sent to the graphical window.

Example:

```
uses CRT, Windows, Messages, Graph;
{ Draws a message box }
procedure ShowBox;
begin
  SetFillColor(Random(256));
  Bar3D(200, 180, 440, 280, 0, FALSE);
  SetTextJustify(CenterText, CenterText);
  SetColor(clBlack);
  OutTextXY(321, 231, 'Click Here');
  SetColor(clWhite);
  OutTextXY(320, 230, 'Click Here');
end;
{ A custom message handler }
function MyWndProc(Window: HWND; Mess, WParam, LParam:
LongInt): LongInt;
begin
  if (Mess = WM_LBUTTONDOWN) then
    if (LOWORD(lParam) > 200) and (LOWORD(lParam) < 440) and
       (HIWORD(lParam) > 180) and (HIWORD(lParam) < 280)
    then
      ShowBox
    else
      Beep(0, 0);
end;
{ Main program }
begin
  GraphWndProc := @MyWndProc;//Install a custom message handler
  SetSVGAMode(640, 480, 8, 0);  // Set desired video mode
  ShowCursor(TRUE);             // A custom message handler
  ShowBox;                      // Show a message box
  ReadKey;                      // Wait for any key
end.
```

Graphic result constants

Constant	Value
grOK	0
grInvalidMode	1
grModeNotSupported	2
grSetModeError	3
grLFBSetupError	4
grError	5
grVESANotFound	6
grVESAError	7
grNoGraphMem	8;
grInvalidDriver	9;
grDirectXNotFound	10;
grDirectXError	11;

IgnoreBreak variable

Targets: Win32 only

You can enable and disable responding to Ctrl+Break and Ctrl+C .

Declaration:

```
var IgnoreBreak: Boolean
```

Remarks:

If IgnoreBreak = TRUE, Ctrl+Break and Ctrl+C will be ignored (default).

If IgnoreBreak = FALSE, Ctrl+Break and Ctrl+C will terminate the application.

IgnoreCloseMessage variable

Targets: Win32 only

Enables or disables responding to a WM_CLOSE message.

Declaration:

```
var IgnoreCaseMessage: Boolean
```

Remarks:

If IgnoreCaseMessage = TRUE, WM_CLOSE message will be ignored (default).

If IgnoreCaseMessage = FALSE, WM_CLOSE message will be processed in the usual way and the application can be closed by pressing Alt+F4.

Justification constants

Use these constants to specify horizontal and vertical justification for *SetTextJustify*.

Horizontal Constant	Value
LeftText	0
CenterText	1
RightText	2

Vertical Constant	Value
BottomText	0
CenterText	1
TopText	2

Note how each justification constant places the output text relative to the output coordinates:

```
TopText     TopText  TopText   TopText
LeftText      CenterText       RightText
BottomText    BottomText       BottomText
```

LineSettingsType

The record that defines the style, pattern, and thickness of a line.

Declaration:

```
LineSettingsType = record
   LineStyle : Word;
   Pattern   : Word;
   Thickness : Word;
end;
```

PaletteType

The record that defines the size and colors of the palette; used by *GetPalette*, *GetDefaultPalette*, and *SetPalette*.

Declaration:

```
PaletteType = record
  Size    : Byte;
  Colors  : array[0..MaxColors] of DWord;
end;
```

The size field reports the number of colors in the palette for the current driver in the current mode. *Colors* contains the actual colors 0..*Size* - 1.

PointType

A type defined for your convenience. Both fields are of type Longint rather than Integer.

Declaration:

```
PointType = record
  X, Y : Longint;
end;
```

RGBType

A type used for access to Red, Green and Blue fields of each palette entry.

Declaration:

```
RGBType = record
  Blue, Green, Red, Aligment: Byte;
end;
```

SVGA mode constants

SVGA mode constants used with **SetSVGAMode** procedure.

Constant	Value	Meaning
LFBorBanked	0	LFB mode (if supported) or banked mode
BankedOnly	1	banked mode only
LFBOnly	2	LFB mode only

TextSettingsType

The record that defines the text attributes used by *GetTextSettings*

Declaration:

```
TextSettingsType = record
  Font           : Pointer;
  FontSize       : DWord;
  FirstChar      : DWord;
  Width          : DWord;
  Height         : DWord;
  Space          : DWord;
  Direction      : DWord;
  Horiz          : DWord;
  Vert           : DWord;
end;
```

Text-Style constants

These constants are used with *SetTextStyle* and *GetTextSettings*.

Constant	Value	Meaning
SmallFont	0	8x8 bit mapped font
MediumFont	1	8x14 bit mapped font
LargeFont	2	8x16 bit mapped font

ViewPortType

A record that reports the status of the current viewport; used by *GetViewSettings*

Declaration:

```
ViewPortType      = record
   x1, y1, x2, y2 : Longint;
   Clip           : Boolean;
end;
```

The points (X1, Y1) and (X2, Y2) are the dimensions of the active viewport and are given in absolute screen coordinates. Clip is a Boolean variable that controls whether clipping is active.

VbeInfoType

Targets: MS-DOS only

A record that stores the VESA VBE information block; used by *GetVbeInfo*.

Declaration:

```
VbeInfoType          = record
  VbeSignature       : DWord;
  VbeVersion         : Word;
  OemStringPtr       : DWord;
  Capabilities       : DWord;
  VideoModePtr       : DWord;
  TotalMemory        : Word;
  OEMSoftwareRev     : Word;
  OEMVendorNamePtr   : DWord;
  OEMProductNamePtr  : DWord;
  OEMProductRevPtr   : DWord;
  Reserved           : array [0..221] of Byte;
  OEMData            : array [0..255] of Byte;
end;
```

VbeModeInfoType

Targets: MS-DOS only

A record that stores VESA VBE mode information block; used by *GetVbeModeInfo*.

Declaration:

```
VbeModeInfoType      = record
  ModeAttributes     : Word;
  WinAAttributes     : Byte;
  WinBAttributes     : Byte;
  WinGranularity     : Word;
  WinSize            : Word;
  WinASegment        : Word;
  WinBSegment        : Word;
  WinFuncPtr         : Pointer;
  BytesPerScanLine   : Word;
  XResolution        : Word;
  YResolution        : Word;
  XCharSize          : Byte;
  YCharSize          : Byte;
  NumberOfPlanes     : Byte;
  BitsPerPixel       : Byte;
  NumberOfBanks      : Byte;
  MemoryModel        : Byte;
  BankSize           : Byte;
  NumberOfImagePages : Byte;
  Reserved           : Byte;
  RedMaskSize        : Byte;
  RedFieldPosition   : Byte;
  GreenMaskSize      : Byte;
  GreenFieldPosition : Byte;
  BlueMaskSize       : Byte;
  BlueFieldPosition  : Byte;
  RsvdMaskSize       : Byte;
  RsvdFieldPosition  : Byte;
  DirectColorModeInfo: Byte;
  PhysBasePtr        : DWord;
  OffScreenMemOffset : DWord;
  OffScreenMemSize   : Word;
  Reserved2          : array [0..205] of Byte;
end;
```

7.3 Graph Unit Procedures and Functions

AnalizeRGBColor procedure

Returns RGB fields of a given color.

Declaration:

```
procedure AnalizeRGB(Color: DWord; var R,G,B: Byte);
```

Remarks:

This function works in HiColor and TrueColor SVGA modes only. RGB field values, returned by *AnalizeRGB*, depend on the current video mode (32K, 64K or 16M colors).

Arc procedure

Draws a circular arc.

Declaration:

```
procedure Arc(X, Y; Integer; StAngle, EndAngle, Radius; Word);
```

Remarks:

The arc begins at *StAngle* (start angle) and ends at *EndAngle*, with radius *Radius*, and with (*X,Y*) as the center point.

Bar procedure

Draws a bar with the current fill color.

Declaration:

procedure Bar(X1, Y1, X2, Y2: Longint);

Remarks:

Bar draws a filled-in rectangle (used in bar charts, for example). It uses style and color defined by *SetFillColor*, *SetFillPattern* or *SetFillStyle*. To draw an outlined bar, call *Bar3D* with a zero depth.

Bar3D procedure

Draws a 3-D bar using the current fill color.

Declaration:

```
procedure Bar3D(X1, Y1, X2, Y2: DWord; Depth: Word; Top:
Boolean);
```

Remarks:

Bar3D draws a filled-in, three-dimensional bar with the pattern and color defined by *SetFillColor*, *SetFillPattern* or *SetFillStyle*. The 3-D outline of the bar is drawn in the current line style and color as set by *SetLineStyle* and *SetColor*. *Depth* is the length in pixels of the 3-D outline. If Top is TopOn, a 3-D top is put on the bar; if *Top* is TopOff, no top is put on the bar (making it possible to stack several bars on top of one another).

A typical depth could be calculated by taking 25% of the width of the bar:

```
Bar3D(X1, Y1, X2, Y2, (X2 - X1 + 1) div 4, TopOn);
```

Circle procedure

Draws a circle in the current color set by *SetColor*, using (*X*,*Y*) as the center point.

Declaration:

```
procedure Circle(X,Y: Longint; Radius: DWord);
procedure Circle(X,Y: Longint; Radius, Color: DWord);
```

Remarks:

Draws a circle in the current color set by *SetColor*. Each graphics mode has an aspect ratio used by *Circle*.

ClearDevice procedure

Clears the currently selected output device and homes the current pointer.

Declaration:

procedure ClearDevice;

Remarks:

ClearDevice moves the current pointer to (0, 0), and clears all accessible video memory with zero value.

ClearPage procedure

Clears the current active page using the background color set by *SetBkColor* and moves the current pointer to (0, 0).

Declaration:

```
procedure ClearPage;
```

Remarks:

Active logical page sets by *SetLogicalPage* may be larger than the physical screen.

ClearViewPort procedure

Clears the current view port using the background color set by *SetBkColor* and moves the current pointer to (0, 0).

Declaration:

```
procedure ClearViewPort;
```

Remarks:

ClearViewPort sets the fill color to the background color and moves the current pointer to (0, 0).

CliRetrace procedure

Switches off interrupts, waits for vertical retrace and restore interrupts.

Declaration:

```
procedure CliRetrace;
```

CliHRetrace procedure

Switches off interrupts, waits for horizontal retrace and restore interrupts.

Declaration:

```
procedure CliHRetrace;
```

CloseGraph procedure

Shuts down the graphics system.

Declaration:

procedure CloseGraph;

Remarks:

CloseGraph restores the original screen mode before graphics was initialized the first time and frees the memory allocated for the graphics buffer.

DetectSVGAMode procedure

Targets: MS-DOS only

Returns a valid VESA VBE mode number for the requested video mode. If the requested video mode is not supported, returns a zero value.

Declaration:

```
function DetectSVGAMode(XRes, YRes, BPP, VMode: Word): Word;
```

Remarks:

This function may be used with the **SetGraphMode** procedure.

The following example tries to set the SVGA mode 640x480 with maximum color depth:

```
uses Crt, Graph;
function SetSVGA640x480: String;
const Bps: array [0..4] of Word = (32, 24, 16, 15, 8);
var   Mode,i: Word;
begin
  for I := 0 to 4 do begin
    Mode:=DetectSVGAMode(640, 480, bps[i], LfbOrBanked);
    if Mode > 0 then begin
      SetGraphMode(Mode);
      if GraphResult = grOk then begin
        Str(bps[i], Result);
        exit;
      end;
    end;
  end;
  RestoreCrtMode;
  Result := '';
end;
// main program
var S: String;
begin
 S := SetSVGA640x480;
 if S <> '' then begin
  SetTextJustify(CenterText, CenterText);
  OutTextXY(320, 240, 'This is SVGA mode 640x480 ' + S + '
bps');
  OutTextXY(320, 260, 'Press any key...');
  ReadKey;
  RestoreCrtMode;
 end else
  Writeln('SVGA mode 640x480 not supported.');
end.
```

DrawEllipse procedure

Draws an ellipse

Declaration:

```
procedure DrawEllipse(X, Y, A, B: Longint)
```

Remarks:

(*X*,*Y*) is the center point; *A* and *B* are the horizontal and vertical axes.

DrawHLine procedure

Draws a horizontal line using the current fill color.

Declaration:

```
procedure DrawHLine(X1,X2,Y: Longint);
```

DrawPoly procedure

Draws the outline of a polygon using the current line style and color.

Declaration:

```
procedure DrawPoly(NumVert: DWord; var Vert);
```

Remarks:

Numvert specifies the number of coordinate pairs in *Vert*. A coordinate pair consists of two Longint values, an X and a Y value.

Ellipse procedure

Draws a portion of an ellipse.

Declaration:

```
procedure Ellipse(X, Y: Longint; StAngle, EndAngle, XRadius,
YRadius: DWord);
```

Remarks:

Draws an arc from *StAngle* (start angle) to *EndAngle*, with radii *Xradius* and *YRadius*, and (*X*,*Y*) as the center point.

ExpandFill procedure

Draws a portion of an ellipse.

Fills a bounded region with the current color.

Declaration:

```
procedure ExpandFill(X, Y: Integer);
```

Remarks:

Fills an enclosed area on bitmap devices. (*X*, *Y*) is a seed within the enclosed area to be filled. The current fill color, as set by *SetFillColor*, is used to flood the area bounded by any different color. If the seed point is within an enclosed area, then the inside will be filled. If the seed is outside the enclosed area, then the outside will be filled.

FillCircle procedure

Draws a filled circle in the current color set by *SetColor*, using (*X*,*Y*) as the center point.

Declaration:

```
procedure FillCircle(X,Y: Longint; Radius: DWord);
```

Remarks:

Draws a circle in the current color set by *SetColor* and fill it using the current fill style and color defined by *SetFillColor, SetFillPattern* or *SetFillStyle*. Each graphics mode has an aspect ratio used by *Circle*.

FillEllipse procedure

Draws a filled ellipse

Declaration:

```
procedure FillEllipse(X, Y, A, B: Longint)
```

Remarks:

Draws an ellipse in the current color set by *SetColor* and fills it using the current fill style and color defined by *SetFillColor, SetFillPattern* or *SetFillStyle*. (*X*,*Y*) is the center point; *A* and *B* are the horizontal and vertical axes.

FillPoly procedure

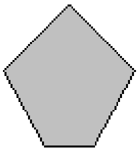
Fills a polygon, using the scan converter.

Declaration:

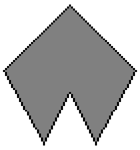
```
procedure FillPoly(NumVert: DWord; var Vert);
```

Remarks:

Vert is an untyped parameter that contains the coordinates of each vertex in the polygon. *NumVert* specifies the number of coordinate pairs in *Vert*. A coordinate pair consists of two Longint values, an *X* and a *Y* value. *FillPoly* calculates all the horizontal intersections, and then fills it using the current fill style and color defined by *SetFillColor, SetFillPattern* or *SetFillStyle*. This function is different from Borland's GRAPH unit. Polygons must have angles less than 180 degrees. That is, they must be convex.

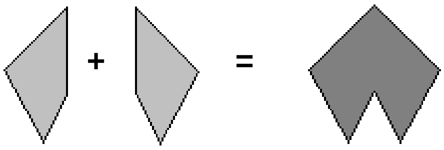


1) Valid polygon



2) Invalid polygon

If you want to draw polygon (2), you must split it up into two valid polygons:



FillTriangle procedure

Draws a filled triangle.

Declaration:

```
procedure FillTriangle(X1, Y1, X2, Y2, X3, Y3: Longint);
```

Remarks:

Draws a triangle in the current color set by *SetColor* and fill it using the current fill style and color defined by *SetFillColor, SetFillPattern* or *SetFillStyle*.

FlipImageOX procedure

Flips an image (BitMap) left to right.

Declaration:

```
procedure FlipImageOX(var BitMap);
```

Remarks:

See **FlipImageOY** for example.

FlipImageOY procedure

Flips an image (BitMap) top to bottom.

Declaration:

```
procedure FlipImageOY(var BitMap);
```

Example:

```
uses Graph, CRT;
var P: Pointer;
    i, j, dx, dy: DWord;
begin
 (* Set SVGA mode 640x480x256. You can set ANY supported mode
*)
 SetSVGAMode(640, 480, 8, LfbOrBanked);
 if GraphResult <> grOk then begin
  ClrScr;
  Writeln(GraphErrorMsg(GraphResult));
  exit;
 end;
 dx := (GetMaxX + 1) div 2;
 dy := (GetMaxY + 1) div 2;
 for i := 0 to dx do
  for j := 0 to dy do
   PutPixel(i, j, i * j div dx);
 SetTextJustify(CenterText, BottomText);
 OutTextXY(dx, dy + dy div 2, 'Press any key...');
 ReadKey;
 GetMem(P, ImageSize(0, 0, dx - 1, dy - 1));
 GetImage(0, 0, dx - 1 , dy - 1, P^);
 FlipImageOX(P^);
 PutImage(dx, 0, P^);
 FlipImageOY(P^);
 PutImage(dx, dy, P^);
 FlipImageOX(P^);
 PutImage(0, dy, P^);
 ReadKey;
 FreeMem(P, ImageSize(0, 0, dx - 1, dy - 1));
 RestoreCrtMode;
end.
```

FlipToMemory procedure

Copies contents of graphic page number 0 to the memory buffer pointed to by *Addr*.

Declaration:

```
procedure FlipToMemory(Addr: Pointer)
```

Remarks:

Use this procedure to flip the contents of the graphic page into the virtual page.

See also: **FlipToScreen**, **SetVirtualMode**, **SetNormalMode**

FlipToScreen procedure

Copies contents of memory buffer pointed by *Addr* to graphic page number 0.

Declaration:

```
procedure FlipToScreen(Addr: Pointer)
```

Remarks:

Use this procedure to flip the contents of the virtual page into the graphic page.

See also: **FlipToMemory, SetVirtualMode, SetNormalMode**

FloodFill procedure

Fills a bounded region with the current color.

Declaration:

```
procedure FloodFill(X, Y: Integer; Border: Word);
```

Remarks:

Fills an enclosed area on bitmap devices. (*X*, *Y*) is a seed within the enclosed area to be filled. The current fill color, as set by *SetFillColor*, is used to flood the area bounded by *Border* color. If the seed point is within an enclosed area, then the inside will be filled. If the seed is outside the enclosed area, then the exterior will be filled.

GetActivePage function

Returns the current active page number.

Declaration:

function GetActivePage: DWord;

GetAspectRatio procedure

Returns the current aspect ratio factor.

Declaration:

```
procedure GetAspectRatio(var AspectRatio: Real);
```

Remarks:
See the **SetAspectRatio** more info.

GetBytesPerScanLine function

Returns the scan line size in bytes.

Declaration:

```
function GetBytesPerScanLine: DWord;
```

GetColor function

Returns the current drawing color.

Declaration:

```
function GetColor: DWord;
```

GetDefaultPalette procedure

Returns the palette definition structure.

Declaration:

```
procedure GetDefaultPalette(var Palette: PaletteType);
```

This structure contains the palette which the new graphic mode initialized.

Remarks:
GetDefaultPalette returns a record of *PaletteType*, which contains the palette.

GetFillColor function

Returns the current fill color as set by **SetFillColor**.

Declaration:

```
function GetColor: DWord;
```

GetFillPattern procedure

Returns the currently selected fill pattern and color as set by **SetFillStyle** or **SetFillPattern**.

Declaration:

```
procedure GetFillPattern(var FillPattern: FillPatternType);
```

Remarks:

If no user call has been made to *SetFillPattern*, *GetFillPattern* returns an array filled with $FF.

GetFillSettings procedure

Gets the current fill pattern and color, as set by *SetFillStyle*, *SetFillPattern* or *SetFillColor*.

Declaration:

```
procedure GetFillSettings(var FillInfo: FillSettingsType);
```

Remarks:

The Pattern field reports the current fill pattern selected. The colors field reports the current fill color selected. Both the fill pattern and color can be changed by calling the *SetFillStyle*, *SetFillPattern* or *SetFillColor* procedure.

If Pattern is equal to *UserFill*, use *GetFillPattern* to get the user-defined fill pattern that is selected.

See also: **FillSettingsType**

GetGraphBufSize function

Returns the size of internal graphic buffer;

Declaration:

```
function GetGraphBufSize: DWord;
```

Remarks:

See the **SetGraphBufSize** procedure.

GetGraphMode function

Targets: MS-DOS only

Returns VESA-compatible mode number of the current graphic mode.

Declaration:

```
function GetGraphMode: Word;
```

Remarks:

The mode number returned by *GetGrapMode* may be used with *SetGraphMode*.

GetHTextel procedure

Gets a horizontal set of pixels (horizontal textel) from the screen and puts it into memory.

Declaration:

```
procedure GetHtextel(X1, X2, Y: Longint; var Textel);
```

GetImage procedure

Saves a bit image of the specified region into a buffer.

Declaration:

```
procedure GetImage(X1, Y1, X2, Y2: Integer; var BitMap);
```

Remarks:

X1, *Y1*, *X2*, and *Y2* are the coordinates of diagonally opposite points of the rectangular region on the screen. *BitMap* is an untyped parameter that must be at least 4 greater than the amount of area defined by the region. The first two words of *BitMap* store the width and height of the region.

The remaining part of *BitMap* is used to save the bit image itself. Use the ImageSize function to determine the size requirements of *BitMap*.

GetLfbAddress function

Targets: MS-DOS only

Returns the physical address of the linear flat frame buffer.

Declaration:

```
function GetLfbAddress: DWord;
```

Remarks:

If LFB is not supported, the function returns zero.

GetLineSettings procedure

Returns the current line style, line pattern, and line thickness, as set by **SetLineStyle**.

Declaration:

```
procedure GetLineSettings(var LineInfo: LineSettingsType);
```

GetLogicalPage procedure

Returns the current logical page size.

Declaration:

```
procedure GetLogicalPage(var SX, SY: Word);
```

Remarks:

See the *SetLogicalPage* procedure for more info.

GetMaxColor function

Returns the highest color that can be passed to the **SetColor** procedure.

Declaration:

```
function GetMaxColor: DWord;
```

Remarks:

For example, in 256 colored VGA/SVGA mode, *GetMaxColor* always returns 255, which means that any call to *SetColor* with a value from 0..255 is valid. On an SVGA in high-color mode, *GetMaxColor* returns a value of 32767, 65535, etc.

GetMaxPage function

Returns the number of the last accessible graphic page.

Declaration:

```
function GetMaxPage: DWord;
```

Win32 target:

Only two graphic pages are available for the Graph unit.

GetMaxX function

Gets the current X resolution.

Declaration:

```
function GetMaxX: DWord;
```

Remarks:

Returns the rightmost column (X resolution) of the logical visual page in the current graphics mode.

GetMaxY function

Gets current Y resolution.

Declaration:

```
function GetMaxX: DWord;
```

Remarks:

Returns the bottommost row (Y resolution) of the logical visual page in the current graphics mode.

GetOemProductName function

Targets: MS-DOS only

Returns the string containing the name of the display controller board.

Declaration:

```
function GetOemProductName: String;
```

Remarks:
If VESA VBE 2.0+ is not supported, this function will return an empty string.

GetOemProductRev function

Targets: MS-DOS only

Returns the string revision or manufacturing level of the display controller board.

Declaration:

```
function GetOemProductRev: String;
```

Remarks:
If VESA VBE 2.0+ is not supported, this function will return an empty string.

GetOemString function

Targets: MS-DOS only

Returns the OEM-defined string.

Declaration:

```
function GetOemString: String;
```

Remarks:
This string may be used to identify the graphics controller chip or OEM product family for hardware specific display drivers. If VESA VBE 1.2+ is not supported, this function will return an empty string.

GetOemVendorName function

Targets: MS-DOS only

Returns the string containing the name of the vendor which produced the display controller board.

Declaration:

```
function GetOemVendorName: String;
```

Remarks:
If VESA VBE 2.0+ is not supported, this function will return an empty string.

GetPageDC function

Targets: Win32 only

Retrieves a handle of a display device context (DC) for the active graphics page.

Declaration:

```
function GetPageDC(PageNo: DWORD): HDC;
```

Remarks:

After painting with a common device context, the *ReleasePageDC* procedure must be called to release the device context.

GetPageSize function

Returns size (in bytes) of the logical video page in the current graphic mode.

Declaration:

```
function GetPageSize: DWord;
```

Remarks:

The size of the graphic page depends on the graphic mode and on the size of logical pages, installed by the *SetLogicalPage*.

GetPalette procedure

Returns the current palette and its size.

Declaration:

```
procedure GetPalette(var Palette: PaletteType);
```

Remarks:

Returns the current palette and its size in a variable of *PaletteType*.

GetPixel function

Gets the pixel value (color) at (*X*,*Y*).

Declaration:

```
function GetPixel(X, Y: Longint): DWord;
```

GetRGBPalette procedure

Returns the palette entries for the VGA, MCGA and 256-colored SVGA modes.

Declaration:

```
procedure GetRGBPalette(ColorNum: Byte; var RedValue,
GreenValue, BlueValue: Byte);
```

Remarks:

ColorNum defines the palette entry to be returned. *RedValue*, *GreenValue*, and *BlueValue* return the component colors of the palette entry.

GetScreenHeight function

Returns the height in pixels of the physical screen.

Declaration:

```
function GetScreenHeight: DWord;
```

GetScreenWidth function

Returns the width in pixels of the physical screen.

Declaration:

```
function GetScreenWidth: DWord;
```

GetTextSettings procedure

Gets settings for text output in graphics mode.

Declaration:

```
procedure GetTextSettings(var TextInfo: TextSettingsType);
```

Remarks:

Returns the current text font, direction, size, and justification as set by *SetTextStyle* or *SetCustomFont* and *SetTextJustify*.

GetTranspSettings procedure

Returns the current transparent mode settings.

Declaration:

```
procedure GetTranspSettings(var Mode: Boolean; var Color:
DWord);
```

Remarks:

See the **SetTranspMode** for more info.

GetVbeCapabilities function

Targets: MS-DOS only

Returns VESA VBE capabilities field.

Declaration:

```
function GetVbeCapabilities: DWord;
```

GetVbeInfo procedure

Targets: MS-DOS only

Returns VESA VBE info.

Declaration:

```
procedure GetVbeInfo(var VbeInfo: VbeInfoType);
```

Remarks:

VbeInfo must be of *VbeInfoType*. You don't need to allocate a real mode memory block for the VBE Information Table. TMT Graph translates it from real mode memory into vi so you can directly access it from normal Pascal code.

GetVbeModeInfo procedure

Targets: MS-DOS only

Returns the video mode information for the specified VBE internal video mode number.

Declaration:

```
procedure GetVbeModeInfo(ModeNo: Word; var VbeModeInfo:
VbeModeInfoType);
```

Remarks:

VbeModeInfo must be of *VbeModeInfoType*. You don't need to allocate real mode memory block for VBE Mode Information Table. TMT Graph translates it from real mode memory into *VbeModeInfo* so you can directly access it from normal Pascal code.

GetVbeModesList procedure

Returns list of supported VESA VBE modes.

Declaration:

```
procedure GetVbeModesList(var ModesList: array of
GraphModeType);
```

Remarks:

Check *ModesList* structure to get information about all supported modes.

See also: **GraphModeType.**

GetVbeVersion function

Targets: MS-DOS only

Returns the version of VESA BIOS Implementation.

Declaration:

```
function GetVbeVersion: Word
```

Remarks:

The Vbe version is a BCD value which specifies what level of the VBE standard is implemented in the software. The higher byte specifies the major version number. The lower byte specifies the minor version number. Note: The BCD value for VBE 2.0 is 0200h and the BCD value for VBE 1.2 is 0102h. In the past we have had some applications misinterpreting these BCD values. For example, BCD 0102h was interpreted as 1.02, which is incorrect.

GetViewSettings procedure

Gets the current viewport and clipping parameters.

Declaration:

```
procedure GetViewSettings(var ViewPort: ViewPortType);
```

Remarks:

GetViewSettings returns a variable of *ViewPortType*.

GetVisualPage function

Gets the current visual page number.

Declaration:

```
function GetVisualPage: DWord;
```

GetWindowHandle function

Targets: Win32 only

Returns the handle of the window where the Graph unit displays output and receives input from the user.

Declaration:

```
function GetWindowHandle: Thandle;
```

GetWriteMode function

Returns the current write mode.

Declaration:

```
function GetWriteMode: DWord;
```

GetX function

Returns the X coordinate of the current pointer (CP).

Declaration:

```
function GetX: Longint;
```

GetY function

Returns the Y coordinate of the current pointer (CP).

Declaration:

```
function GetY: Longint;
```

GraphResult function

Returns an error code for the last graphics operation.

Declaration:

```
function GraphResult: Integer;
```

See also: **Graphic result constants**

GraphDefaults procedure

Homes the current pointer (CP) and resets the graphics system to specified default values.

Declaration:

```
procedure GraphDefaults;
```

Remarks:

Homes the current pointer (CP) and resets the graphics system to the default values for:

viewport
palette
draw and background colors
line style and line pattern
fill color
active font, text style, text justification, and user Char size

GraphErrorMsg function

Returns an error message string for the specified ErrorCode.

Declaration:

```
function GraphErrorMsg(ErrorCode: Integer): string;
```

See also: **Graph Error Constants**.

HRetrace procedure

Waits for horizontal retrace.

Declaration:

```
procedure HRetrace;
```

ImageSize function

Returns the number of bytes required to store a rectangular region of the screen.

Declaration:

```
function ImageSize(X1, Y1, X2, Y2: Longint): DWord;
```

Remarks:

X1, *Y1*, *X2*, and *Y2* are the coordinates of diagonally opposite vertices of a rectangular region on the screen. *ImageSize* determines the number of bytes necessary for *GetImage* to save the specified region of the screen. The image size includes space for two words. The first stores the width of the region and the second stores the height.

InvertImage procedure

Inverts an image.

Declaration:

```
procedure InvertImage(var BitMap);
```

Remarks:

This procedure performs the logical NOT operation on each byte of the *BitMap* Image.

IsLfbUsed function

Returns True if the Linear Flat framebuffer is used by the current graphic mode.

Declaration:

```
function IsLFBUsed: Boolean;
```

Win32 target:

IsLfbUsed always returns TRUE.

Line procedure

Draws a line from the point (*X1, Y1*) to (*X2, Y2*).

Declaration:

```
procedure Line(X1, Y1, X2, Y2: Longint);
procedure Line(X1, Y1, X2, Y2: Longint; Color: DWORD);
```

Remarks:

Draws a line in the style defined by *SetLineStyle* and uses the color set by *SetColor*. Use *SetWriteMode* to determine whether the line is copied or XOR'd to the screen.

Note that

```
MoveTo(100, 100);
LineTo(200, 200);
```

is equivalent to

```
Line(100, 100, 200, 200);
MoveTo(200, 200);
```

Use *LineTo* when the current pointer (CP) is at one endpoint of the line. If you want the CP updated automatically when the line is drawn, use *LineRel* to draw a line a given direction and distance from the CP. Line doesn't update the CP.

LineRel procedure

Draws a line from the current pointer (CP) along the vector (Dx, Dy), and moves the CP to (X1, X1):=(X0, Y0) + (Dx, Dy)

Declaration:

```
procedure LineRel(Dx, Dy: LongInt);
```

Remarks:

Draws the line from the CP (X0, Y0) to a point (X1, Y1), where
x1 = x0 + Dx
y1 = y0 + Dy

LineTo procedure

Draws a line from the current pointer to (*X*,*Y*).

Declaration:

```
procedure LineTo(X, Y: Longint);
```

Remarks:

Draws a line in the style *SetLineStyle* and uses the color set by *SetColor*. Use *SetWriteMode* to determine whether the line is copied or XOR'd to the screen.

Note that

```
MoveTo(100, 100);
LineTo(200, 200);
```

is equivalent to

```
Line(100, 100, 200, 200);
```

The first method is slower and uses more code. Use *LineTo* only when the current pointer is at one endpoint of the line. Use *LineRel* to draw a line a given distance and direction from the CP. The second method doesn't change the value of the CP.

LineTo moves the current pointer to (*X*, *Y*).

MoveRel procedure

Displaces the current pointer (CP) from its current position.

Declaration:

```
procedure MoveRel(Dx, Dy: Longint);
```

Remarks:

If the CP is at (X1,Y1), *MoveRel* moves it to ((X1 + *Dx*),(Y1 + *Dy*)).

MoveTo procedure

Moves the current pointer (CP) to (*X*,*Y*).

Declaration:

```
procedure MoveTo(X, Y: Longint);
```

Remarks:

The CP is similar to a text mode cursor except that the CP is not visible. The following routines move the CP:

ClearDevice
ClearViewPort
GraphDefaults
SetGraphMode
SetSVGAMode
LineRel
LineTo
MoveRel
MoveTo
OutText
SetViewPort

OutCharXY procedure

Sends a char to the output device.

Declaration:

```
procedure OutCharXY(X, Y: Longint; C: Char; Color: DWord);
```

Remarks:

Displays char *C* at (*X*, *Y*) using given *Color*.

OutCharXY has no affects on the CP.

OutText procedure

Sends a string to the output device at the current pointer.

Declaration:

```
procedure OutText(TextString: string);
```

Remarks:

Displays *TextString* at the CP using the current justification settings.

TextString is truncated at the viewport border if it is too long.

OutText uses the font set by *SetTextStyle*. To maintain code compatibility when using several fonts, use the TextWidth and TextHeight calls to determine the dimensions of the string.

OutText uses the output options set by *SetTextJustify* (justify and center).

The CP is updated by OutText only if the direction is horizontal with left justification. Text output direction is set by *SetTextStyle* (horizontal or vertical); text justification is set by *SetTextJustify* (CP at the left of the string, string centered around CP, or CP at the right of the string—written above CP, below CP, or centered around CP).

OutTextXY procedure

Sends a string to the output device.

Declaration:

```
procedure OutTextXY(X, Y: Longint; TextString: string);
```

Remarks:

Displays *TextString* at (*X, Y*). *TextString* is truncated at the viewport border if it is too long. Use *OutText* to output text at the current pointer; use *OutTextXY* to output text elsewhere on the screen. *OutTextXY* uses the font set by *SetTextStyle*. To maintain code compatibility when using several fonts, use the TextWidth and TextHeight calls to determine the dimensions of the string. *OutTextXY* uses the output options set by *SetTextJustify* (justify and center).

PutHTextel procedure

Puts horizontal set of pixels (horizontal textel) onto the screen.

Declaration:

```
procedure PutHtextel(X1, X2, Y: Longint; var Textel);
```

PutImage procedure

Puts a bit image onto the screen.

Declaration:

```
procedure PutImage(X, Y: Longint; var BitMap);
```

Remarks:

(*X, Y*) is the upper left corner of a rectangular region on the screen. *BitMap* is an untyped parameter that contains the height and width of the region, and the bit image that will be put onto the screen.

PutPixel procedure

Plots a pixel at (*X,Y*).

Declaration:

```
procedure PutPixel(X, Y: Longint; Pixel: DWord);
procedure PutPixel(X, Y: Longint);
```

Remarks:

Plots a point in the color defined by *Pixel* at (*X, Y*).

PutSprite procedure

Puts a bit sprite onto the screen.

Declaration:

```
procedure PutSprite(X1, Y1, X2, Y2: Longint; var Sprite);
```

Rectangle procedure

Draws a rectangle, using the current line style and color.

Declaration:

```
procedure Rectangle(X1, Y1, X2, Y2: Longint);
procedure Rectangle(X1, Y1, X2, Y2: Longint; Color: DWORD);
```

Remarks:

(*X1*, *Y1*) defines the upper left corner of the rectangle, and (*X2*, *Y2*) defines the lower right corner. The rectangle can cross the screen borders.

Draws the rectangle in the current line style and color, as set by *SetLineStyle* and *SetColor*. Use *SetWriteMode* to determine whether the rectangle is copied or XOR'd to the screen.

ReleasePageDC procedure

Targets: Win32 only

Releases a device context (DC) for the active graphical page retrived with the *GetPageDC* function.

Declaration:

```
procedure ReleasePageDC(PageNo: DWORD);
```

RestoreCrtMode procedure

Restores the original screen mode before graphics was initialized the first time.

Declaration:

```
procedure RestoreCrtMode;
```

Remarks:

Restores the original video mode detected by first *SetGraphMode* or *SetSVGAMode* execution. Can be used in conjunction with *SetGraphMode* and *SetSVGAMode* to switch back and forth between text and graphics modes.

Win32 target:

RestoreCrtMode destroys a graphic surface (window) and restores the original video mode.

Retrace procedure

Waits for vertical retrace.

Declaration:

```
procedure Retrace;
```

RGBColor procedure

Packs a set of RGB values into a color value for passing to the primitive drawing routines that are appropriate for the current video mode.

Declaration:

```
function RGBColor(R, G, B: Byte): DWord;
```

Remarks:

This routine is intended to work with RGB video modes such as the 15, 16, 24 and 32 bits per pixel modes.

Use this routine to convert all color values to ensure that they work correctly on the different types of direct color video modes available.

SetActivePage procedure

Set the active page for graphics output.

Declaration:

```
procedure SetActivePage(Page: DWord);
```

Remarks:

Makes *Page* the active graphics page, directing all subsequent graphics output to *Page*. Multiple pages are supported only by the SVGA graphics cards. With multiple graphics pages, a program can direct graphics output to an offscreen page, then quickly display the offscreen image by changing the visual page with the *SetVisualPage* procedure. This technique is especially useful for animation.

Win32 target:

Only two graphic pages are available for the Graph unit.

SetAllPalette procedure

Changes all palette colors as specified.

Declaration:

```
procedure SetAllPalette (var Palette);
```

SetAspectRatio procedure

Changes the default aspect-ratio correction factor.

Declaration:

```
procedure SetAspectRatio(AspectRatio: Real);
```

Remarks:

SetAspectRatio is used to change the default aspect ratio of the current graphics mode. The aspect ratio is used to draw circles. If circles appear elliptical, the monitor is not alligned properly. This can be corrected in hardware by realigning the monitor, or it can be corrected in software by changing the aspect ratio using *SetAspectRatio*. To read the current aspect ratio from the system, use *GetAspectRatio*.

SetBkColor procedure

Sets the current background color, using the palette.

Declaration:

```
procedure SetBkColor(ColorNum: Word);
```

Remarks:

Background color must be in range range [0..*GetMaxColor*], depending on the current graphics driver and the current graphics mode. The background color is used by the *ClearViewPort* and *ClearPage* procedures. *SetBkColor* does not change the first color in the active palette! To change it, use the *GetRGBPalette* and *SetRGBPalette* procedures.

SetColor procedure

Sets the current drawing color, using the palette.

Declaration:

```
procedure SetColor(Color: DWord);
```

Remarks:

Drawing colors must be in range [0..*GetMaxColor*], depending on the current graphics driver and the current graphics mode.

SetCustomFont procedure

Sets user-defined bit-fonts.

Declaration:

```
procedure SetCustomFont(AddrPtr: Pointer; Width, Height, Start,
Space: DWord);
```

Remarks:

AddrPtr points to the fonts data location in memory. The *Width* parameter is the horizontal size of char (in pixels) divided by 8. *Height* is the vertical size of the char (in pixels). The *Start* parameter is an offset of the first symbol in the char’s table.

SetFillColor procedure

Selects a user-defined fill color.

Declaration:

```
procedure SetFillColor(Color: DWord);
```

Remarks:

Sets the color for all filling done by *FillPoly, Bar, Bar3D, FillCircle, FillEllipse, FloodFill* and *ExpandFill*. This procedure cancels the settings made by *SetFillStyle* and enables a solid fill mode.

SetFillPattern procedure

Selects a user-defined fill pattern.

Declaration:

```
procedure SetFillPattern(Pattern: FillPatternType; Color:
DWord);
```

Remarks:

Sets the fill pattern for all filling done by *FillPoly, Bar, Bar3D, FillCircle, FillEllipse* and *FillTriangle* procedures. The Graph unit does not support pattern fill for the *FloodFill* and *ExpandFill* procedures.

FillPatternType is predefined as follows:

```
type
  FillPatternType = array[1..8] of byte;
```

See also: **SetFillColor**

SetFillStyle procedure

Sets the fill pattern and color.

Declaration:

```
procedure SetFillStyle(Pattern: DWord; Color: DWord);
```

Remarks:

Sets the pattern and color for all filling done by *FillPoly, Bar, Bar3D, FillCircle, FillEllipse* and *FillTriangle* procedures. The Graph unit does not support pattern fill for the *FloodFill* and *ExpandFill* procedures. If invalid input is passed to *SetFillStyle, GraphResult* returns a value of *grError*, and the current fill settings will be unchanged. If *Pattern* equals *UserFill*, the user-defined pattern (set by a call to *SetFillPattern*) becomes the active pattern.

See also: **SetFillColor, SetFillPattern**

SetGraphBufSize procedure

Changes the default graphics buffer size used for ellipse drawing.

Declaration:

```
procedure SetGraphBufSize(BufSize: DWord);
```

Remarks:

The internal buffer size is set to *BufSize*, and a buffer is allocated on a call made to *SetGraphMode* or *SetSVGAMode. CloseGraph* frees the allocated buffer. Use *GetGraphBufSize* to get the size in bytes of the internal graphic buffer.

SetGraphMode procedure

Sets the system to graphics mode and clears the screen.

Declaration:

```
procedure SetGraphMode(Mode: Word);
```

Remarks:

Mode must be a valid mode for the current video adapter. This procedure supports all VESA VBE graphic modes (100h - FFFh), which are supported by the video adapter.

Example:

```
SetGraphMode($101) // set SVGA 256-colored mode 640x480;
SetGraphMode($114) // set SVGA 64K-colored mode 800x600;
SetGraphMode($12A) // set SVGA 16M+A-colored mode 1024x768;
```

Keep in mind what your video card may not support all these modes. See **GetVbeModesList** and **TotalVbeModes**.

SetLineStyle procedure

Sets the current line width and style.

Declaration:

```
procedure SetLineStyle(LineStyle: Word; Pattern: Word;
Thickness: Word);
```

Remarks:

Affects all lines drawn by *Line, LineTo, LineRel, Rectangle, DrawPoly*, and so on. Lines can be drawn solid, dotted, centerline, or dashed. If invalid input is passed to *SetLineStyle*, *GraphResult* returns a value of grError, and the current line settings will be unchanged. See *LineSettingsType* for a list of constants used to determine line styles.

LineStyle is a value from SolidLn to UserBitLn(0..4), *Pattern* is ignored unless LineStyle equals UserBitLn, and *Thickness* is NormWidth or ThickWidth. When LineStyle equals UserBitLn, the line is output using the 16-bit pattern defined by the Pattern parameter. For example, if Pattern = $AAAA, then the 16-bit pattern looks like this:

```
1010101010101010        { NormWidth }

1010101010101010        { ThickWidth }
1010101010101010
1010101010101010
```

SetLogicalPage procedure

Sets the logical page size. The *SX* and *SY* valus are the new logical size.

Declaraton:

```
procedure SetLogicalPage(SX, SY: Word);
```

Remarks:

Many SVGA adapters support logical pages. A logical page can exceed the size of the physical screen. For instance, it is possible to install a logical page 1280 x 480 with a screen of physical resolution 640x480. In this case only half the logical page will be seen on the screen. Logical pages are used for hardware scrolling. The maximum size of a logical page depends on the SVGA adapter and the size of the video memory. Use the function *GetLogicalPage* to get the current logical page size. Remember that the number of available graphic pages depends on the logical page size. Thus *SetLogicalPage* influences the number of available graphic pages and resets the viewport to the whole logical page size. Keep in mind, that *SX* and *SY* can't be less than the physical screen size.

Here is an example of use of the logical page:

```
// This example sets a logical page 1280x600 and performs
// hardware scrolling.
// A VESA-compatible SVGA card with 1Mb is required.
//
uses Graph,Crt;
var ErrorCode,i: Longint;
    SX, SY: Word;
begin
 // setup SVGA mode 640x480x256
 SetSVGAMode(640, 480, 8, LfbOrBanked);
 ClearDevice;
 if GraphResult <> 0 then begin
  ErrorCode:=GraphResult;
  CloseGraph;
  Writeln(GraphErrorMsg(ErrorCode));
 end;
 // setup logical page 1280x600
 SetLogicalPage(1280, 600);
 // check logical page size
 GetLogicalPage(SX, SY);
if (SX = 640) and (SY = 480) then begin
  CloseGraph;
  Writeln(' Logical pages not supported...');
 end;
 // draw on logical page
 SetLineStyle(SolidLn, 0, ThickWidth);
 SetColor(clRed);
 Line(0, 0, GetMaxX, GetMaxY);
 Line(GetMaxX, 0, 0, GetMaxY);
 SetColor(clWhite);
 Rectangle(0, 0, GetMaxX, GetMaxY);
 // scroll the screen left
 for i := 0 to (SX - 640) div 4 do
 SetScreenStart(i * 4, 0, True);
 // scroll the screen up
 for i := 0 to (SY - 480) div 4 do
 SetScreenStart(SX - 640, i * 4, True);
 // scroll the screen right
 for i := (SX - 640) div 4 downto 0 do
 SetScreenStart(i * 4, SY - 480, True);
 // scroll the screen down
 for i := (SY - 480) div 4 downto 0 do
 SetScreenStart(0, i * 4, True);
 // Wait a key
 ReadKey;
// Close Graph and restore the old video mode.
 CloseGraph;
end.
```

SetNormalMode procedure

Cancels action of the procedure *SetVirtualMode* and re-directs graphic operations to the active graphic page.

Declaration:

```
procedure SetNormalMode
```

See also: **SetVirtualMode**, **FlipToMemory** and **FlipToScreen**

SetPalette procedure

Changes one palette color as specified by *ColorNum* and *Color*.

Declaration:

```
procedure SetPalette(ColorNum: Word; Color: Word);
```

Remarks:

Changes the *ColorNum* entry in the palette to *Color*. SetPalette (0, clLightCyan) makes the first color in the palette light cyan. *ColorNum* can range from 0 to 255 and works only in 256-colored (palette) modes. If an invalid input is passed to *SetPalette*, *GraphResult* returns a value of grError, and the palette remains unchanged.

SetRGBPalette procedure

Modifies palette entries for the VGA, MCGA and 256-colored SVGA modes.

Declaration:

```
procedure SetRGBPalette(ColorNum, RedValue, GreenValue,
BlueValue: Byte);
```

Remarks:

ColorNum defines the palette entry to be loaded, while *RedValue*, *GreenValue*, and *BlueValue* define the component colors of the palette entry.

SetScreenStart procedure

Targets: MS-DOS only

Sets the CRTC display starting address to the specified value.

Declaration:

```
procedure SetScreenStart(X,Y: DWord; WaitForRetrace: Boolean);
```

Remarks:

You can use this procedure to implement hardware scrolling. You can also use this function to perform double buffering. Keep in mind, that your video card may not support extended CRTC addressing!

If the *WaitForRetrace* flag is False, the routine will not wait for a vertical retrace before programming the CRTC starting address. Otherwise the routine will sync to a vertical retrace. Under VBE 1.2 it is not guaranteed what the behavior will be (some wait and some don’t).

SetSVGAMode procedure

Sets SVGA video mode with given resolution and color depth.

Declaration:

```
procedure SetSVGAMode(XRes, YRes, BPP, VMode: Word);
```

Remarks:

XRes and *YRes* parameters define the required resolution; *BPP* defines the color depth (bits per pixel) and must be in range [8,15,16,24,32]. The *VMode* parameter must be in range [1..3]:

```
  LFBorBanked        = 0 - Set Lfb or Banked modes.
  BankedOnly         = 1 - Set Banked modes only;
  LFBOnly            = 2 - Set Lfb modes only;
```

Example:

`SetSVGAMode(640,480,8,LfbOrBanked)` sets a 256-colored (8 bits per pixel) SVGA mode with resolution 640x480. This procedure will try to install Lfb mode, if it is supported. If not - Banked mode will be installed.

SetTextJustify procedure

Sets text justification values used by *OutText* and *OutTextXY*.

Declaration:

```
procedure SetTextJustify(Horiz, Vert: DWord);
```

Remarks:

Text output after a *SetTextJustify* will be justified around the current pointer in the manner specified. Given the following,

```
SetTextJustify(CenterText, CenterText);
OutTextXY(100, 100, ‘ABC’);
```

The point (100, 100) will appear in the middle of the letter *B*. The default justification settings can be restored by *SetTextJustify* (LeftText, TopText). If an invalid input is passed to *SetTextJustify*, *GraphResult* returns a value of grError, and the current text justification settings will be unchanged.

SetTextStyle procedure

Sets style for text output in graphics mode.

Declaration:

```
procedure SetTextStyle(Font, Direction: DWord);
```

SetSplineLineSteps procedure

Adjusts smoothing factor used to draw the spline.

Declaration:

```
procedure SetSplineSteps(Steps: DWord);
```

Remarks:

By default smoothing factor (Steps value) = 30. See **Spline** procedure.

SetTranspMode procedure

Sets/resets transparent mode for BitMaps output.

Declaration:

```
procedure SetTranspMode(Mode: Boolean; Color: DWord);
```

Remarks:

TMT Graph supports transparent BitMaps blt. If the *Mode* parameter is True, all pixels consisting of BitMap pixels with a color which is different from the given *Color* value will be put to the screen. Any pixel which coincides with the value *Color* will be ignored (will not be placed on the screen). This is very useful for games programming. Transparent BitMaps blt are supported in any graphic mode (MCGA/VGA, SVGA 256, 32K, 64K, 16M and 16M+A colored modes, with LFB and banked modes)!

SetTranspMode affects calls to the following routines only: *PutHTextel, PutSprite* and *PutImage.*

SetViewPort procedure

Sets the current output viewport or window for graphics output

Declaration:

```
procedure SetViewPort(X1, Y1, X2, Y2: Longint; Clip: Boolean);
```

Remarks:

(*X1, Y1*) defines the upper left corner of the viewport, and (*X2, Y2*) defines the lower right corner. The upper left corner of a viewport is (0, 0).

The Boolean parameter *Clip* determines whether drawings are clipped at the current viewport boundaries.

```
SetViewPort (0, 0, GetMaxX, GetMaxY, True)
```

always sets the viewport to the entire graphics screen. If (*X1>=X2*) or (*Y1>=Y2*), *GraphResult* returns grError, and the current view settings will be unchanged. The TMT Graph unit allows the assignment of viewport outside the screen!

All graphics commands (for example: *GetX*, *OutText*, *Line* and so on) are viewport-relative.

If the Boolean parameter *Clip* is set to True when a call to *SetViewPort* is made, all drawings will be clipped to the current viewport. Note that the current pointer is never clipped. The following will not draw the whole line requested because the line will be clipped to the current viewport:

```
SetViewPort(10, 10, 20, 20, ClipOn);
Line(0, 5, 15, 5);
```

The line would start at absolute coordinates (10,15) and terminate at absolute coordinates (25, 15) if no clipping were performed. But because clipping was performed, the actual line drawn would start at absolute coordinates (10, 15) and terminate at coordinates (20, 15). *SetSVGAMode, GraphDefaults,* and *SetGraphMode* all reset the viewport to the entire graphics screen. The current viewport settings are available by calling the procedure *GetViewSettings*, which accepts a parameter of *ViewPortType*.

SetViewPort moves the current pointer to (0, 0).

SetVirtualMode procedure

Sets virtual graphic mode.

Declaration:

```
procedure SetVirtualMode(BuffAddr: Pointer)
```

Remarks:

This procedure re-directs all graphic operations directly to the virtual video buffer pointed to by *BuffAddr* in the memory. You must previously allocate a memory block for the virtual buffer using *GetMem* procedure. Use *GetPageSize* to obtain the size of the video page to be allocated.

See also: **SetNormalMode**, **FlipToScreen** and **FlipToMemory**

SetVisualPage procedure

Sets the visual graphics page number.

Declaration:

```
procedure SetVisualPage(Page: DWord; WaitForRetrace: Boolean);
```

Remarks:

Makes Page the visual graphics page.

Multiple pages are supported only by the SVGA graphics modes. With multiple graphics pages, a program can direct graphics output to an offscreen page, then quickly display the offscreen image by changing the visual page with the *SetVisualPage* procedure. This technique is especially useful for animation. If the *WaitForRetrace* flag is not True, this routine will not sync to the vertical retrace. This flag is now part of the VBE 2.0 spec, and if you have a VBE 1.2 device you cannot be sure that the device will sync to the retrace anyway.

Win32 target:

Only two graphic pages are available for Graph unit.

SetWriteMode procedure

Sets the writing mode for line drawing.

Declaration:

```
procedure SetWriteMode(WriteMode: DWord);
```

Remarks:

Each binary operation constant corresponds to a binary operation between each byte in the line and the corresponding bytes on the screen. CopyPut uses the assembly language MOV instruction, overwriting the line with whatever is on the screen. XORPut uses the XOR command to combine the line with the screen. Two successive XOR commands will erase the line and restore the screen to its original appearance.

SetWriteMode affects calls to the following routines only: *DrawPoly, PutPixel, Line, LineRel, LineTo, Rectangle, Circle, Ellipse, Bar3D, PutHTextel, PutSprite, DrawEllipse, Triangle* and *PutImage.*

Spline procedure

Draws a spline.

Declaration:

```
procedure Spline(Nodes: Byte; Points: array of PointType);
```

Remarks:

Fits a smooth curve through a given set of points. Nodes specifies the number of Points. A coordinate pair consists of two Longints, an X and a Y value.

Use the *SetSplineLineSteps* procedure to adjust the spline.

Stretch procedure

Targets: MS-DOS only

Stretches screen vertical in *Param* times.

Declaration:

```
procedure Stretch(Param: Byte);
```

TextHeight function

Returns the height of a string, in pixels.

Declaration:

```
function TextHeight(TextString: string): DWord;
```

Remarks:

Takes the current font size and multiplication factor, and determines the height of the *TextString* in pixels. This is useful for adjusting the spacing between lines, computing viewport heights, sizing a title to make it fit on a graph or in a box, etc.

For example, with an 8x8 bit-mapped font and a multiplication factor of 1 (set by *SetTextStyle*), the string Pascal is 8 pixels high.

It is important to use *TextHeight* to compute the height of strings, instead of doing the computation manually. In that way, no source code modifications have to be made when different fonts are selected.

TextWidth function

Returns the width of a string in pixels.

Declaration:

```
function TextWidth(TextString: string): DWord;
```

Remarks:

Takes the string length, current font size, and multiplication factor, and determines the width of *TextString* in pixels. This is useful for computing view-port widths, sizing a title to make it fit on a graph or in a box, and so on.

For example, with the 8x8 bit-mapped font (set by *SetTextStyle*), the string TMT is 24 pixels wide.

It is important to use *TextWidth* to compute the width of strings, instead of doing the computation manually. In that way, no source code modifications have to be made when different fonts are selected.

TotalVbeMemory function

Returns the size in bytes of the total video memory.

Declaration:

```
function TotalVbeMemory: DWord;
```

MS-DOS target:

If VESA VBE 1.1+ is not supported, this function will return zero.

TotalVbeModes function

Returns the total number of supported VESA modes.

Declaration:

```
function TotalVbeModes: Word;
```

Remarks:

Use *GetVbeModesList* to get info on supported modes.

Triangle procedure

Draws a triangle.

Declaration:

```
procedure Triangle(X1, Y1, X2, Y2, X3, Y3: Longint);
```

```
procedure Triangle(X1, Y1, X2, Y2, X3, Y3: Longint; Color:
DWORD);
```

Chapter 8

The Keyboard Unit

Targets: MS-DOS, OS/2, Win32 console

The Keyboard unit contains 13 procedures and functions for advanced keyboard control. These routines allow one to create Win32 console applications easily.

8.1 Keyboard Unit Constants

The following constants are defined in the Keyboard Unit:

const

```
  ESC_Scan:     Byte = $01;      Ent_Scan:     Byte = $1c;
  Back_Scan:    Byte = $0e;      Rsh_Scan:     Byte = $36;
  Ctrl_Scan:    Byte = $1d;      Prt_Scan:     Byte = $37;
  Lsh_Scan:     Byte = $2a;      Alt_Scan:     Byte = $38;
  Caps_Scan:    Byte = $3a;      Home_Scan:    Byte = $47;
  F1_Scan:      Byte = $3b;      Up_Scan:      Byte = $48;
  F2_Scan:      Byte = $3c;      PgUp_Scan:    Byte = $49;
  F3_Scan:      Byte = $3d;      Min_Scan:     Byte = $4a;
  F4_Scan:      Byte = $3e;      Left_Scan:    Byte = $4b;
  F5_Scan:      Byte = $3f;      Mid_Scan:     Byte = $4c;
  F6_Scan:      Byte = $40;      Right_Scan:   Byte = $4d;
  F7_Scan:      Byte = $41;      Plus_Scan:    Byte = $4e;
  F8_Scan:      Byte = $42;      End_Scan:     Byte = $4f;
  F9_Scan:      Byte = $43;      Down_Scan:    Byte = $50;
  F10_Scan:     Byte = $44;      PgDn_Scan:    Byte = $51;
  F11_Scan:     Byte = $d9;      Ins_Scan:     Byte = $52;
  F12_Scan:     Byte = $da;      Del_Scan:     Byte = $53;
  Scrl_Scan:    Byte = $46;      Num_Scan:     Byte = $45;
  Tab_Scan:     Byte = $0f;      Space_Scan:   Byte = $39;
```

8.2 Keyboard Unit Procedures and Functions

Targets: MS-DOS, Win32 console

AsciiToScan

Translates a given OEM ASCII character into the scan code.

Declaration:

```
function AsciiToScan(AsciiChar: Char): Byte;
```

See also: **ScanToAscii**

GetKey

Returns a character and an extended scan code.

Declaration:

```
function GetKey: WORD;
```

Remarks:

The *GetKey* function returns the character code in its low-order byte *(LoByte(GetKey))* and the extended scan code in its high-order byte *(HiByte(GetKey))*.

FlushKeyboard

Flushes all contents of the keyboard buffer.

Declaration:

```
procedure FlushKeyboard;
```

Win32 console target:

The *FlushKeyboard* procedure flushes the console input buffer. All input records currently in the input buffer are discarded.

MultikeysDone

Stops a multi-keys mode.

Declaration:

```
procedure MultikeysDone
```

See also: **MultikeysInit**

MultikeysInit

Initializes a multi-keys mode.

Declaration:

```
procedure MultikeysInit;
```

Remarks:

The multi-keys mode allows one to control the state of simultaneously pressed keys.

See also: **MultikeysDone, GetKey**

ScanToAscii

Translates a given scan code into the OEM ASCII character.

Declaration:

```
function ScanToAscii(ScanCode: Byte): Char;
```

Remarks:

The *ScanToAscii* function returns the OEM ASCII character in uppercase.

See also: **AsciiToScan**

TestAlt

Returns TRUE if the ALT key is pressed, FALSE otherwise.

Declaration:

```
function TestAlt: Boolean;
```

TestCapsLock

Returns TRUE if the CAPS LOCK light is on, FALSE otherwise.

Declaration:

```
function TestCapsLock: Boolean;
```

TestCtrl

Returns TRUE if the CTRL key is pressed, FALSE otherwise.

Declaration:

```
function TestCtrl: Boolean;
```

TestNumLock

Returns TRUE if the NUM LOCK light is on, FALSE otherwise.

Declaration:

```
function TestNumLock: Boolean;
```

TestScrollLock

Returns TRUE if the SCROLL LOCK light is on, FALSE otherwise.

Declaration:

```
function TestScrollLock: Boolean;
```

TestShift

Returns TRUE if the SHIFT key is pressed, FALSE otherwise.

Declaration:

```
function TestShift: Boolean;
```

TestKey

Checks the state of a key with a given scan code.

Declaration:

```
function TestKey(Scan: Byte): Boolean;
```

Remarks:

The *TestKey* function returns TRUE if the key with the given scan code is pressed, FALSE otherwise. This function works in multi-keys mode only (see **MultikeysInit**).

Example:

```
{$ifndef __CON__}
  This program must be compiled as an MS-DOS or Win32 console
application
{$endif}
uses Keyboard;
begin
  MultikeysInit;
  Writeln('Press [Esc]+[Space] to exit..');
  repeat
    (* Wait loop *)
  until (TestKey(Space_Scan) and TestKey(Esc_Scan));
  Writeln('Ok.');
  MultikeysDone;
end.
```

Chapter 9

The Math Unit

Targets: MS-DOS, OS/2, Win32

9.1 Math Unit Constants

Following constants are defined in the Math unit

```
MinSingle   = 1.5E-45;
MaxSingle   = 3.4E+38;
MinDouble   = 5.0E-324;
MaxDouble   = 1.7E+308;
MaxExtended = MaxDouble;
MinExtended = MinDouble;

HalfLnMax   = 3.54863405227661E+0002;
Deg2Rad     = 1.74532925199433E-0002;
Rad2Deg     = 5.72957795130823E+0001;
Grad2Rad    = 1.57079632679490E-0002;
Rad2Grad    = 6.36619772367581E+0001;
DoublePI    = 6.28318530717959E+0000;
e           = 2.71828182845905E+0000;
```

9.2 Math Unit Procedures and Functions

ArcCos function

Calculates the inverse cosine of the given number.

Declaration:

```
function ArcCos(X: Extended): Extended;
```

ArcCosH function

Calculates the inverse hyperbolic cosine of the given number.

Declaration:

```
function ArcCosH(X: Extended): Extended;
```

ArcCotan function

Calculates the inverse cotangent of the given number.

Declaration:

```
function ArcCotan(X: Extended): Extended;
```

ArcCotanH function

Calculates the inverse hyperbolic cotangent of the given number.

Declaration:

```
function ArcCotanH(X: Extended): Extended;
```

ArcCsc function

Calculates the inverse cosecant of the given number.

Declaration:

```
function ArcCsc(X: Extended): Extended;
```

ArcCscH function

Calculates the inverse hyperbolic cosecant of the given number.

Declaration:

```
function ArcCscH(X: Extended): Extended;
```

ArcSec function

Calculates the inverse secant of the given number.

Declaration:

```
function ArcSec(X: Extended): Extended;
```

ArcSecH function

Calculates the inverse hyperbolic secant of the given number.

Declaration:

```
function ArcSecH(X: Extended): Extended;
```

ArcSin function

Calculates the inverse sine of the given number.

Declaration:

```
function ArcSin(X: Extended): Extended;
```

ArcSinH function

Calculates the inverse hyperbolic sine of the given number.

Declaration:

```
function ArcSinH(X: Extended): Extended;
```

ArcTan2 function

Calculates the arctangent angle and quadrant of the given number.

Declaration:

```
function ArcTan2(Y, X: Extended): Extended;
```

Remarks:

The *ArcTan2* function calculates ArcTan(Y/X), and returns an angle in the correct quadrant. The values of *X* and *Y* must be between -264 and 264. *X* can not be 0. The return value will fall in the range from -Pi to Pi radians.

ArcTanH function

Calculates the inverse hyperbolic tangent of the given number.

Declaration:

```
function ArcTanH(X: Extended): Extended;
```

Ceil function

Rounds variables up toward positive inifinity.

Declaration:

```
function Ceil(X: Extended): Extended;
function Ceil(X: Extended): Longint;
```

Remarks:

Use *Ceil* to obtain the lowest integer greater than or equal to X.

CelsToFahr function

Converts Celsius to Fahrenheit.

Declaration:

```
function CelsToFahr(Celsius: Extended): Extended;
```

Remarks:

Use *CelsToFahr* to convert temperature measured in Celsius into Fahrenheit. *CelsToFahr* uses the formula

Celsius * 9.0 / 5.0 + 32.0

ChgSign function

Reverses the sign of a double-precision floating-point argument.

Declaration:

```
function ChgSign(X: Extended): Extended;
```

Remarks:

ChgSign returns a value equal to its floating-point argument *X*, but with its sign reversed.

CmToInch function

Converts centimetres to inches.

Declaration:

```
function CmToInch(Cm: Extended): Extended;
```

Remarks:

Use *CmToInch* to convert length measured in centimetres into inches. *CmToInch* uses the formula

Cm / 2.54

CopySign function

Return one value with the sign of another.

Declaration:

```
function CopySign(X, Y: Extended): Extended;
```

Remarks:

CopySign returns its floating point argument *X* with the same sign as its floating-point argument *Y*. There is no error return.

CosH function

Calculates the hyperbolic cosine of given angle.

Declaration:

```
function CosH(Angle: Extended): Extended;
```

Remarks:

Use the *CosH* function to calculate the hyperbolic cosine of *X*.

Cotan function

Returns the cotangent of an *Angle* in radians.

Declaration:

```
function Cotan(Angle: Extended): Extended;
```

Remarks:

Cotan(Angle) = Cos(Angle) / Sin(Angle).

Csc function

Returns the cosecant of *Angle* in radians.

Declaration:

```
function Csc(Angle: Extended): Extended;
```

Remarks:

Csc(Angle) = 1 / Sin(Angle).

CscH function

Returns the hyperbolic cosecant of *Angle* in radians.

Declaration:

```
function CscH(Angle: Extended): Extended;
```

Cterm function

Returns the number of compounding periods.

Declaration:

```
function Cterm(Rate: Extended; FutureValue, PresentValue:
Extended): Extended;
```

CycleToRad function

Converts an angle measurement from cycles to radians

Declaration:

```
function CycleToRad(Cycle: Extended): Extended;
```

Remarks:

Use *CycleToRad* to convert angles measured in cycles into radians. *CycleToRad* uses the formula

Cycle * 2 * PI

DeltaPercent function

Compute percent deviation.

Declaration:

```
function DeltaPercent(Value1, Value2: Extended): Extended;
```

Remarks:

The *DeltaPercent* returns deviation between Value2 percent (%) from Value1. The result is less than 1.

DegToRad function

Returns the value of a degree measurement expressed in radians.

Declaration:

```
function DegToRad(Degrees: Extended): Extended;
```

Remarks:

Use *DegToRad* to convert angles expressed in degrees to the corresponding value in radians. *DegToRad* uses the formula

Degrees * PI / 180

Evaluate procedure

Evaluates the given expression.

Declaration:

```
procedure Evaluate(Expr: String; var Result: Extended; var
ErrCode: Longint);
```

Remarks:

Use *Evaluate* to evaluate any valid expression given in *Expr*. The *Result* variable returns the evaluated expression. If it succeeds, the *ErrCode* variable is 0.

Example:

```
uses Math, Strings;
const
  Expr = ‘cos(0)*(5^2-3.1)’;

var

  Result: Extended;
  ErrCode: Longint;
begin
      Evaluate(Expr, Result, ErrCode);
      if ErrCode = 0 then
        Writeln(Expr, ‘ = ‘, Fls(Result))
      else
        Writeln(‘Invalid expression’);
end.
```

The example above will print 21.9.

FahrToCels function

Converts Fahrenheit to Celsius.

Declaration:

```
function FahrToCels(Fahr: Extended): Extended;
```

Remarks:

Use *FahrToCels* to convert temperature measured in Fahrenheit into Celsius. *FahrToCels* uses the formula

(Fahr - 32.0) * 5.0 / 9.0

Floor function

Rounds variables toward negative infinity.

Declaration:

```
function Floor(X: Extended): Extended;
function Floor(X: Extended): Longint;
```

Remarks:

Use *Floor* to obtain the highest integer less than or equal to X.

FMod function

Calculates the floating-point remainder.

Declaration:

```
function Fmod(X, Y: Extended): Extended;
```

Remarks:

The *Fmod* function calculates the floating-point remainder F of X / Y such that $X = i * Y + F$, where I is an integer, F has the same sign as X, and the absolute value of F is less than the absolute value of Y.

Fv function

Calculates the Future Value.

Declaration:

```
function Fv(Payment: Extended; Rate: Extended; Term: DWord):
Extended;
```

GalToLitre function

Converts US gallons to litres.

Declaration:

```
function GalToLitre(Gallons: Extended): Extended;
```

Remarks:

Use *GalToLitre* to convert volume measured in gallons into litres. *GalToLitre* uses the formula:

Gallons * 3.785411784

GradToRad function

Converts grads to radians.

Declaration:

```
function GradToRad(Grad: Extended): Extended;
```

Remarks:

The *GradToRad* function converts angles of grad measure into radians. *GradToRad* uses the formula

Grad * PI / 200

Hypot function

Calculates the length of the hypotenuse.

Declaration:

```
function Hypot(X, Y: Extended): Extended;
```

Remarks:

The *Hypot* function returns the length of the hypotenuse of a right triangle. Specify the lengths of the sides adjacent to the right angle as *X* and *Y*. Hypot uses the formula:

$$\sqrt{x^2 + y^2}$$

InchToCm function

Converts inches to centimetres.

Declaration:

```
function InchToCm(Inches: Extended): Extended;
```

Remarks:

Use *InchToCm* to convert length measured in inches into centimetres. *InchToCm* uses the formula

Inches * 2.54

KgToLb function

Converts kilograms to pounds.

Declaration:

```
function KgToLb(Kilograms: Extended): Extended;
```

Remarks:

Use *KgToLb* to convert weight measured in kilograms into pounds. *KgToLb* uses the formula

Kg / 0.45359237

LbToKg function

Converts pounds to kilograms.

Declaration:

```
function LbToKg(Pounds: Extended): Extended;
```

Remarks:

Use *LbToKg* to convert weight measured in pounds into kilograms. *LbToKg* uses the formula

Pounds * 0.45359237

LitreToGal function

Converts litres to US gallons.

Declaration:

```
function LitreToGal(Litre: Extended): Extended;
```

Remarks:

Use *LitreToGal* to convert volume measured in litres into gallons. *LitreToGal* uses the formula

Litre / 3.785411784

Log10 function

Calculates log base 10.

Declaration:

```
function Log10(X: Extended): Extended;
```

Remarks:

The *Log10* function returns the log (base 10) of *X*.

Log2 function

Calculates log base 2.

Declaration:

```
function Log2(X: Extended): Extended;
```

Remarks:

The *Log2* function returns the log (base 2) of *X*.

LogN function

Calculates log with a specified base.

Declaration:

```
function LogN(Base, X: Extended): Extended;
```

Remarks:

The *LogN* function returns the log (with specified base) of *X*.

LRotL and LRotR functions

Rotate bits to the left (*LRotL*) or right (*LRotR*).

Declaration:

```
function LRotL(Value, Shift: DWORD): DWORD;
function LRotR(Value, Shift: DWORD): DWORD;
```

Remarks:

The *LRotL* and *LRotR* functions rotate *Value* by *Shift* bits. *LRotL* rotates the value left. *LRotR* rotates the value right. Both functions «wrap» bits rotated off one end of *Value* to the other end.

Max function

Returns the greater of two numeric values.

Declaration:

```
function Max(A, B: Longint): Longint;
function Max(A, B: Single): Single;
function Max(A, B: Double): Double;
function Max(A, B: Extended): Extended;
```

Remarks:

Use *Max* to compare two numeric values. *Max* returns the greater value of the two.

Min function

Returns the smaller of two numeric values.

Declaration:

```
function Min(A, B: Longint): Longint;
function Min(A, B: Single): Single;
function Min(A, B: Double): Double;
function Min(A, B: Extended): Extended;
```

Remarks:

Use *Min* to compare two numeric values. *Min* returns the smaller value of the two.

Modf function

Splits a floating-point value into fractional and integer parts.

Declaration:

```
function Modf(X: Extended; var Y: Longint): Extended;
function Modf(X: Extended; var Y: Integer): Extended;
```

Remarks:

The *Modf* function breaks down the floating-point value *X* into fractional and integer parts, each of which has the same sign as *X*. The signed fractional portion of *X* is returned. The integer portion is stored as a floating-point value at *Y*.

Npv function

Calculates the Net Present Value.

Declaration:

```
function Npv(Rate: Extended; Series: array of Double):
Extended;
```

Percent function

Compute percent (%).

Declaration:

```
function Percent(Value1, Value2: Extended): Extended;
```

Remarks:

The *Percent* returns Value2 percent (%) from Value1. *Percent* uses the formula:

100 * Value2 / Value1

Pmt function

Calculates a fully amortized payment.

Declaration:

```
function Pmt(Principal: Extended; Rate: Extended; Term: DWord):
Extended;
```

Power function

Raises *Base* to any power specified.

Declaration:

```
function Power(Base, Exponent: Extended): Extended;
function Power(Base: Extended; Exponent: Longint): Extended;
```

Remarks:

The *Power* function raises *Base* to any power by *Exponent* parameter. *Base* must be a positive real number or an integer.

Pv function

Calculates the Present value.

Declaration:

```
function Pv(Payment: Extended; Rate: Extended; Term: DWord):
Extended;
```

RadToCycle function

Converts radians to cycles.

Declaration:

```
function RadToCycle(Rad: Extended): Extended;
```

Remarks:

Use *RadToCycle* to convert angles measured in radians into cycles. *RadToCycle* uses the formula:

Rad / (2 * PI)

RadToDeg function

Returns the value of a radian measurement expressed in degrees.

Declaration:

```
function RadToDeg(Rad: Extended): Extended;
```

Remarks:

Use *RadToDeg* to convert angles measured in radians to degrees. *RadToDeg* uses the formula:

Rad * 180 / PI

RadToGrad function

Converts radians to grads.

Declaration:

```
function RadToGrad(Rad: Extended): Extended;
```

Remarks:

The *RadToGrad* function converts angles of radian measure into grads. *RadToGrad* uses the formula:

Rad * 200.0 / PI

Rate function

Returns the periodic interest rate.

Declaration:

```
function Rate(FutureValue, PresentValue: Extended; Term:
DWord): Extended;
```

Sec function

Returns the secant of *Angle* in radians.

Declaration:

```
function Sec(Angle: Extended): Extended;
```

Remarks:

Sec(Angle) = 1 / Cos(Angle).

SecH function

Returns the hyperbolic secant of *Angle* in radians.

Declaration:

```
function SecH(Angle: Extended): Extended;
```

Sgn function

Returns the sign of a variable.

Declaration:

```
function Sgn(X: Extended): Longint;
function Sgn(X: Longint): Longint;
```

Remarks:

Use *Sgn* to obtain the sign of X. For example:

Sgn(3.2) = 1
Sgn(-5) = -1
Sgn(0)   = 0

SinH function

Calculates the hyperbolic sine of a given angle.

Declaration:

```
function SinH(Angle: Extended): Extended;
```

Remarks:

Use the *SinH* function to calculate the hyperbolic sine of *X*.

Sln function

Returns the straight-line depreciation allowance of an asset.

Declaration:

```
function Sln(InitialValue, Residue: Extended; Time: DWord):
Extended;
```

Remarks:

The *Sln* function calculates the straight-line depreciation allowance for an asset over one period of its life. The function divides the *InitialValue* minus the *Residue* by the number of years of useful *Time* of the asset. *InitialValue* is the amount initially paid for the asset. *Residue* is the value of the asset at the end of its useful life.

Syd function

Returns the sum of the year digits depreciation.

Declaration:

```
function Syd(InitialValue, Residue: Extended; Period, Time:
DWord): Extended;
```

Remarks:

The *Syd* function calculates depreciation amounts for an asset using an accelerated depreciation method. This allows for higher depreciation in the earlier years of an asset's life. *InitialValue* is the initial cost of the asset. *Residue* is the value of the asset at the end of its life expectancy. *Time* is the length of the asset's life expectancy. *Period* is the period for which to calculate the depreciation.

Tan function

Tan returns the tangent of *Angle* in radians.

Declaration:

```
function Tan(Angle: Extended): Extended;
```

Remarks:

Tan(Angle) = Sin(Angle) / Cos(Angle).

TanH function

Calculates the hyperbolic tangent of given angle.

Declaration:

```
function TanH(Angle: Extended): Extended;
```

Remarks:

Use the *TanH* function to calculate the hyperbolic tangent of *X*.

Term function

Returns the number of payments.

Declaration:

```
function Term(Payment: Extended; Rate: Extended; FutureValue:
Extended): Extended;
```

Chapter 10

The MMedia Unit

Targets: Win32 only

The MMedia unit implements only one object, called **TMMedia**, which provides an easy and powerful access to multimedia devices such as a CD-ROM player, audio player/recorder, video player/recorder, or MIDI sequencer.

10.1 TMMedia Object Fields

TMMedia.AutoRewind

Determines if the TMMedia object rewinds before playing or recording.

Declaration:

AutoRewind: Boolean;

TMMedia.MCErrorProc

Points to the error handler procedure which gets called when the TMMedia object tries to open non-supported media files. This procedure will be also called every time the TMMedia object tries to perform any operation if the media device has not been previously open.

Declaration:

```
MCIErrorProc: procedure(ErrorCode: UINT);
```

Remarks:

By default the *MCIErrorProc* points to the following error handler procedure:

```
procedure DefaultMCIErrorProc(ErrorCode: UINT);
var
  Temp: array [0..MAX_PATH] of char;
begin
  if isConsole then
    Writeln('MCI device fault. Error code #' +
IntToStr(ErrorCode))
  else
    MessageBox(0, StrPCopy(Temp, 'MCI device fault. Error code
#' + IntToStr(ErrorCode)), 'Error', MB_ICONERROR);
end;
```

TMMedia.Notify

Determines whether a MM_MCINOTIFY message will be send to an application.

Declaration:

```
Notify: Boolean;
```

Remarks:

If *Notify* is TRUE, the next TMMedia object method generates a MM_MCINOTIFY message upon completion. If *Notify* is FALSE, the method does not generate a MM_MCINOTIFY message.

Notify is set to FALSE by default.

TMMedia.PlayFullScreen

Determines if the TMMedia object plays an animation in fullscreen mode.

Declaration:

```
PlayFullScreen: Boolean;
```

Remarks:

PlayFullScreen field affects Digital-Video and Animation media types.

TMMedia.PlayRepeat

Determines if the TMMedia object will start again at the beginning when the end of the content is reached.

Declaration:

```
PlayRepeat: Boolean;
```

Remarks:

The *PlayRepeat* field affects Digital-Video and Animation media types.

TMMedia.StopAtClose

Determines whether a TMMedia object stops media playing when the **Close** method is called.

Declaration:

```
StopAtClose: Boolean;
```

TMMedia.Wait

Determines whether a TMMedia method returns control to the application only after it has been completed.

Declaration:

```
Wait: Boolean;
```

Remarks:

If *Wait* is TRUE, the TMMedia object waits until the next TMMedia object method has finished before returning control to the application. If *Wait* is FALSE, the application won't wait for the next TMMedia object method to finish before continuing.

Wait is set to FALSE by default, so it is recommended to set **Notify** to TRUE so the application is notified when the TMMedia object method finishes.

10.2 TMMedia Object Methods

TMMedia.CDMediaIsPresent

Returns TRUE if the media is inserted in the device; it is set to FALSE otherwise.

Declaration:

```
function CDMediaPresent: Boolean;
```

TMMedia.Close

Closes the open multimedia device.

Declaration:

```
destructor Close;
```

Remarks:

The **StopAtClose** field determines whether the currently playing media is stopped when the *Close* method is called. The **Wait** field determines whether control is returned to the application before the *Close* method has finished. The **Notify** field determines whether *Close* generates a MM_MCINOTIFY message.

TMMedia.Create

Creates the TMMdeia object.

Declaration:

```
constructor Create(OwnerHandle: THandle);
```

Remarks:

The *OwnerHandle* parameter specifies a handle of the window, which will receive the MM_MCINOTIFY message, if **Notify** filed is TRUE.

TMMedia.ErrorCode

Returns the error code after the last performed multimedia operation.

Declaration:

```
function Error: UINT;
```

TMMedia.GetDevice

Returns a multimedia device type.

Declaration:

```
function GetDevice: DWORD;
```

Remarks:

The result value may be one of following MCI_AutoSelect, MCI_AVIVideo, MCI_CDAudio, MCI_DAT, MCI_DigitalVideo, MCI_MMMovie, MCI_Other, MCI_Overlay, MCI_Scanner, MCI_Sequencer, MCI_VCR, MCI_VideoDisc, MCI_WaveAudio.

TMMedia.GetDeviceCaps

Retrieves static information about a device. All devices recognize this command. Information is returned in the *DeviceCaps* variable of the *TMCIDeviceCaps* structure (see below).

Declaration:

```
procedure GetDeviceCaps(var DeviceCaps: TMCIDeviceCaps);
```

Remarks:

TMCIDeviceCaps structure is defined as follows:

```
TMCIDeviceCaps = record
    CanPlay:            Boolean;
    CanRecord:          Boolean;
    CanEject:           Boolean;
    CanSave:            Boolean;
    CanChangePos:       Boolean;
    HasVideo:           Boolean;
    HasAudio:           Boolean;
    ActiveRect:         TRect;
    ActiveRectWidth:    Longint;
    ActiveRectHeight:   Longint;
  end;
```

TMMedia.GetDeviceID

Returns the multimedia device ID.

Declaration:

function GetDeviceID: DWORD;

Remarks:

The multimedia device ID is used for low-level operations such as **mciSendCommand** (see the **Microsoft Multimedia Programmer’s Guide**).

TMMedia.GetFirstAudioTrack

Returns the number of the first playable CD audio track if there is one. Otherwise it returns zero.

Declaration:

```
function GetFirstAudioTrack: DWORD;
```

TMMedia.GetLength

Returns the length of the medium in the open multimedia device.

Declaration:

```
function GetLength: DWORD;
```

Remarks:

Length is specified using the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.GetPos

Returns the current position within the currently loaded (opened) medium.

Declaration:

```
function GetPos: Longint;
```

Remarks:

The result is returned in the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.GetStatus

Returns the state of the currently open multimedia device.

Declaration:

```
function GetStatus: DWORD
```

Remarks:

The following table lists the possible return values for the *GetStatus* function:

Value	Mode
MCI_MODE_NOT_READY	Not ready
MCI_MODE_STOP	Stopped
MCI_MODE_PLAY	Playing
MCI_MODE_RECORD	Recording
MCI_MODE_SEEK	Seeking
MCI_MODE_PAUSE	Paused
MCI_MODE_OPEN	Open

TMMedia.GetTimeFormat

Returns the time format used to obtain and specify position information.

Declaration:

```
function GetTimeFormat: DWORD;
```

Remarks:

Not all formats are supported by every device. When trying to set an unsupported format, the assignment is ignored. The current timing information is always passed as a 4-byte DWORD value.

The following table lists the possible return values for of the *GetTimeFormat* function:

Value	Time format
MCI_FORMAT_MILLISECONDS	Milliseconds are stored as a 4-byte variable.
MCI_FORMAT_HMS	Hours, minutes, and seconds packed into a 4-byte variable.
MCI_FORMAT_MSF	Minutes, seconds, and frames packed into a 4-byte variable.
MCI_FORMAT_FRAMES	Frames are stored as a 4-byte variable.
MCI_FORMAT_SMPTE_24	24-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_25	25-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_30	30-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_30DROP	30-drop-frame SMPTE packs data into the 4-byte variable.
MCI_FORMAT_BYTES	Bytes are stored as a 4-byte variable
MCI_FORMAT_SAMPLES	Samples are stored as a 4-byte integer variable
MCI_FORMAT_TMSF	Tracks, minutes, seconds, and frames are packed in the 4-byte variable.

TMMedia.GetTrackFormat

Returns the format of the given CD audio track.

Declaration:

```
function GetTrackFormat(TrackNo: DWORD): DWORD;
```

Remarks:

The return value is set to one of the following values:

Value	Meaning
MCI_CDA_TRACK_AUDIO	current track is an audio track
MCI_CDA_TRACK_OTHER	current track is a data track

TMMedia.GetTrackLength

Returns the length of the given track.

Declaration:

```
function GetTrackLength(TrackNum: DWORD): DWORD;
```

Remarks:

The returned value is specified according to the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.GetTracksCount

Returns the number of playable tracks on the open multimedia device.

Declaration:

```
function GetTracksCount: Longint;
```

Remarks:

The resulting value is undefined for devices that don’t use tracks.

TMMedia.GetTrackPos

Returns the starting position of the given track.

Declaration:

```
function GetTrackPos(TrackNum: UINT): Longint;
```

Remarks:

The returned value is specified according to the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.GotoFirstAudioTrack

Moves to the beginning of the first CD audio track (if any).

Declaration:

```
function GotoFirstAudioTrack: Boolean;
```

Remarks:

Returns TRUE if successful, FALSE otherwise.

The **Wait** field determines whether control is returned to the application before the *GotoFirstAudioTrack* function has finished. The **Notify** field determines whether *GotoFirstAudioTrack* generates a MM_MCINOTIFY message.

TMMedia.Next

Moves to the beginning of the next track of currently loaded media.

Declaration:

```
procedure Next;
```

Remarks:

If the current position is at the last track when the *Next* method is called, *Next* makes the current position the beginning of the last track. If the multimedia device doesn’t use tracks, *Next* goes to the end of the medium.

The **Wait** field determines whether control is returned to the application before the **Next** method has finished. The **Notify** field determines whether *Next* generates a MM_MCINOTIFY message.

TMMedia.Open

Opens a multimedia device and loads a given file.

Declaration:

```
procedure Open(FileName: String);
```

Remarks:

You must specify the multimedia devise using the **SetDevice** method before you call this method. If the current device is a CD audio device, you may specify an initial track number as a *FileName* parameter.

The following example will create a *MyMedia* object, open a CD audio device and set the current position to the fifth track:

```
uses MMedia;
var
  MyMedia: TMMedia;
begin
  MyMedia.Create(0);
  MyMedia.SetDevice(MCI_CDAudio);
  MyMedia.Open('5');
  ...
end.
```

TMMedia.Pause

Pauses the open multimedia device.

Declaration:

```
procedure Pause;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Pause* method has finished. The **Notify** field determines whether *Pause* generates a MM_MCINOTIFY message.

TMMedia.Play

Plays the media loaded in the open multimedia device.

Declaration:

```
procedure Play;
```

Remarks:

The **Wait** filed determines whether control is returned to the application before the *Play* method has finished. The **Notify** field determines whether MCI generates a MM_MCINOTIFY message.

TMMedia.Previous

Moves to the beginning of the previous track of the currently loaded medium if the current position was at the beginning of a track.

Declaration:

```
procedure Previous;
```

Remarks:

If the device doesn’t use tracks, *Previous* sets the current position to the beginning of the medium, which is specified using the **SetStartPos** method.

The **Wait** field determines whether control is returned to the application before the *Previous* method has finished. The **Notify** field determines whether *Previous* generates a MM_MCINOTIFY message.

TMMedia.Rec

Begins recording from the current position specified using the **SetStartPos** method.

Declaration:

```
procedure Rec;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Rec* method has finished. The **Notify** field determines whether *Rec* generates a MM_MCINOTIFY message.

TMMedia.ResetEndPos

Disregards the ending position settings specified by **SetEndPos** method..

Declaration:

```
procedure ResetEndPos;
```

Remarks:

Forces **Play** and **Rec** methods to ignore the ending position setting.

TMMedia.ResetStartPos

Disregards the staring position settings specified by the **SetStartPos** method.

Declaration:

```
procedure ResetStartPos;
```

Remarks:

Forces **Play** and **Rec** methods to ignore the start position setting.

TMMedia.Resume

Resumes playing or recording the currently paused multimedia device.

Declaration:

```
procedure Resume;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Resume* method has finished. The **Notify** field determines whether *Resume* generates a MM_MCINOTIFY message.

TMMedia.Save

Saves the currently loaded medium to the specified file.

Declaration:

```
procedure Save(FileName: String);
```

Remarks:

Save is ignored for devices that don’t use media stored in files (videodiscs, for example).

The **Wait** field determines whether control is returned to the application before the *Save* method has finished. The **Notify** field determines whether or not *Save* generates a MM_MCINOTIFY message.

TMMedia.SetDevice

Sets a multimedia device type to open the media.

Declaration:

```
procedure SetDevice(Device: DWORD);
```

Remarks:

The valid values for *Device* are MCI_AutoSelect, MCI_AVIVideo, MCI_CDAudio, MCI_DAT, MCI_DigitalVideo, MCI_MMMovie, MCI_Other, MCI_Overlay, MCI_Scanner, MCI_Sequencer, MCI_VCR, MCI_VideoDisc, MCI_WaveAudio.

If **Device** specified as MCI_AutoSelect, the device type is determined by the file extension during the **Open** method.

A multimedia device is typically associated with an appropriate file-name extension when the device is installed. Associations are specified in the registry or **SYSTEM.INI** file. See the documentation for the specific device for instructions on how to associate file-name extensions with the device.

TMMedia.SetDisplayRect

Specifies a rectangular area in the window specified by the **SetDisplayWindow** method that is used to display output from a multimedia device.

Declaration:

```
procedure SetDisplayRect(DisplayRect: TRect);
```

Remarks:

Media that use a rectangle to display output usually perform best if the default *DisplayRect* size is used. To set *DisplayRect* to the default size, position the rectangle in the upper left corner and use (0, 0) for the lower right corner.

TMMedia.SetDisplayWindow

Assigns the display window for a multimedia device that uses a window for output.

Declaration:

```
procedure SetDisplayWindow(WndHandle: THandle);
```

Remarks:

Use the handle of any window to switch animation output into that window.

TMMedia.SetDoorClosed

Closes the media cover (if any).

Declaration:

```
procedure SetDoorClosed;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *SetDoorClose* method has finished. The **Notify** field determines whether *SetDoorClose* generates a MM_MCINOTIFY message.

TMMedia.SetDoorOpen

Opens the media cover (if any).

Declaration:

```
procedure SetDoorOpen;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *SetDoorOpen* method has finished. The **Notify** field determines whether *SetDoorOpen* generates a MM_MCINOTIFY message.

TMMedia.SetEndPos

Sets the end position within the currently loaded medium for playing or recording.

Declaration:

```
procedure SetEndPos(EndPos: Longint);
```

Remark:

EndPos value must be presented in the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.SetPos

Sets the current position within the currently loaded (opened) medium.

Declaration:

```
procedure SetPos(Position: Longint);
```

Remark:

The position value must be presented in the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.SetStartPos

Sets the start position within the currently loaded medium for playing or recording.

Declaration:

```
procedure SetStartPos(StartPos: Longint);
```

Remark:

StartPos value must be presented in the current time format, which is specified using the **SetTimeFormat** method.

TMMedia.SetTimeFormat

Sets the time format used to obtain and specify position information.

Declaration:

```
procedure SetTimeFormat(TimeFormat: DWORD);
```

Remarks:

Not all formats are supported by every device. When trying to set an unsupported format, the assignment is ignored. The current timing information is always passed as a 4-byte DWORD value.

Value	Time format .
MCI_FORMAT_MILLISECONDS	Milliseconds are stored as a 4-byte variable.
MCI_FORMAT_HMS	Hours, minutes, and seconds are packed into a 4-byte variable.
MCI_FORMAT_MSF	Minutes, seconds, and frames are packed into a 4-byte variable.
MCI_FORMAT_FRAMES	Frames are stored as a 4-byte variable.
MCI_FORMAT_SMPTE_24	24-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_25	25-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_30	30-frame SMPTE packs values in a 4-byte variable.
MCI_FORMAT_SMPTE_30DROP	30-drop-frame SMPTE packs data into a 4-byte variable.
MCI_FORMAT_BYTES	Bytes are stored as a 4-byte variable
MCI_FORMAT_SAMPLES	Samples are stored as a 4-byte integer variable
MCI_FORMAT_TMSF	Tracks, minutes, seconds, and frames are packed in the 4-byte variable.

TMMedia.Step

Moves forward or backward (depending on the sign of the *Frames* parameter) a given number of frames in the currently loaded medium.

Declaration:

```
procedure Step(Frames: Longint);
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Step* method has finished. The **Notify** field determines whether *Step* generates a MM_MCINOTIFY message.

TMMedia.Stop

Stops playing or recording.

Declaration:

```
procedure Stop;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Stop* method has finished. The **Notify** field determines whether *Stop* generates a MM_MCINOTIFY message.

TMMedia.Rewind

Sets the current position to the beginning of the medium.

Declaration:

```
procedure Rewind;
```

Remarks:

The **Wait** field determines whether control is returned to the application before the *Rewind* method has finished. The **Notify** field determines whether *Rewind* generates a MM_MCINOTIFY message.

Chapter 11

The Mouse Unit

Targets: MS-DOS, Win32 console

The Mouse unit is a programming interface for controlling a mouse pointing device in MS-DOS and Win32 console applications, CONSOLE_APP. Included is the ability to define and install your own interrupt mouse driver. This driver is called whenever a button on the mouse is pressed or released or when the mouse is moved. It is also possible to poll for mouse movement.

The benefits of having an interrupt mouse driver are that mouse events can take place at any time; one doesn't have to wait for them. Polling for mouse events must occur within some type of loop where the status of the mouse is constantly checked. There are many benefits in polling for mouse events.

The Mouse unit gives a Pascal program access to the main mouse-support functions:

- Determining the presence or absence of a mouse
- Mouse cursor positioning
- Getting information about the position and the button states
- Installing a mouse-driven Pascal interrupt handler

11.1 Mouse Unit Procedures and Functions

ClearMouseHandler procedure

Removes the user's mouse interrupt handler.

Declaration:

```
procedure ClearMouseHandler;
```

See also: **SetMouseHandler**

DoneMouse procedure

Does the following:

– removes (hides) the mouse cursor;
– frees the callback address for the mouse interrupt handler;
– frees the stack for the mouse interrupt handlers;
– resets internal variable *ButtonCount* to 0.

Declaration:

```
procedure DoneMouse;
```

See also: **InitMouse**

GetButtonCount function

Returns a number of mouse buttons.

Declaration:

```
function GetButtonCount: DWord;
```

Remark:

The *GetButtonCount* must be called after execution of **InitMouse** procedure.

GetMouseInfo procedure

Returns information about the position of the mouse cursor (*X*,*Y*) and the pressed buttons (*ButtonMask*).

Declaration:

```
procedure GetMouseInfo(var ButtonMask: Word; var X: Word; var
Y: Word);
```

See also: **MickyToText**

GetMouseX function

Returns X-position of the mouse cursor.

Declaration:

```
function GetMouseX: DWord;
```

See also: **MickyToText**, **GetMouseY**

GetMouseY function

Returns the Y-position of the mouse cursor.

Declaration:

```
function GetMouseY: DWord;
```

See also: **MickyToText**, **GetMouseX**

HideMouse procedure

Makes the mouse cursor invisible when the mouse is present.

Declaration:

```
procedure HideMouse;
```

Remarks:

This function is ignored under Win32.

Win32 Target:

The *HideMouse* procedure does nothing.

See also: **ShowMouse**

InitMouse procedure

Does the following:

- Allocates a 4K stack for the user's mouse interrupt-driven procedure.
- Determines the presence or absence of the mouse.
- If the mouse is present:

– enables the mouse cursor (calls *ShowMouse*);
– positions the cursor to the upper-left screen corner;
– reserves the callback address for the users' interrupt-driven mouse procedure (regardless of whether it will get installed or not).

Declaration:

```
procedure InitMouse;
```

See also: **DoneMouse**

LeftButtonPressed function

Returns True, if the left mouse button is currently pressed.

Declaration:

```
function LeftButtonPressed: Boolean;
```

See also: **MiddleButtonPressed, RightButtonPressed**

MickyToText function

Converts mouse coordinates into text mode coordinates.

Declaration:

```
function MickyToText(Coord: DWord): DWord;
```

See also: **TextToMicky, GetMouseX, GetMouseY, GetMouseInfo**

MiddleButtonPressed function

Returns True, if the middle mouse button is currently pressed.

Declaration:

```
function MiddleButtonPressed: Boolean;
```

See also: **LeftButtonPressed, RightButtonPressed**

RightButtonPressed function

Returns True, if the right mouse button is currently pressed.

Declaration:

```
function RightButtonPressed: Boolean;
```

See also: **LeftButtonPressed**, **MiddleButtonPressed**

SetMouseHandler procedure

Installs a Pascal interrupt-driven user's mouse handler (*Hnd*).

Declaration:

```
procedure SetMouseHandler (Mask: DWord; Procedure Hnd (Mask,
Buttons, X, Y, MovX, MovY: System.Word));
```

Remarks:

The *Mask* parameter defines the classes of the events that call the handler; its format corresponds to the function 0Ch (INT 33h). When *Hnd* is called the Mask parameter contains the mask with the event type that occurred; *Buttons* contain the mask of the currently pressed buttons; *X* and *Y* contain the cursor's absolute position, and *MovX* and *MovY* contain the relative (signed) change of the last cursor position [negative numbers mean left or down; positive mean right or up]. These values are given in mouse position units; to convert them to symbols, they should be divided by 8.

Using *SetMouseHandler* you can install several handlers with different masks without having to clear previous handlers.

```
Event Mask (events which you want sent to your handler)
bit 0 = mouse movement          (Mask = $01)
bit 1 = left button pressed     (Mask = $02)
bit 2 = left button released    (Mask = $04)
bit 3 = right button pressed    (Mask = $08)
bit 4 = right button released   (Mask = $10)
bit 5 = center button pressed   (Mask = $20)
bit 6 = center button released  (Mask = $40)
        all events:      Mask = 007fH
        disable handler: Mask = 0000H
```

Example:

```
program Test;
{$ifndef __DOS__}
{$ifndef __WIN32__}
This program can be compiled for MS-DOS and Win32 console
targets only
{$endif}
{$endif}
uses CRT, Mouse;
var
  isExit: Boolean := FALSE;
  X, Y:   DWord;
  star:   Word := $0F2A;
  space:  Word := $0F00;

procedure MyHnd(Mask, Buttons, X, Y, MovX, MovY: System.Word);
begin
  GotoXY(1, 2);
  Writeln('XPos = ', MickyToText(X),', YPos = ',
MickyToText(Y), '  ');
end;

begin
 ClrScr;
 Writeln('Press Esc to exit');
 HideCursor;
 InitMouse;
 SetMouseHandler($FFFF, MyHnd);
 repeat
  X := MickyToText(GetMouseX) + 1;
  Y := MickyToText(GetMouseY) + 1;
  if (LeftButtonPressed) and (Y > 2) and (Y < 25) then
    WriteAttr (X, Y, Star, 1);
  if (RightButtonPressed) and (Y > 2) and (Y < 25) then
    WriteAttr (X, Y, Space, 1);
  if Keypressed then
    isExit := ReadKey = #27;
 until isExit;
 ShowCursor;
 DoneMouse;
end.
```

See also: **ClearMouseHandler**

SetMousePos procedure

Positions the mouse cursor to the point (X,Y).

Declaration:

```
procedure SetMousePos(X, Y: DWord);
```

Remarks:

This function is ignored under Win32.

See also: **SetMouseRange**, **TextToMicky**

SetMouseRange procedure

Sets the mouse movement range.

Declaration:

```
procedure SetMouseRange(MinX, MinY, MaxX, MaxY: DWord);
```

Remarks:

This function ignored under Win32.

See also: **TextToMicky**

ShowMouse procedure

Makes the mouse cursor visible if the mouse is present.

Declaration:

```
procedure ShowMouse;
```

Remarks:

This function is ignored under Win32.

Win32 Target:

ShowMouse procedure do nothing.

See also: **HideMouse**

TextToMicky function

Converts text mode coordinates into mouse coordinates.

Declaration:

```
function TextToMicky(Coord: DWord): DWord;
```

See also: **MickyToText**, **SetMousePos, SetMouseRange**

Chapter 12

OS/2 API Interface Units

Targets: OS/2 only

TMT Pascal comes with set of special run-time library units, which provide an OS/2 API interface. The names of these units are listed below.

DosCall
OS2Ord
OS2PMAPI
OS2Types

For more information refer to the OS/2 Programmer's Reference by the IBM Corporation. Also, you will find the source of all OS/2 API interface units in the **\TMTPL\SOURCE\OS2** directory.

Chapter 13

The Printer Unit

Targets: MS-DOS, OS/2, Win32

The Printer unit declares a text file called *Lst* and associates it with the LPT1 device.

```
var Lst : Text;
```

When you use the Printer unit, you don't need to declare, assign, open, and close a text file yourself if you use your printer from within a program.

Example:

```
uses Printer;
begin
  WriteLn(Lst, 'Your printer works properly!');
end.
```

Chapter 14

The Strings Unit

Targets: MS-DOS, OS/2, Win32

The Strings unit provides procedures and functions to manipulate strings, encrypt and decrypt data, perform search and replace on strings and more. While the majority of the functions and procedures on Strings can be written with TMT Pascal, the ones in the Strings unit are all written in assembly language and thus have must faster execution times.

14.1 Strings Unit Overloaded Operators

The Strings unit overloads the following operators

```
overload  +:= = StrAppend
overload  +:= = StrAppendC
overload   *  = Dup_SI
overload   *  = Dup_CI
```

14.2 Strings Unit Procedures and Functions

Align function

Pads the argument up to width with spaces.

Declaration:

```
function Align(Str: String; Width: LongInt): String;
```

Remarks:

Spaces are on the right if width > 0, and on the left otherwise.

AnsiCompareStr function

Compares strings based on the current Windows locale. It is case sensitive.

Declaration:

```
function AnsiCompareStr(S1, S2: String): Longint;
```

Remarks:

AnsiCompareStr compares *S1* to *S2*, with case sensitivity. The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

Win32 target:

AnsiCompareStr uses the current Windows locale.

AnsiLowerCase function

Converts a string to lower case.

Declaration:

```
function AnsiLowerCase(S: String): String;
```

Win32 target:

AnsiLowerCase uses the current Windows locale.

AnsiCompareText function

Compares strings based on the current Windows locale and is not case sensitive.

Declaration:

```
function AnsiCompareText(S1, S2: String): Longint;
```

Remarks:

AnsiCompareText compares *S1* to *S2*, without case sensitivity. The compare operation is controlled by the current Windows locale. The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

Win32 target:

AnsiCompareText uses the current Windows locale.

AnsiStrComp function

Compares null-terminated character strings case sensitively.

Declaration:

function AnsiStrComp(S1, S2: PChar): Longint;

Remarks:

AnsiStrComp compares *S1* to *S2*, with case sensitivity. The return value is:

Condition	Return Value
S1 > S2	> 0
S1 < S2	< 0
S1 = S2	= 0

Win32 target:

AnsiStrComp uses the current Windows locale.

AnsiStrIComp function

Compares null terminated character strings case insensitively.

Declaration:

```
function AnsiIStrComp(S1, S2: PChar): Longint;
```

Remarks:

AnsiStrIComp compares *S1* to *S2*, without case sensitivity. The return value is:

Condition	Return Value
S1 > S2	> 0
S1 < S2	< 0
S1 = S2	= 0

Win32 target:

AnsiStrIComp uses the current Windows locale.

AnsiStrLComp function

Compares the first *MaxLen* bytes of two character sequences. It is case sensitive.

Declaration:

```
function AnsiStrLComp(S1, S2: PChar; MaxLen: DWord): Longint;
```

Remarks:

AnsiStrLComp compares *S1* to *S2*, with case sensitivity. If *S1* or *S2* is longer than *MaxLen* bytes, *AnsiStrLComp* only compares the first *MaxLen* bytes. The return value is:

Condition	Return Value
S1 > S2	> 0
S1 < S2	< 0
S1 = S2	= 0

Win32 target:

AnsiStrLComp uses the current Windows locale.

AnsiStrLIComp function

Compares two strings, case-insensitively, up to the first *MaxLen* bytes.

Declaration:

```
function AnsiStrLIComp(S1, S2: PChar; MaxLen: DWord): Longint;
```

Remarks:

AnsiStrLIComp compares *S1* to *S2*, without case sensitivity. If *S1* or *S2* is longer than *MaxLen* characters, *AnsiStrLIComp* only compares up to the first *MaxLen* characters. The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

Win32 target:

AnsiStrLIComp uses the current Windows locale.

AnsiStrLower function

Converts all characters in a null-terminated character sequence to lower case.

Declaration:

```
function AnsiStrLower(Str: PChar): PChar;
```

Remarks:

AnsiStrLower returns a null-terminated character sequence where all characters are lower case.

Win32 target:

AnsiStrLower uses the current Windows locale.

AnsiStrUpper function

Converts all characters in a null-terminated character sequence to upper case.

Declaration:

```
function AnsiStrUpper(Str: PChar): PChar;
```

Remarks:

AnsiStrUpper returns a null-terminated character sequence where all characters are upper case.

Win32 target:

AnsiStrUpper uses the current Windows locale.

AnsiUpperCase function

Converts a string to upper case.

Declaration:

```
function AnsiUpperCase(S: String): String;
```

Win32 target:
AnsiUpperCase uses the current Windows locale.

AppendPathDelimiter function

Appends a path delimiter (\) to the specified path-string.

Declaration:

```
function AppendPathDelimiter(const S: String): String;
```

Example:

```
AppendPathDelimiter('C:\TMTPL');          // returns
'C:\TMTPL\'

AppendPathDelimiter('C:\TMTPL\SAMPLES\'); // returns
'C:\TMTPL\SAMPLES\'
```

Bin function

Converts an argument into a standard binary 32-character string.

Declaration:

```
function Bin(n: Longint): String[32];
```

Examples:

```
Bin($000000FF) = '00000000000000000000000011111111'

Bin(126479829) = '00000111100010011110110111010101'
```

Dup_CI function

Creates a string by repeating a character *N* times.

Declaration:

```
function Dup_CI(C: char; N: Longint): String;
```

Dup_SI function

Copies the string *N* times.

Declaration:

```
function Dup_SI(C: String; N: Longint): String;
```

Fix function

Converts an argument into a fixed point string.

Declaration:

```
function Fix(X: Extended; W, Pr: LongInt): String[15];
```

Remark:

Pr is the number of digits after the decimal point.

FloatToStr function

Converts a floating point value to a string.

Declaration:

```
function FloatToStr(Value: Extended): String;
```

Remark:

FloatToStr converts the floating-point value given by *Value* to its string representation.

Fls function

Converts its argument into a standard floating-point string.

Declaration:

```
function Fls(X: Extended): String;
```

Examples:

```
Fls(1E+1) = 10.
Fls(2.855E-2) = 0.02855
```

Flt function

Converts its argument into a formatted floating-point string.

Declaration:

```
function Flt(X: Extended; W: Longint): String;
```

Hex function

Converts its argument into a hex string.

Declaration:

```
function Hex(n: Longint): String[12];
```

HexVal function

Converts a hex string into a Longint.

Declaration:

```
function HexVal(const S: String): Longint;
```

IntToBin function

Returns the bin representation of an integer.

Declaration:

```
function IntToBin(Value: Longint; Digits: DWord): String;
```

Remarks:

The *IntToBin* function converts a number into a string containing the number's binary (base 2) representation. *Value* is the number to convert. *Digits* indicates the number of binary digits to return.

IntToHex function

Returns the hex representation of an integer.

Declaration:

```
function IntToHex(Value: Longint; Digits: DWord): String;
```

Remarks:

The *IntToHex* function converts a number into a string containing the number's hexadecimal (base 16) representation. *Value* is the number to convert. *Digits* indicates the number of hexadecimal digits to return.

IntToStr function

Converts an integer to a string.

Declaration:

```
function IntToStr(Value: Longint): String;
```

Remarks:

The *IntToStr* function converts an integer into a string containing the decimal representation of that number. *Value* is the number to convert.

IsDelimiter function

Returns TRUE if a specified character in a string matches one of a set of delimiters.

Declaration:

```
function IsDelimiter(const Delimiters, S: String; IndexPos:
Longint): Boolean;
```

IsPathDelimiter function

Returns TRUE if the byte at position *IndexPos* of a string is the backslash character (\).

Declaration:

```
function IsPathDelimiter(const S: String; IndexPos: Longint):
Boolean;
```

IsValidIdent function

Tests for a valid Pascal identifier.

Declaration:

```
function IsValidIdent(const Ident: String): Boolean;
```

Remarks:

IsValidIdent returns TRUE if the given string is a valid Pascal identifier. Identifiers are tokens that have a special meaning in TMT Pascal. Identifiers begin with a letter (A-Z or a-z) or underscore, and may contain letters, underscores, and digits (0-9).

LastDelimiter function

Returns the byte index in *S* of the last character that matches any character in the *Delimiters* string.

Declaration:

```
function LastDelimiter(const Delimiters, S: String): Longint;
```

Example:

```
LastDelimiter('\.', 'C:\TMTPL\BIN\TMTPC.EXE');  // returns 19
```

LowerCase function

Converts an ASCII string to lowercase.

Declaration:

```
function LowerCase(Str: String): String;
```

Remarks:

LowerCase returns a string with the same text as the string passed in *Str*, but with all letters converted to lowercase. The conversion affects only 7-bit ASCII characters between 'A' and 'Z'.

QuotedStr function

Returns the quoted version of a string.

Declaration:

```
function QuotedStr(const S: String): String;
```

Remarks:

A quote character (') will be inserted at the beginning and end of *S*, and each single quote character in the string is repeated.

StrAppend procedure

Appends *Src* at the end of *Dst*.

Declaration:
```
procedure StrAppend(var Dst: String; const Src: String);
```

StrAppendC procedure

Appends character *Src* to the end of string *Dst*.

Declaration:
```
procedure StrAppendC(var Dst: String; Src: Char);
```

StrCat function

StrCat appends *Src* to the end of *Dst* and returns the concatenated string of PChar type.

Declaration:
```
function StrCat(Dst, Src: PChar): PChar;
```

StrComp function

StrComp compares S1 to S2

Declaration:
```
function StrComp(S1, S2 : PChar): Longint;
```

Remarks:
The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

StrCopy function

Copies *Src* string to the *Dst*.

Declaration:
```
function StrCopy(Dst, Src: PChar): PChar;
```

StrDispose procedure

StrDispose disposes of a string allocated by *StrNew*.

Declaration:
```
procedure StrDispose(Str: PChar);
```

StrECopy function

StrECopy copies *Src* to *Dst* and returns a pointer to the null character at the end of *Dst*.

Declaration:

```
function StrECopy(Dst, Src: PChar): PChar;
```

StrEnd function

StrEnd returns a pointer to the end of a null terminated string.

Declaration:

```
function StrEnd(Str: PChar): PChar;
```

Remarks:

The *StrEnd* function returns a pointer to the null character at the end of *Str*.

StrLCat function

StrLCat appends a specified maximum number of characters to specified string.

Declaration:

```
function StrLCat(Dst, Src: PChar; MaxLen: Longint): PChar;
```

StrIComp function

StrIComp compares *S1* to *S2* without case sensitivity.

Declaration:

```
function StrIComp(S1, S2:PChar): Longint;
```

Remarks:

The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

StrLComp function

StrLComp compares a specified maximum number of characters (*MaxLen*) in two strings of PChar type.

Declaration:

```
function StrLComp(Str1, Str2: PChar; MaxLen: Longint): Longint;
```

Remarks:

The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

StrLCopy function

StrLCopy copies a specified maximum number of characters from *Src* to *Dst*.

Declaration:

```
function StrLCopy(Dst, Src: PChar; MaxLen: Longint): PChar;
```

Remarks:

The *StrLCopy* function copies *MaxLen* characters from *Src* to *Dst* and returns pointer on *Dst*.

StrLen function

StrLen returns number of character in a string excluding the null terminator.

Declaration:

```
function StrLen(Str: PChar): LongInt;
```

Remarks:

The *StrLen* function calculates the number of characters in *Str*. The null terminator is not included.

StrLIComp function

StrLIComp compares two strings up to a specified maximum number (*MaxLen*) of characters, not case sensitive.

Declaration:

```
function StrLIComp(Str1, Str2: PChar; MaxLen: Longint):
Longint;
```

Remarks:

The return value is:

Condition	*Return Value*
S1 > S2	*> 0*
S1 < S2	*< 0*
S1 = S2	*= 0*

StrLower function

The *StrLower* function returns a string in lower case.

Declaration:

```
function StrLower(Str: PChar): PChar;
```

StrMove function

StrMove copies a specified number of characters to the null terminated string.

Declaration:

```
function StrMove(Dst, Src: PChar; Count: LongInt): PChar;
```

Remak:

The *StrMove* function copies *Count* characters from *Src* to *Dst* and returns pointer on *Dst*.

StrNew function

The *StrNew* function allocates space on and copies a string to the heap; then it returns a pointer to the string.

Declaration:

```
function StrNew(Str: PChar): PChar;
```

StrPas function

StrPas converts a null-terminated (PChar) string Str to a Pascal-style string.

Declaration:

```
function StrPas(Str: PChar): String;
```

StrPCopy function

StrPCopy copies a Pascal-style string *Src* to a null-terminated (PChar) string *Dst*.

Declaration:

```
function StrPCopy(Dst: PChar; const Src: String): PChar;
```

StrPos function

Finds first entry of *Str2* in *Str1*.

Declaration:

```
function StrPos(Str1, Str2: PChar): PChar;
```

Remarks:

StrPos returns a pointer to the first entry of *Str2* in *Str1*.

StrRScan function

StrRScan returns a pointer to the last occurrence of *Chr* in *Str*.

Declaration:

```
function StrRScan(Str: PChar; Chr: Char): PChar;
```

Remarks:

The *StrRScan* function returns a pointer to the first entry of *Chr* in *Str* from end of *Str*. If *Chr* is not found *Str*, *StrRScan* returns *nil*.

StrScan function

StrScan returns a pointer to first occurrence of a specified character in a string of PChar type.

Declaration:

function StrScan(Str: PChar; Chr: Char): PChar;

Remark:

If *Chr* does not found in *Str*, *StrScan* returns **nil**.

StrToInt function

Converts a string representing a long integer to a number.

Declaration:

```
function StrToInt(const S: String): Longint;
```

Examples:

```
StrToInt('11')  = 11;
StrToInt('$FF') = 256;
```

StrToIntDef function

Converts a string representing a long integer to a number.

Declaration:

```
function StrToIntDef(const S: String; Default: Longint):
Longint;
```

Remark:

If *S* is not a valid number, *StrToIntDef* returns the number passed in *Default*.

StrUpper function

The *StrUpper* function returns a string in upper case.

Declaration:

```
function StrUpper(Str: PChar): PChar;
```

Trim Function

Trims a string of leading and trailing spaces and control characters.

Declaration:

```
function Trim(const S: String): String;
```

TrimLeft Function

Trims a string of leading spaces and control characters.

Declaration:

```
function TrimLeft(const S: String): String;
```

TrimRight Function

Trims a string of trailing spaces and control characters.

Declaration:

```
function TrimRight(const S: String): String;
```

Uns function

Converts its argument into a string unsigned integer.

Declaration:

```
function Uns(N: DWord): String[12];
```

UpperCase function

Converts an ASCII string to uppercase.

Declaration:

```
function UpperCase(Str: String): String;
```

Remarks:

UpperrCase returns a string with the same text as the string passed in *Str*, but with all letters converted to uppercase. The conversion affects only 7-bit ASCII characters between ‘a’ and ‘z’.

Whl function

Converts the argument into signed integer.

Declaration:

```
function Whl(N: LongInt): String[12];
```

Chapter 15

The System Unit

Targets: MS-DOS, OS/2, Win32

The System unit provides low level support routines such as low level file I/O, heap management, a random number generator, string handling and more. The system unit is automatically linked with every program and need not be stated in a **Uses** statement.

Keep in mind, that the System unit always uses the FPU for floating point arithmetic.

15.1 System Unit Constants and Variables

_environ variable

Targets: MS-DOS, Win32

The *_environ* variable contains the environment address.

Declaration:

```
var _environ: Pointer;
```

_psp variable

Targets: MS-DOS only

The *_psp* variable contains the logical 32-bit address of the PSP of the program.

Declaration:

```
var _psp: Pointer;
```

_zero variable

Targets: MS-DOS, OS/2, Win32

The *_zero* variable is always 0 and is provided for compatibility with old versions of TMT Pascal compiler.

Declaration:

```
var _zero: DWord := 0;
```

CmdLine variable

Targets: Win32 only

Points to the command-line arguments specified when the application is invoked.

Declaration:

```
var CmdLine: PChar;
```

Remarks:

The *CmdLine* variable contains a pointer to a null-terminated string that contains the command-line arguments specified when the application was started. Use **ParamStr** to access individual arguments. In a library (DLL), CmdLine is always **nil**.

CmdShow variable

Targets: Win32 only

Passed to the Windows API *ShowWindow* routine.

Declaration:

```
var CmdShow: DWord;
```

Remarks:

In a program, the *CmdShow* variable contains the parameter value that Windows expects to be passed to *ShowWindow* when the application creates its main window. In a library (DLL), *CmdShow* is always zero.

ErrorAddr variable

ExitProc, *ExitCode*, and *ErrorAddr* variables are used to implement exit procedures.

The *ExitProc* pointer variable allows one to install an exit procedure. The exit procedure always gets called as part of a program's termination.

An exit procedure takes no parameters and must be compiled with a far procedure directive to force it to use the far call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. The procedures on the exit chain get executed in reverse order of installation.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure.

The first statement in your exit procedure must reinstall the saved value of *ExitProc*.

An exit procedure may learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

- In case of normal termination, *ExitCode* is 0 and *ErrorAddr* is nil.
- In case of termination through a call to **Halt**, *ExitCode* contains the value passed to Halt and *ErrorAddr* is nil.
- In case of termination due to a run-time error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the run-time library) closes the Input and Output files. If *ErrorAddr* is not nil, it outputs a run-time error message.

ExeName variable

Targets: MS-DOS, Win32

The *ExeName* variable contains a pointer to a null-terminated string that contains the name of the program (executable file).

Declaration:

```
var ExeName: PChar;
```

ExeSize variable

Targets: MS-DOS only

The *ExeSize* variable contains the size of the program (executable file).

Declaration:

```
var ExeSize: DWord;
```

ExitCode variable

ExitProc, *ExitCode*, and *ErrorAddr* variables are used to implement exit procedures.

The *ExitProc* pointer variable allows one to install an exit procedure. The exit procedure always gets called as part of a program's termination.

An exit procedure takes no parameters and must be compiled with a far procedure directive to force it to use the far call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. The procedures on the exit chain get executed in reverse order of installation.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure.

The first statement in your exit procedure must reinstall the saved value of *ExitProc*.

An exit procedure may learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

- In case of normal termination, *ExitCode* is 0 and *ErrorAddr* is nil.
- In case of termination through a call to **Halt**, *ExitCode* contains the value passed to Halt and *ErrorAddr* is nil.
- In case of termination due to a run-time error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the run-time library) closes the Input and Output files. If *ErrorAddr* is not nil, it outputs a run-time error message.

ExitProc variable

ExitProc, *ExitCode*, and *ErrorAddr* variables are used to implement exit procedures.

The *ExitProc* pointer variable allows one to install an exit procedure. The exit procedure always gets called as part of a program's termination.

An exit procedure takes no parameters and must be compiled with a far procedure directive to force it to use the far call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. The procedures on the exit chain get executed in reverse order of installation.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure.

The first statement in your exit procedure must reinstall the saved value of *ExitProc*.

An exit procedure may learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

- In case of normal termination, *ExitCode* is 0 and *ErrorAddr* is nil.
- In case of termination through a call to **Halt**, *ExitCode* contains the value passed to Halt and *ErrorAddr* is nil.
- In case of termination due to a run-time error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the run-time library) closes the Input and Output files. If *ErrorAddr* is not nil, it outputs a run-time error message.

FarPointer type

Declaration:

MS-DOS target:

```
type FarPointer = record
   Ofs: Pointer;
   Seg: Word
end;
```

OS2 and Win32 targets:

```
type FarPointer = Pointer;
```

Remark:

FarPointer type is used by the **SetIntVec** and **GetIntVec** procedures.

FileMode variable

The *FileMode* variable determines the access code to pass to the Operating System when typed and untyped files are opened using the **Reset** procedure. By default *FileMode* is 2. The following modes may be assigned to *FileMode*:

0	Read only
1	Write only
2	Read/Write

HeapHandle variable

Targets: Win32 only

The *HeapHandle* variable contains a handle of the global memory heap used by TMT Pascal.

Declaration:

```
var HeapHandle: DWord;
```

hInstance variable

Targets: Win32 only

HInstance is the handle provided by Windows operating system for an applicaiton or library.

Declaration:

```
var hInstance: Longint;
```

Remarks:

HInstance contains the instance handle of the application or library as provided by the Windows operating system environment. *HInstance* is not global, inheritable, or duplicative, and it cannot be used by another process.

InOutRes variable

The built-in I/O routines use *InOutRes* to store the value that the next call to the **IOResult** standard function will return.

Input variable

Input and *Output* are the standard I/O files required by every Pascal implementation.

IsConsole variable

IsConsole is a boolean varible which is TRUE if the program was compiled as a console application.

Declaration:

```
  var IsConsole: Boolean;
```

Remarks:

IsConsole is FALSE if the program was compiled as GUI (Win32) or PM (OS/2) application.

MS-DOS target:

IsConsole variable is always TRUE.

IsLibrary variable

IsLibrary is a boolean varible which is TRUE if the application was compiled as a dynamic-link library (DLL) and FALSE if the application was compiled as an executable (EXE).

Declaration:

```
  var IsLibrary: Boolean;
```

MS-DOS target:

IsLibrary variable is always FALSE.

LongRec type

Declaration:

```
type
  LongRec = record
    Lo, Hi: Word;
  end;
```

MaxCardinal constant

The maximum value of the Cardinal data type.

Declaration:

```
const MaxCardinal = High(Cardinal);
```

MaxCardinal represents the highest value in the range of the Cardinal data type (4294967295).

MaxDWord constant

The maximum value of the DWord data type.

Declaration:

```
const MaxDWord = High(DWord);
```

MaxDWord represents the highest value in the range of the DWord data type (4294967295).

MaxInt constant

The maximum value of the Integer data type.

Declaration:

```
const MaxInt = High(Integer);
```

MaxInt represents the highest value in the range of the Integer data type (32767).

MaxLongint constant

The maximum value of the Longint data type.

Declaration:

```
const MaxLongint = High(Longint);
const MaxLong = High(Longint);
```

MaxLongint represents the highest value in the range of the Integer data type (2147483647).

MaxWord constant

The maximum value of the Word data type.

Declaration:

```
const MaxWord = High(Word);
```

MaxWord represents the highest value in the range of the Integer data type (65535).

Output variable

Input and *Output* are the standard I/O files required by every Pascal implementation.

RandSeed variable

RandSeed stores the built-in random number generator's seed. By assigning a specific value to *RandSeed*, the Random function can be made to generate repeatedly a specific sequence of random numbers.

This is useful for applications that deal with data encryption, statistics, and simulations.

StdErrorHandle variable

Targets: Win32 only

The *StdErrorHandle* variable contains a handle for the standard error device of the console's active screen buffer.

Declaration:

```
var StdErrorHandle: DWord;
```

Remarks:

TMT Pascal uses GetStdHandle(STD_ERROR_HANDLE) Windows API function to initialize *StdErrorHandle* variable. In a GUI application or library (DLL), *StdErrorHandle* is always 0.

StdInputHandle variable

Targets: Win32 only

The *StdInputHandle* variable contains a handle for the standard input device of the console's active screen buffer.

Declaration:

```
var StdInputHandle: DWord;
```

Remarks:

TMT Pascal uses the GetStdHandle(STD_INPUT_HANDLE) Windows API function to initialize the *StdInputHandle* variable. In a GUI application or library (DLL), the *StdInputHandle* is always 0.

StdOutputHandle variable

Targets: Win32 only

The *StdOutputHandle* variable contains a handle for the standard output device of the console's active screen buffer.

Declaration:

```
var StdOutputHandle: DWord;
```

Remarks:

TMT Pascal uses the GetStdHandle(STD_OUTPUT_HANDLE) Windows API function to initialize the *StdOutputHandle* variable. In a GUI application or library (DLL), *StdOutputHandle* is always 0.

Test8086 variable

Identifies the type of 80x86 processor (CPU) the system contains.

Declaration:

```
const
  Test8086: DWord = 2;
```

Remarks:

Test8086 always is 2 (Intel 80386 or higher). Use **CPU_getProcessorType** function to get exact processor type (see **The ZenTimer Unit** for more info).

Test8087 variable

Identifies the type of 80x87 processor (FPU) the system contains.

Declaration:

```
const
  Test8087: DWord = 3;
```

OS2 and Win32 targets:

Test8086 always is 3 (Intel 80386 or higher).

MS-DOS target:

Test8087 is 3 for 387 or later or 0 if FPU is not detected.

15.2 System Unit Procedures and Functions

Abs function

Returns the absolute value of the argument.

Declaration:

```
function Abs(X);
```

Remarks:

Use *Abs* to determine the absolute value of an integer or real type argument.

Example:

```
Var
  x : Integer;
  r : Single;
Begin
  x := Abs(-50);    // 50
  r := Abs(-60.34); // 60.34
End.
```

Addr function

Returns the address of a specified object.

Declaration:

```
function Addr(X): Pointer;
```

Remarks:

Use *Addr* to determine the address of an object. Using *Addr* is the same as using the @ operator.

Example:

```
Var
  b : Byte;
  p : Pointer
Begin
  p := Addr(b);  // Same as p := @b;
End.
```

Append procedure

Opens an existing file for appending.

Declaration:

```
procedure Append(var F:Text);
```

Append works only if the file exists. It sets the file pointer to the end of the file. If *F* is open when *Append* is called then *F* is first closed and then reopened. If one assigns an empty file name to *F*, output will be directed to the standard output file. Errors are returned through **IOResult** if `{$I-}.` If there are no errors then **IOResult** returns zero.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only
{$endif}
var
  Fi:Text;
begin
  Assign(Fi, 'THE_FILE.TXT');
  Append(Fi);
  WriteLn(Fi, 'Just testing Append.');
  Close(Fi);
end.
```

ArcTan function

Returns the arctangent of the argument.

Declaration:

```
function ArcTan(X: Real): Real;
```

Remarks:

System unit does not provide a *Tan* function, but tangents can be calculated with the expression: **Sin**(x) / **Cos**(x)

Assign procedure

Assigns the name of an external file to a file variable.

Declaration:

```
procedure Assign(var F; FileName; String);
```

Remarks:

After a call to *Assign*, all file operations on *F* are associated with *FileName*. may consist of a drive and directory specification. If no drive or directory is specified then the current directory is used. If *FileName* is empty then the standard output defined by the operating system is used. *Assign* should not be used on an open file.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only
{$endif}
var
  FiText:Text;
begin
  Assign(FiText, 'TEXTFILE.TXT');
  Rewrite(FiText);
  WriteLn(FiText, 'TextFile is now written to!');
  Close(FiText);
end.
```

Assigned function

Tests to determine if a pointer or procedural variable is **nil**.

Declaration:

```
function Assigned(var P): Boolean;
```

P must be a variable reference of a pointer or procedural type. `Assigned(P)` corresponds to the test *P*<> **nil** for a pointer variable, and @*P* <> **nil** for a procedural variable. Returns True if *P* is **nil**, False otherwise.

BlockRead procedure

Reads one or more records into a variable.

Declaration:

```
procedure BlockRead(var F: File; var Buf; Count: Longint [; var
Result: Longint]);
```

where:

```
 F        untyped file variable
 Buf      any variable
 Count    an expression of type Word
 Result   a variable of type Word
```

Remarks:

BlockRead works only on open files and advances the current file pointer. **IOResult** returns an error code if an error occurred, otherwise **IOResult** is set to zero. Also note that *BlockRead* is not limited to 65,535 (64K) bytes while reading from a file, i.e. *Count* may be >> 64K.

BlockWrite procedure

Writes one or more records from a variable.

Declaration:

```
procedure BlockWrite(var F: File; var Buf; Count: Longint [;
var Result: Longint]);
```

where:

```
 F        untyped file variable
 Buf      any variable
 Count    an expression of type Word
 Result   a variable of type Word
```

Remarks:

BlockWrite works only on open files and advances the current file pointer. **IOResult** returns an error code if an error occurred, otherwise **IOResult** is set to zero. Also note that *BlockWrite* is not limited to 65,535 (64K) bytes while reading from a file, i.e. *Count* may be >> 64K.

Break procedure

Terminates a **for**, **while**, or **repeat** statement.

Declaration:

```
procedure Break;
```

Remarks:

Causes the innermost enclosing for, while, or repeat statement to be exited immediately.

Is similar to a *Goto* statement addressing a label just after the end of the innermost enclosing repetitive statement.

ChDir procedure

Changes the current directory.

Declaration:

```
procedure ChDir(S: String);
```

Remarks:

With `{$I-}`, **IOResult** returns an error code if an error occurred, otherwise **IOResult** is set to zero. The current directory is changed to the path specified by *S*. If *S* specifies a drive letter, the current drive is also changed.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only
{$endif}
begin
{$I- }
  ChDir('MyDir');
{$I+ }
  if IOResult <> 0 then
    WriteLn('Directory does not exist');
end.
```

Chr function

Returns a character with a specified ordinal number.

Declaration:

```
function Chr(X: Byte): Char;
```

Remarks:

Use *Chr* to generate a character by specifying an integer type number.

Close procedure

Closes an open file.

Declaration:

```
procedure Close(var F);
```

Remarks:

F is a file variable of any file type previously opened with **Reset**, **ReWrite** or **Append**. The external file associated with *F* is completely updated and then closed, freeing its file handle for reuse.

With `{$I-}`, **IOResult** returns zero if the operation was successful; otherwise, it returns a nonzero error code.

Example:

```
var
  Fi: Text;
begin
  Assign(Fi, 'C:\DATAFILE.DOC');
  ReWrite(Fi);
  WriteLn(Fi, 'A New File.');
  Close(Fi);  // Close and flush buffer to disk
end.
```

Concat function

Concatenates a sequence of strings.

Declaration:

```
function Concat(s1 [, s2,..., sn]: String): String;
```

Remarks:

The *ConCat* function concatenates up to 255 characters. Additional characters are ignored. For instance, the following statements return the same results:

```
s:= ConCat(‘1234’,’567890’);  // ‘1234567890’

s:= ‘1234’ + ‘567890’;        // ‘1234567890’
```

Continue procedure

Continues a for, while, or repeat statement.

Declaration:

```
procedure Continue;
```

Remarks:

Causes the innermost enclosing for, while, or repeat statement to immediately proceed with the next iteration.

The compiler will report an error if a call to *Continue* is not enclosed by a for, while, or repeat statement.

Copy function

Returns a substring of a string.

Declaration:

```
function Copy(Str: String; StartPos, Count: Longint):String;
```

Remarks:

If *StartPos* is greater than the length of *Str* than an empty string is returned.

Example:

```
S := Copy(‘Hello World’,7,5);           // ‘World’
S := Copy(‘Use Protected Mode!’,45,10); // “
```

Cos function

Returns the cosine of the argument (*X* is an angle, in radians).

Declaration:

```
function Cos(X: Real): Real;
```

Dec procedure

Decrements an ordinal type variable either by one or by a specified value.

Declaration:

```
procedure Dec(Var X [; Value:Longint]);
```

Remarks:

```
Dec(Int,2);  // Int  := Int - 2;
Dec(Long);   // Long := Long - 1;
```

Delete procedure

Deletes a substring from a string.

Declaration:

```
procedure Delete(var S: String; Index, Count: Longint);
```

Remarks:

S is a string-type variable. **Index** and **Count** are integer-type expressions. *Delete* deletes *Count* characters from *S* starting at the *Index* position. If *Index* is larger than the length of *S*, no characters are deleted. If *Count* specifies more characters than remain starting after the *Index*th position, the remainder of the string is deleted.

Example:

```
Delete(‘AAABBBCCC’,4,3);  // ‘AAACCC’
Delete(‘123456789’,22,5); // ‘123456789’
```

Dispose procedure

Disposes of a dynamic variable.

Declaration:

```
procedure Dispose(var P: Pointer [ , Destructor ]);
```

Remarks:

P points to a typed variable that was previously assigned either by calling New or by an assignment. *Dispose* returns the allocated memory, pointed to by *P*, back to heap . If *P* does not reference heap, a run time error occurs. References to *P^* after a call to *Dispose* results in undefined data.

Note that dynamic variables are not limited to 65,535 (64K) bytes. For more information about heap, refer to the Heap Management chapter.

Example:

```
type
  StrAry  = Array[0..$200] of Char;
var
  StrPtr  : ^StrAry;
begin
  New(StrPtr);
  if StrPtr = nil then
  Halt(1);
  FillChar(StrPtr,$200,'0');
  Dispose(StrPtr);  // Release back to Heap
end.
```

Eof function

Returns the end-of-file status.

Declaration:

```
function Eof(var F): Boolean;          // for Typed or untyped
files
```

```
function Eof[(var F: Text)]: Boolean;  // for Text files
```

Remarks:

If an error occurred, **IOResult** returns an error code, otherwise **IOResult** is set to zero.

Eoln function

Returns the end-of-line status of a text file.

Declaration:

```
function Eoln[(var F: Text) ]: Boolean;
```

Remarks:

If an error occurred, **IOResult** returns an error code, otherwise **IOResult** is set to zero.

Erase procedure

Erases an external file.

Declaration:

```
procedure Erase(var F);
```

Remarks:

Erase must not be used on an open file. **IOResult** returns zero if *Erase* was successful, otherwise **IOResult** contains an error number.

Exit procedure

Exits immediately from the current block.

Declaration:

```
procedure Exit;
```

While in a procedure or function, *Exit* causes the subroutine to return to the parent routine. *Exit* terminates the program if called while in the main routine.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
uses Crt;
procedure ChgColor(ForeGround,BackGround:Byte);
begin
  if not IsColor then Exit;
  SetColor(ForeGround);
  TextBackGround(BackGround);
end;

begin
  ChgColor(Green, Black);
  WriteLn('Hellow Green World!');
end.
```

Exclude procedure

Excludes an element in a set.

Declaration:

```
procedure Exclude(var S: set of T;I:T);
```

Remarks:

S is a set type variable, and *I* is an expression of a type compatible with the base type of *S*. The element given by *I* is excluded in the set given by *S*.

Exp function

Returns the exponential of the argument.

Declaration:

```
function Exp(X: Real): Real;
```

Remarks:

Exp raises the value e to the power of *X*, where e is the base of the natural logarithm.

FilePos function

Returns the current file position of a file.

Declaration:

```
function FilePos(var F): Longint;
```

Remarks:

FilePos returns zero if the file pointer is at the top of the file and returns the file size if at the end of the file. *FilePos* works only on open files. If an error occurs, **IOResult** contains the error code, otherwise it is set to zero.

FileSize function

Returns the current size of a file.

Declaration:

```
function FileSize(var F): Longint;
```

Remarks:

FileSize returns zero if the file is empty, otherwise it returns the number of components in the file. *FileSize* works only on open files. If an error occurs, **IOResult** contains the error code, otherwise it is set to zero.

FillChar procedure

Fills a specified number (*Count)* of contiguous bytes with a specified value (can be type Byte or Char).

Declaration:

```
procedure FillChar(var X; Count: Longint; Ch: Char);
```

Remarks:

FillChar does not perform range checking and stores *Ch* as contiguous bytes starting from *X* for *Count* length. If used on strings types, set the length byte after calling *FillChar*.

Note that *Count* is not limited to 65,535 (64K) bytes.

Flush procedure

Flushes the buffer of a text file open for output.

Declaration:

```
procedure Flush(var F: Text);
```

Remarks:

Flush only works on files opened for output. Use **ReWrite** or **Append** to open a file for output. **IOResult** contains an error code if an error occurred, otherwise it is set to zero.

Calling **Close** flushes buffers to the disk before closing the file.

Frac function

Returns the fractional part of the argument.

Declaration:

```
function Frac(X: Real): Real;
```

Remarks:

Frac simply subtracts the integer part of *X* from *X*. `Frac(X) = X - Int(X)`.

FreeMem procedure

Disposes of a dynamic variable of a given size.

Declaration:

```
procedure FreeMem(var P: Pointer; Size: Longint);
```

Remarks:

P points to any type variable that was previously assigned either by calling **GetMem** or by an assignment. *FreeMem* returns the allocated memory, pointed to by *P*, back to heap. *Size* must match the size of the variable that was allocated by **GetMem**. If *P* does not reference heap, a run time error occurs.

The Boolean variable HeapCheck determines whether TMT Pascal tracks the size of dynamic variables. When True, an error occurs if *FreeMem* is called to deallocate a variable with thewrong size. This is particularly useful while debugging. References to *P*^ after a call to *FreeMem* result in undefined data. For more information about heap, refer to the Heap

GetDir procedure

Returns the current directory of a specified drive.

Declaration:

```
procedure GetDir(D: Byte; var S: String);
```

where D is set to:

0	Default
1	Drive A
2	Drive B
3	Drive C

Remarks:

IOResult returns the error code if an error occurred, otherwise it is set to zero.

GetCurDir performs the same function as *GetDir* but it takes a null-terminated string as an argument instead of a Pascal-style string.

GetMem procedure

Creates a dynamic variable of the specified size and puts the address of the block in a pointer variable.

Declaration:

```
procedure GetMem(var P: Pointer; Size: Longint);
```

Remarks:

To reference the new variable, use *P*^. If *GetMem* finds that there is an insufficient amount of free heap, a runtime error is generated. The boolean variable HeapCheck determines whether TMT Pascal tracks the size of heap variables. This size is for debugging purposes only and is stored in the four bytes before the start of the variable. For more information about heap allocation see the Heap Management chapter.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}

uses Crt;
var
  ScrnImage : Pointer;
  Size      : Longint;
begin
  Size := SaveScreenSize(10, 10, 20, 20);
  GetMem(ScrnImage, Size);
  if ScrnImage = nil then
    Halt(1);
  SaveScreen(10,1 0, 20, 20, ScrnImage^);
  ClrScr;
  ...
end.
```

Halt procedure

Stops program execution and returns to the operating system.

Declaration:

```
procedure Halt [ ( Exitcode: Word ) ];
```

where:

Exitcode is an optional expression that specifies the exit code of your program.

Remarks:

The exit code can be retrieved in DOS by using ErrorLevel in a batch file. If the program does not return to DOS, use *DosExitCode* to determine the exit code. For more information about ErrorLevel consult your DOS reference manual.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
begin
  if ParamCount <> 1 then
  begin
    WriteLn('Error: No parameters passed.');
    Halt(1);
  end;
  ...
end.
```

Hi function

Returns the high-order byte of the argument.

Declaration:

```
function Hi(X): Byte;
```

Remarks:

Use *Hi* to retrieve the high order byte from an ordinal type argument.

High function

Returns the highest value in the range of the argument.

Declaration:

function High(X)

Inc procedure

Increments a variable.

Declaration:

```
procedure Inc(var X [ ; N: Longint ] );
```

Remarks:

If *N* is not passed to *Inc*, *X* is incremented by one. Otherwise, *Inc* is identical to `X := X + n`. *Inc* produces more efficient code.

Example:

```
Inc(Int,7); // Int  := Int + 7
Inc(Long);  // Long := Long + 1
```

Include procedure

Includes an element in a set.

Declaration:

```
procedure Include(var S: set of T; I:T);
```

Remarks:

S is a set type variable, and *I* is an expression of a type compatible with the base type of *S*. The element given by *I* is included in the set given by *S*.

The construct Include (S,I) corresponds to S := S + (I) but the Include procedure generates more efficient code.

Insert procedure

Inserts a substring into a string.

Declaration:

```
procedure Insert(Src: string; var Dst: string; Pos: Longint);
```

Remarks:

Src is a string-type expression. *Dst* is a string-type variable. *Pos* is an integer expression. *Insert* inserts *Src* into *Dst* at the *Pos*th position. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

Int function

Returns the Integer part of the argument.

Declaration:

```
function Int(X: Real): Real;
```

Remarks:

Int returns the integer portion of a real number rounded toward zero.

Example:

```
i := Int(-456.332);   // -456
i := Int(1231.98192); // 1231
```

IOResult function

Returns the status of the last I/O operation performed. Return zero (0) if successful.

Declaration:

```
function IOResult: Integer;
```

Remarks:

I/O-checking must be off `{$I-}` to trap I/O errors using *IOResult*. If an I/O error occurs and I/O-checking is off, all subsequent I/O operations are ignored until a call is made to *IOResult*. A call to *IOResult* clears the internal error flag. A call to *IOResult* clears its internal error flag. I/O-checking must be off `{$I-}.`

Length function

Returns the dynamic length of a string.

Declaration:

```
function Length(S: String): Integer;
```

Ln function

Returns the natural logarithm of the argument.

Declaration:

```
function Ln(X: Real): Real;
```

Lo function

Returns the low-order Byte of the argument.

Declaration:

```
function Lo(X): Byte;
```

LoCase function

Converts a character to lowercase.

Declaration:

```
function LoCase(Ch: Char): Char;
```

Remarks:

Ch is of char type.

LoCase simply returns an lowercase character if *Ch* is uppercase.

Low function

Returns the lowest value in the range of the argument.

Declaration:

```
function Low(X);
```

Remarks:

Result type is *X*, or the index type of *X* where *X* is either a type identifier or a variable reference.

MaxAvail function

Returns the lowest value in the range of the argument.

Returns the size of the largest contiguous free block in the heap.

Declaration:

```
function MaxAvail: Longint;
```

Win32:

Returns the number of bytes of physical memory available.

MemAvail function

Returns the amount of all free memory in the heap.

Declaration:

```
function MemAvail: Longint;
```

Win32:

Returns the total number of bytes that can be stored in the paging file. Note that the return value does not represent the actual physical size of the paging file on the disk.

MkDir procedure

Creates a subdirectory with given name.

Declaration:

```
procedure MkDir(S: String);
```

Move procedure

Copies bytes from source to dest.

Declaration:

```
procedure Move(var Source, Dest; Count: Longint);
```

Remarks:

Copies a specified number of contiguous bytes (*Count*) from a source range to a destination range. No range-checking is performed. Whenever possible, use **SizeOf** to determine the count.

Note that *Count* is not limited to 65,535 (64K) bytes.

New procedure

Creates a new dynamic variable and sets a pointer variable to point to it.

Declaration:

```
procedure New(Var P: Pointer);
```

Remarks:

New determines the amount of heap to allocate by the size of the typed variable pointed to by *P*. To reference the new variable, use *P^*. If New finds that there is an insufficient amount of free heap, a runtime error is generated.

The Boolean variable HeapCheck determines whether TMT Pascal tracks the size of heap variables. This size is for debugging purposes only and is stored in the four bytes before the start of the variable. For more information about heap allocation see the Heap Management chapter. Note that the size of heap variables is not limited to 66,535 (64K) bytes.

Example:

```
{$ifndef __CON__}
  This program must be compiled as console application only
{$endif}
var
  P: ^String[16];
begin
  New(P);
  P^ := 'How are you?';
  Writeln(P^);
  Dispose(P);
end.
```

Odd function

Tests if the argument is an odd number.

Declaration:

```
function Odd(X: Longint): Boolean;
```

Ofs function

Returns the linear offset in memory of a specified object.

Declaration:

```
function Ofs(X): DWord;
```

Ord function

Returns the ordinal value of an ordinal-type expression.

Declaration:

```
function Ord(X): Longint;
```

ParamCount function

Returns the number of parameters passed to the program on the command line.

Declaration:

```
function ParamCount: Word;
```

Remarks:

Command line parameters are separated by spaces or tabs. To retrieve command line parameters call **ParamStr**.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
begin
  if ParamCount = 0 then
  begin
    WriteLn('No parameters specified.');
    Halt(1);
  end;
end.
```

ParamStr function

Returns a specified command-line parameter.

Declaration:

```
function ParamStr(Index: Word): String;
```

Remarks:

If Index is zero then the path and file name of the current program is returned. Command line parameters are separated by spaces or tabs. To determine the number of command line parameters use **ParamCount**.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
var
  Cnt: Longint;
begin
  WriteLn('Program Name: ',ParamStr(0));
  if ParamCount = 0 then
     Halt(0);
  for Cnt := 1 to ParamCount do
   WriteLn(ParamStr(Cnt));
end.
```

Pi function

Returns the value of Pi, which is defined as 3.1415926535897932385.

Declaration:

```
function Pi: Extended;
```

Pos function

Searches for a substring in a string.

Declaration:

```
function Pos(SubStr: String; S: String): Byte;
```

Remarks:

Pos returns the index of the first character of *SubStr* in *S*. If *SubStr* is not found then *Pos* returns zero.

Example:

```
Index := Pos(‘23’,’123’); // Index is 2
Index := Pos(‘z’,’ABC’);  // Index is 0
```

Pred function

Returns the predecessor of the argument.

Declaration:

```
function Pred(X);
```

Remarks:

X is an ordinal-type expression. The result, of the same type as *X*, is the predecessor of *X*.

Example:

```
n := Pred(500);  // n is 499
```

Ptr function

Converts an offset address to a pointer-type value.

Declaration:

```
function Ptr(Ofs: DWord): Pointer;
```

Random function

Returns a random number.

Declaration:

```
function Random [ ( Range: Word) ];
```

Remarks:

Range, if specified, results in 0 <= Result < *Range*. If not specified, the range is 0 <= Result < 1.

A call to **Randomize** should be made prior to *Random*. This initializes the random number generator.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
begin
  Randomize;
  repeat
    Writeln(Random(65535));
  until KeyPressed;
end.
```

Randomize procedure

Initializes the built-in random number generator with a random value (obtained from the system clock).

Declaration:

```
procedure Randomize;
```

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
begin
  Randomize;
  repeat
    Writeln(Random(65535));
  until KeyPressed;
end.
```

Read procedure

Read from a file into one or more variables.

Declaration:

```
procedure Read(F, var V1[, V2,...,Vn ] );          // For typed
files
procedure Read([var F: Text; ] V1[, V2,...,Vn ]);  // For text
files
```

Remarks:

Depending on the type of the variable or variables passed, *Read* copies the contents of the file, from the current file pointer, into the variable and advances the pointer. If *Read* cannot match the type of the variable with the contents of the file, an I/O error occurs. End of line (#13) as well as end of file (#26) cause *Read* to terminate.

For integers, Read expects an integer number in string form. Tabs, blank spaces, or end of line markers before the number are all skipped. When encountered thereafter, Read terminates.

For real variables, *Read* expects a number in real format. Blanks, tabs or end of line markers preceding the real string are skipped. When encountered thereafter, *Read* stops.

With strings, *Read* reads up to either an end of line marker or an end of file marker. If the string read is longer than 255 characters, it is truncated. *Read* does not advance the file.

ReadLn procedure

Executes the Read procedure then skips to the next line of the file.

Declaration:

```
procedure ReadLn([ var F: Text; ] V1 [, V2, ...,Vn ]);
```

Remarks:

Depending on the type of the variable or variables passed, *ReadLn* copies the contents of the file, from the current file pointer, into the variable and advances the pointer. If *ReadLn* cannot match the type of the variable with the contents of the file, an I/O error occurs. End of line (#13) as well as end of file (#26) cause *ReadLn* to terminate.

The file must be a text file or standard input. *ReadLn* is identical to **Read** except that it advances passed the end of line marker.

With `{$I-}`, **IOResult** returns an error code if the operation was not successful. If no error was encountered, **IOResult** is set to zero.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
var
  Name: String;
  Age:  DWord;
begin
  Write('Enter your name: ');
  ReadLn(Name);
  Write('Enter your age : ');
  ReadLn(Age);
  if Age < 21 then
    WriteLn('You’re so young, ', Name, '!')
  else if Age < 40 then
    WriteLn(Name, ', you`re still in your prime!')
  else if Age < 60 then
    WriteLn('You`re over the hill, ', Name, '!')
  else if Age < 80 then
    WriteLn('I bow to your wisdom, ', Name, '!')
  else
    Writeln('Are you really ', Age, ', ', Name, '?');
end.
```

Rename procedure

Renames an external file.

Declaration:

```
procedure Rename(var F; NewName);
```

Remarks:

Rename must not be used on an open file. All operations after a successful call to *Rename* use the new name. With `{$I-}`, **IOResult** returns zero if *Rename* was successful, otherwise **IOResult** contains an error number.

Reset procedure

Opens an existing file.

Declaration:

```
procedure Reset(var F [: File; RecSize: Longint ] );
```

Remarks:

F is a file variable of any type which was previously associated with a file name by a call to Assign. The external file must exist.

RecSize indicates the size in bytes of each record and can only be passed if *F* is an untyped file. Note that *RecSize* is not limited to 65,535 (64K) bytes.

If F is open when *Reset* is called, *F* is first closed and then reopened. If *F* is assigned to an empty string (‘’) then standard input is used. Text files are opened as read-only. A record size of 128 bytes is assumed if *F* is not a text file and *RecSize* is not passed.

Reset sets the file pointer to the top of the file. With `{$I-}`, **IOResult** returns an error code if an error occurred, otherwise it is set to zero.

ReWrite procedure

Creates and opens a new file.

Declaration:

```
procedure ReWrite(var F: File [; RecSize: Longint ] );
```

Remarks:

F is a file variable of any type which was previously associated with a file name by a call to Assign.

RecSize indicates the size in bytes of each record and can only be passed if *F* is an untyped file. Note that *RecSize* is not limited to 65,535 (64K) bytes.

If F exists or is open when *ReWrite* is called, *F* is closed, deleted, and then recreated. If *F* is assigned to an empty string (‘’), standard input is used. Text files are created as write-only. A record size of 128 bytes is assumed if F is not a text file and *RecSize* is not passed.

With `{$I-}`, **IOResult** returns an error code if an error occurred, otherwise it is set to zero.

RmDir procedure

Removes an empty subdirectory.

Declaration:

```
procedure RmDir(Dir: String);
```

Remarks:

IOResult returns an error code if *Dir* is an empty string, if *Dir* does not exist, or if *Dir* is not an empty directory. Otherwise, **IOResult** is set to zero.

Round function

Rounds a Real-type value to an Integer-type value.

Declaration:

```
function Round(X: Real): Longint;
```

Remarks:

Round returns *X*, rounded to the nearest whole number, as a longint.

Example:

```
i := Round(-456.332);    // -456
i := Round(1231.98192);  // 1232
```

RunError procedure

Stops program execution.

Declaration:

```
procedure RunError [ ( ExitCode: Word ) ];
```

Remarks:

ExitCode, if specified, is the runtime error number. If not passed, the exit code is zero.

RunError is used for debugging purposes. It generates an error during execution along with an offset. This procedure is similar to **Halt**.

Seek procedure

Moves the current position of a file to a specified component.

Declaration:

```
procedure Seek(var F; NewPos: Longint);
```

Remarks:

F is a file of any type except text.

NewPos is the record or component to move the file pointer to.

Seek works only on open files and cannot be used with text files. The first component of any file is zero. With `{$I-}` **IOResult** returns an error code if an error occurred, otherwise it is set to zero.

SeekEof procedure

Returns the end-of-file status of a file.

Declaration:

```
function SeekEof [ (var F: Text) ]: Boolean;
```

Remarks:

SeekEof works only on text files and is similar to *Eof* however it ignores blanks, tabs, and end of line markers that may exist before the end of file marker. With `{$I-}`, **IOResult** returns an error code if an error occurred, otherwise it is set to zero.

SeekEoln procedure

Returns the end-of-line status of a file.

Declaration:

```
function SeekEoln [ (var F: Text) ]: Boolean;
```

Remarks:

Must be used on text files. File (*F*) must be open.

SetString procedure

Sets the contents and length of the given string.

Declaration:

```
procedure SetString(var S: string; Buffer: PChar; Len: DWORD);
```

Remarks:

The *SetString* sets the length indicator character (the character at *S[0]*) to the value given by *Len* and then, if the *Buffer* parameter is not **nil**, copies *Len* characters from *Buffer* into the string starting at *S[1]*. For a short string variable, the *Len* parameter must be a value between 0 and 255.

SetTextBuf procedure

Assigns an I/O buffer to a text file.

Declaration:

```
procedure SetTextBuf(var F: Text; var Buf [ ; Size: LongInt ]);
```

Remarks:

F is a text file that has not yet been opened. *Buffer* is a variable of any type. *Size*, if specified, is the size of the new buffer. Otherwise **SizeOf**`(Buffer)` is assumed.

Note that the new buffer's size is not limited to 65,535 (64K) bytes.

By default each text file has a 128 byte buffer that is used for I/O. *SetTextBuf* assigns a new buffer to the text file. By creating a larger buffer I/O operations become faster because there are fewer disk reads and writes.

Sin function

Returns the sine of the argument.

Declaration:

```
function Sin(X: Real): Real;
```

SizeOf function

Returns the number of bytes occupied by the argument.

Declaration:

```
function SizeOf(X): Longint;
```

Remarks:

X is a variable reference or type identifier. *SizeOf* returns the number of bytes used by the argument.

Sqr function

Returns the square of the argument.

Declaration:

```
function Sqr(X);
```

Remarks:

X is either an integer type or real type. *Sqr* returns (*X* * *X*).

Sqrt function

Returns the square root of the argument.

Declaration:

```
function Sqrt(X: Real): Real;
```

Remarks:

X is either single, double, or extended.

Str procedure

Converts a numeric value to a string.

Declaration:

```
procedure Str(X [: Width [: Decimals ]]; var S:string);
```

Remarks:

X is either a real or integer type. *Width*, if specified, is the number of characters to reserve for the integer portion of the number. *Decimals*, if specified, is the number of character to reserve for the decimal portion of a real type number.

S returns the result of the operation. *Str* formats numbers exactly like Write does when printing integers or real types.

Succ function

Returns the successor of the argument.

Declaration:

```
function Succ(X);
```

Remarks:

X is an expression of ordinal type.

Example:

```
n := Succ(4);   // n = 5
c := Succ(‘a’); // c = ‘b’
```

Swap function

Swaps the high- and low-order bytes of the argument.

Declaration:

```
function Swap(X: Integer): Integer;
function Swap(X: Longint): Longint;
```

Example:

```
n := Swap($1234);  // n = $3412
```

Trunc function

Truncates a real-type value to an Integer-type value.

Declaration:

```
function Trunc(X: Real): Longint;
```

Remarks:

X is either single, double, or extended. *Trunc* returns *X*, rounded towards zero, as a Longint.

Truncate procedure

Truncates the file at the current file position.

Declaration:

```
procedure Truncate(var F);
```

Remarks:

F is any type file except text.

Truncate deletes all data passed the current file pointer of *F*. An end of file marker, Ctrl-Z, is inserted at the current file pointer. With `{$I-}` **IOResult** returns an error code if the operation failed. Otherwise **IOResult** is set to zero.

UpCase function

Converts a character to uppercase.

Declaration:

```
function UpCase(Ch: Char): Char;
```

Remarks:

Ch is of char type.

UpCase simply returns an uppercase character if *Ch* is lowercase.

Val procedure

Converts a string value to its numeric representation.

Declaration:

```
procedure Val(S; var Value; var Code: Integer);
```

Remarks:

S is of string type. *Value* is an integer or real type variable. *Code* returns an integer value indicating whether the operation was successful or where it failed.

S is a string of numeric characters. *Val* attempts to convert *S* to a valid integer or real types number. If successful, *Code* returns zero, otherwise *Code* returns the index in *S* where the conversion failed.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
uses Strings;
var
  Num: String;
  Code: Longint;
  Value: Extended;
begin
  Write('Enter a number: ');
  ReadLn(Num);
  Val(Num, Value, Code);
  if Code <> 0 then
```

```
    Writeln('Error at position: ', Code)
  else
    Writeln('(Value * 2) = ', Fls(Value * 2));
end.
```

Write procedure

For typed files, writes a variable into a file component. For text files, writes one or more values to the file

Declaration:

```
procedure Write( [ var F: Text; ] P1 [,P2,...,Pn ];
```

Remarks:

F, if specified, is a file variable. If omitted Output is assumed.

P1 may be of char, string, integer, real, or Boolean type. *P*1 through *Pn* are output to *F*. Each *P* parameter may be formatted as follows:

P [:MinWidth [:Decimals]]

where *P* is the value to output. MinWidth, which must be greater than zero, specifies the minimum width of *P*. Decimals specifies the number of decimal places to be output when P is of real type.

Example:

```
{$ifndef __CON__}
  This program must be compiled as a console application only.
{$endif}
begin
  Write('A wonderful string!');
  Write(500);
  Write(5.5:5:2)
  Write(True);
end.
```

WriteLn procedure

Executes the Write procedure, then outputs an end-of-line marker to the file.

Declaration:

```
procedure Writeln([ var F: Text; ] P1 [, P2, ...,Pn ] );
```

Remarks:

F, if specified, is a file variable. If omitted Output is assumed. *P1* may be of char, string, integer, real, or Boolean type. *P1* through *Pn* are output to *F*. Each *P* parameter may be formatted as follows:

P [:MinWidth [:Decimals]]

where *P* is the value to output. MinWidth, which must be greater than zero, specifies the minimum width of *P*. Decimals specifies the number of decimal places to be output when *P* is of real type.

The file used by *WriteLn* must be open for output. An end of line marker is written to *F* after output.

Example:

```
{$ifndef __CON__}
```

```
  This program must be compiled as a console application only.
{$endif}
var
  Name: String;
  Age:  DWord;
begin
  Write('Enter your name: ');
  ReadLn(Name);
  Write('Enter your age : ');
  ReadLn(Age);
  if Age < 21 then
    WriteLn('You’re so young, ', Name, '!')
  else if Age < 40 then
    WriteLn(Name, ', you`re still in your prime!')
  else if Age < 60 then
    WriteLn('You`re over the hill, ', Name, '!')
  else if Age < 80 then
    WriteLn('I bow to your wisdom, ', Name, '!')
  else
    Writeln('Are you really ', Age, ', ', Name, '?');
end.
```

Chapter 16

The Use32 Unit

Targets: MS-DOS, OS/2, Win32

Contains redefinition of integer types for 32-bit computing as follows:

```
type
   SmallInt   = System.Integer;
   SmallWord  = System.Word;
   Integer    = System.Longint;
   Word       = System.Longint;
const
   MaxInt     = high(longint);
type
   PByte      = ^Byte;
   PWord      = ^Word;
   PLongint   = ^Longint;
   PSmallInt  = ^SmallInt;
   PSmallWord = ^SmallWord;
```

Chapter 17

Win32 API Interface Units

Targets: Win32 only

TMT Pascal comes with set of special run-time library units, which provide a Win32 API interface. The names of these units are listed below.

AccCtrl	- Windows 32bit Common new style Win32 Access Control unit
AclAPI	- Windows 32bit acl and trusted server access control APIs interface unit
CommCtrl	- Windows 32bit Common Controls interface unit
CommDlg	- Windows 32bit Common Dialog APIs interface unit
Cpl	- Windows 32bit Control panel extension DLL definitions unit
Ddeml	- Windows 32bit DDEML API interface unit
Dlgs	- Windows 32bit UI dialog header information unit
HtmlHelp	- Windows 32bit Html Help API unit
ImageHlp	- Windows 32bit Image help routines
Imm	- Windows 32bit Input Method Manager definitions unit
LZExpand	- Windows 32bit Data Decompression library functions
MAPI	- Windows 32bit Messaging Applications Programming Interface unit
Messages	- Windows 32bit Messages interface unit
MMSystem	- Windows 32bit Multimedia interface unit
NB30	- Windows 32bit NetBIOS 3.0 interface unit
Regstr	- Windows 32bit Registry interface unit
RichEdit	- Windows 32bit RichEdit 2.0 control interface unit
ShellApi	- Windows 32bit Shell API interface unit
TlHelp32	- Windows 32bit Tool help unit
Windows	- Windows 32bit Base API interface unit
WinINet	- Microsoft Windows Internet Extensions API interface unit
WinSock	- WINSOCK.DLL API interface unit
WinSpool	- Windows 32bit Print API interface unit
WinSvc	- Windows 32bit Service Control Manager unit

For more information refer to Microsoft Win32 Programmer's Reference and Microsoft Multimedia Programmer's Reference. Also, you will find sources of all Win32 API interface units in the **\TMTPL\SOURCE\WIN32** directory.

Chapter 18

The WinCRT Unit

Targets: Win32 GUI only

The WinCrt unit emulates a terminal-like colored text screen in a Win32 GUI window. If your program uses WinCrt, you do not need to write «Windows-specific» code.

18.1 WinCRT Unit Constants and Variables

AutoTracking variable

Controls automatic cursor tracking in the CRT window.

Declaration:

```
var
  AutoTracking: Boolean := TRUE;
```

Remarks:

When *AutoTracking* is TRUE, the WinCRT window automatically scrolls to ensure that the cursor is visible after each *Write(Ln).*

If *AutoTracking* is FALSE, the WinCRT window will not scroll automatically, and text written to the window may not be visible to the user.

CloseOnExit variable

Defines when the WinCRT will automatically be closed at the end of the program.

Declaration:

```
var
  CloseOnExit: Boolean := FALSE;
```

Remarks:

When *CloseOnExit* is TRUE, the WinCRT window will be closed automatically after program termination.

If *CloseOnExit* is FALSE, the WinCRT window will not be closed after program termination, and it will enter an inactive state.

Color constants

Use these color constants with **TextColor** and **TextBackGround** procedures.

Declaration:

```
const
  Black         = $000000;
  Maroon        = $000080;
  Green         = $008000;
  Olive         = $008080;
  Navy          = $800000;
  Purple        = $800080;
  LightCyan     = $800080;
  Teal          = $808000;
  Gray          = $808080;
  Silver        = $C0C0C0;
  Red           = $0000FF;
  Lime          = $00FF00;
  Yellow        = $00FFFF;
  Blue          = $FF0000;
  Fuchsia       = $FF00FF;
  Aqua          = $FFFF00;
  LightGray     = $C0C0C0;
  DarkGray      = $808080;
  White         = $FFFFFF;
```

CurOrg variable

Determines the cursor origin.

Declaration:

```
const
  CurOrg: TLongPoint = (X: 0; Y: 0);
```

Remarks:

The upper left corner of cursor corresponds to (0, 0).

Cursor variable

Contains the current position of the cursor within the WinCRT window.

Declaration:

```
const
  Cursor: TLongPoint = (X: 0; Y: 0);
```

Remark:

The upper left corner corresponds to **CurOrg**. *Cursor* is a read-only variable, so do not assign values to it.

InactiveTitle variable

Points to a null-terminated string. Use when constructing the title of an inactive WinCRT window.

Declaration:

```
const
  InactiveTitle: PChar = '(Inactive %s)';
```

ScreenSize variable

Determines the width and height in characters of the virtual screen within the WinCRT window.

Declaration:

```
const
  ScreenSize: TLongPoint = (X: 80; Y: 25);
```

ScrollCrtWindow variable

Controls scrolling of the virtual window by means of arrow keys.

Declaration:

```
var
  ScrollCrtWindow: Boolean := TRUE;
```

Remarks:

When *ScrollCrtWindow* is TRUE, the virtual window can be scrolled by means of arrow keys.

If *ScrollCrtWindow* is FALSE, the WinCRT window cannot be scrolled with arrow keys.

WindowOrg variable

Determines the initial location of the WinCRT window.

Declaration:

```
const
WindowOrg: TLongPoint = (X: CW_USEDEFAULT; Y: CW_USEDEFAULT);
```

Remark:

You can change the initial location by assigning new values to the *X* and *Y* coordinates before the WinCRT window is created.

WindowSize variable

Determines the initial size of the WinCRT window.

Declaration:

```
const
 WindowSize: TLongPoint = (X: CW_USEDEFAULT; Y: CW_USEDEFAULT);
```

Remark:

You can change the initial size by assigning new values to the *X* and *Y* coordinates before the WinCRT window is created.

WindowTitle variable

Determines the title of the WinCRT window.

Declaration:

```
const
  WindowTitle: array [0..79] of Char;
```

18.2 WinCRT Unit Procedures and Functions

CursorTo procedure

Moves the cursor to the given coordinates within the WinCRT window.

Declaration:

```
procedure CursorTo(X, Y: Longint);
```

Remark:

CursorTo(X, Y) is equivalent to GotoXY(X + 1, Y + 1).

See also: **GotoXY**.

DoneWinCRT procedure

Destroys the CRT GUI window if it has not already been destroyed.

Declaration:

```
procedure DoneWinCrt;
```

Remarks:

Call *DoneWinCrt* before the program ends to prevent the WinCRT window from entering the inactive state.

See also: **CloseOnExit** variable.

InitWinCRT procedure

Creates the CRT GUI window if it has not already been created.

Declaration:

```
procedure InitWinCrt;
```

Remark:

InitWinCrt is automatically called if you have used *Read(Ln) or Write(Ln)* in a file that has been assigned to the WinCRT.

ReadBuf procedure

Inputs a line from the WinCRT window.

Declaration:

```
procedure ReadBuf(Buffer: PChar; Count: DWORD): DWORD;
```

ScrollTo procedure

Scrolls the WinCRT window to show the virtual screen location given by (*X*,*Y*) in the upper left corner.

Declaration:

```
procedure ScrollTo(X, Y: Longint);
```

TrackCursor procedure

Scrolls the WinCRT window if necessary to ensure that the cursor is visible.

Declaration:

```
procedure TrackCursor;
```

WriteBuf procedure

Writes a block of characters to the WinCRT window.

Declaration:

```
procedure WriteBuf(Buffer: PChar; Count: Word);
```

Remark:

Buffer points to the first character in the block. *Count* contains the number of characters to write.

WriteChar procedure

Writes a single character to the WinCRT window.

Declaration:

```
procedure WriteChar(Ch: Char);
```

Chapter 19

The WinDos Unit

Targets: MS-DOS, OS/2, Win32

The WinDos unit allows easy access to most of the functions provided by the MS-DOS operating system for a 32-bit protected mode application. Also the Dos unit emulates MS-DOS functions under OS/2 and Win32 using the standard API, provided by the OS/2 and Win32 operating systems. Operations such as find file, disk size or status, time and date, get environment strings and more are provided by the WinDos unit. It provides a PChar interface to the file handling functions.

19.1 WinDos Unit Constants and Variables

TDataTime type

The **UnpackTime** and **PackTime** procedures use variables of type *DateTime* to examine and construct 4-byte, packed date-and-time values for the **GetFTime**, **SetFTime**, **FindFirst**, and **FindNext** procedures:

Declaration:

```
type
  TDateTime = record
    Year, Month, Day, Hour,
    Min, Sec: Word;
  end;
```

TRegisters type

Targets: MS-DOS only

The **Intr** and **MsDos** procedures use variables of type *Registers* to specify the input register contents and examine the output register contents of a software interrupt.

Declaration:

```
type TRegisters =
  record
  case Integer of
    1: (edi, esi, ebp, _res, ebx, edx, ecx, eax: Longint;
        flags, es, ds, fs, gs, ip, cs, sp, ss: Word);
    2: (_dmy2: array [0..15] of byte; bl, bh, b1, b2, dl,
        dh, d1, d2, cl, ch, c1, c2, al, ah: Byte);
    3: (di, i1, si, i2, bp, i3, i4, i5, bx, b3, dx, d3, cx,
        c3, ax: Word);
  end;
```

TSearchRec type

The **FindFirst** and **FindNext** procedures use variables of type *SearchRec* to scan directories:

Declaration:

MS-DOS target:

```
type
  TSearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : string[12];
end;
```

OS/2 target:

```
type
  TSearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : string;
end;
```

Win32 target:

```
type
  TSearchRec = record
    Fill : array[1..21] of Byte;
    Attr : Byte;
    Time : Longint;
    Size : Longint;
    Name : TFileName;
    ExcludeAttr: Longint;
    FindHandle: THandle;
    FindData: TWin32FindData;
  end;
```

Information for each file found by FindFirst or FindNext is reported back in a *SearchRec*.

Field	Meaning
Attr	File’s attributes
Time	File’s packed date and time
Size	File’s size, in bytes
Name	File’s name

The *Fill* field is reserved by DOS and should never be modified.

19.2 WinDos Unit Procedures and Functions

CreateDir procedure

Creates a new subdirectory.

Declaration:

```
procedure CreateDir(Dir: PChar);
```

Remarks:

Performs the same functions as **MkDir**, but uses a null-terminated (PChar) string rather than a Pascal-style string.

FileExpand function

Expands a file name into a fully-qualified file name.

Declaration:

```
function FileExpand(Dest, Name: PChar): PChar;
```

FileSearch function

Searches for a file.

Declaration:

```
function FileSearch(Dest, Name, List: PChar): PChar;
```

Remarks:

List is a list of the directories to include in the search; each is delimited with a semicolon (;).

FileSearch returns the directory and file name if the file has been located. If *Path* is not found then an empty string is returned. *FileSearch* always begins with the current directory and then checks the directories listed in *List* in the order that they appear.

See also: **FindFirst, FileExpand, FileSplit**

FileSplit procedure

Splits a file name into its three components.

Declaration:

```
procedure FileSplit(Path, Dir, Name, Ext: PChar): DWORD
```

Remarks:

Use this procedure to break down a file specification into three parts: path, file name, and file extension. *Dir* returns the path or directory part of *Path*. *Name* returns the actual file name without extension. *Ext* returns the file extension preceded by a period (.).

It is possible that one or more of the components is returned empty. This occurs if *Path* contains no such component. For instance, if there is no path, *Dir* is empty.

GetArgCount function

Returns the number of parameters passed to the program on the command line.

Declaration:

```
function GetArgCount: Longint;
```

Remarks:

Command line parameters are separated by spaces or tabs. To retrieve command line parameters call **GetArgCount**.

Example:

```
uses WinDos;
begin
  if GetArgCount = 0 then
  begin
    WriteLn('No parameters specified.');
    Halt(1);
  end;
end.
```

GetArgStr function

Returns a specified command-line parameter.

Declaration:

```
function GetArgStr(Dest: PChar; Index: Longint; MaxLen: DWORD):
PChar;
```

GetCurDir function

Returns the current directory of a specified drive.

Declaration:

```
function GetCurDir(Dir: PChar; Drive: Byte): PChar;
```

Remarks:

where

0	Default drive
1	Drive A
2	Drive B
3	Drive C

and so on...

GetEnvVar function

Returns the value of a specified environment variable.

Declaration:

```
function GetEnvVar(var Name: PChar): PChar;
```

Remark:

Name is the name of the variable to retrieve. If *Name* does not exist as an environment variable then an empty string is returned.

RemoveDir procedure

Removes an empty subdirectory.

Declaration:

```
procedure RemoveDir(Dir: PChar);
```

SetCurDir procedure

Changes the current directory to the specified path.

Declaration:

```
procedure SetCurDir(Dir: PChar);
```

Chapter 20

The ZenTimer Unit

Targets: MS-DOS, OS/2, Win32

Description

The ZenTimer unit is a full-featured port to TMT Pascal of the ZTimer library by SciTech Software and is based on the original C/C++ code by Kendall Bennett, SciTech Software. The ZenTimer description is based on the MegaGraph Graphics Library Reference Manual Copyright © 1996 SciTech Software Inc.

Copyrights

Features

- High resolution Zen Timer (based on code from book “Zen of Assembly Language” Volume 1, Knowledge by Michael Abrash).
- Ultra long Zen Timer (interface to the BIOS Timer Tick for timing code that takes up to 24 hours).
- RDTSC support (uses supports the Intel RDTSC instruction, for high precision timing).
- Routines to obtain CPU information (type, speed, features, 3DNow!™ and MMX™ support).
- LZTimer and ULZTimer objects (provide a set of Pascal objects to manipulate the Zen Timers).

20.1 ZenTimer Unit Procedures and Functions

CPU_getCPUIdFeatures function

Returns CPUID features (family/model/stepping/features).

Declaration:

```
function CPU_getCPUIdFeatures: DWord;
```

CPU_getProcessorSpeed function

Returns the speed of the processor in Mhz.

Declaration:

```
function CPU_getProcessorSpeed: DWord;
```

Remarks:

This function returns the speed of the CPU in Mhz. Note that if the speed cannot be determined, this function will return 0.

Example:

```
{$ifndef __CON__}
  This program must be compiled for MS-DOS, OS/2 or Win32
console mode.
{$endif}
uses ZenTimer;
begin
  Writeln('CPU speed is ', CPU_getProcessorSpeed,' Mhz');
end.
```

CPU_getProcessorType function

Returns the type of processor in the system.

Declaration:

```
function CPU_getProcessorType: DWord;
```

Remarks:

Returns the type of processor in the system. Note that if the CPU is an unknown Pentium family processor that we don't have an enumeration for, the return value will be greater than or equal to the value of CPU_UnkPentium (depending on the value returned by the CPUID instruction).

The following constants are defined:

```
const

  CPU_unknown     = 0;   // Unknown proccessor
  CPU_i386        = 1;   // Intel 80386 processor
  CPU_i486        = 2;   // Intel 80486 processor
  CPU_Pentium     = 3;   // Intel Pentium® processor
  CPU_PentiumPro  = 4;   // Intel PentiumPro® processor
  CPU_PentiumII   = 5;   // Intel PentiumII® processor
  CPU_PentiumIII  = 6;   // Intel PentiumIII® processor
  CPU_UnkPentium  = 7;   // Unknown Intel Pentium family
processor
```

CpuTypes array strings are also defined.

```
const
  CpuTypes: array [0..7] of string =
  (
  'Unknown',
  'Intel 80386',
  'Intel 80486',
  'Intel Pentium(R)',
  'Intel PentiumPro(R)',
  'Intel PentiumII(R)',
  'Intel PentiumIII(R)',
  'Unknown Pentium'
  );
```

Example:

```
{$ifndef __CON__}
  This program must be compiled for MS-DOS, OS/2 or Win32
console mode.
{$endif}
uses ZenTimer;
begin
  Writeln('CPU is ', CpuTypes[CPU_getProcessorType]);
end.
```

CPU_haveMMX function

Returns True if the processor supports Intel MMX™ extensions.

Declaration:

```
function CPU_haveMMX: Boolean;
```

Remarks:

This function determines if the processor supports the Intel MMX™ extended instruction set. If the processor is not an Intel or Intel clone CPU, this function will always return False.

Example:

```
{$ifndef __CON__}
  This program must be compiled for MS-DOS, OS/2 or Win32
console mode.
{$endif}
uses ZenTimer;
begin
  if CPU_haveMMX then
    Writeln('MMX technology supported')
  else
    Writeln('MMX technology not supported');
end.
```

CPU_have3DNow function

Returns True if the processor supports AMD 3DNow!™ extensions.

Declaration:

```
function CPU_have3DNow: Boolean;
```

Remarks:

This function determines if the processor supports the AMD 3DNow!™ extended instruction set.

Example:

```
{$ifndef __CON__}
  This program must be compiled for MS-DOS, OS/2 or Win32
console mode.
{$endif}
uses ZenTimer;
begin
```

```
  if CPU_have3DNow then
    Writeln('3DNow! technology supported')
  else
    Writeln('3DNow! technology not supported');
end.
```

LZDelay procedure

Delays a specified number of 1e-6 sec.

Declaration:

```
procedure LZDelay (Value: DWord);
```

LZTimerCount function

Returns the current count for the Long Period Zen Timer.

Declaration:

```
function LZTimerCount: Dword;
```

Remarks:

Returned value is a current count that has elapsed between calls to **LZTiemerOn** and **LZTiemerOff** in microseconds.

Example:

```
program LZTest;

{$ifndef __CON__}
  This program must be compiled for MS-DOS, OS/2 or Win32
console mode.
{$endif}
uses ZenTimer;
function lu06(val: Longint): String;
var
  i: Longint;
  s: String;
begin
  Str(val:6,s);
  for i := 1 to 6 do
    if s[i] = ' ' then s[i] := '0';
  lu06 := s;
end;

procedure ReportTime(count: Longint);
var
  secs: Longint;
begin
  secs := count div 1000000;
  count := count - secs * 1000000;
  Writeln('Time taken: ', secs, '.', lu06(count), ' seconds');
end;

var
  i, j: DWord;
```

```
begin
  LZTimerOn;
   for j := 0 to 9 do
     for i := 0 to 19999 do
       i := i;  // do something
  LZTimerOff;
  ReportTime(LZTimerCount);
end.
```

LZTimerLap function

Returns the current count for the Long Period Zen Timer and keeps it running (count that has elapsed in microseconds).

Declaration:

```
function LZTimerLap: Dword;
```

Remarks:

Returned value is the current count that has elapsed since the last call to **LZTiemerOn** in microseconds. The time continues to run after this function is called so you can call this function repeatedly.

LZTiemerOff procedure

Stops the Long Period Zen Timer counting.

Declaration:

```
procedure LZTimerOff;
```

Remarks:

Stops the Long Period Zen Timer counting and fixes the count. Once you have stopped the timer you can read the count with **LZTimerCount**. If you need highly accurate timing, you should use the on and off functions rather than the lap function since the lap function does not subtract the overhead of the function calls from the timed count.

LZTiemerOn procedure

Starts the Long Period Zen Timer counting.

Declaration:

```
procedure LZTimerOn;
```

Remarks:

Starts the Long Period Zen Timer counting. Once you have started the timer, you can stop it with **LZTiemerOff** or you can latch the current count with **LZTimerLap**.

The Long Period Zen Timer uses a number of different high precision timing mechanisms to obtain microsecond accurate timings results whenever possible. The following different techniques are used depending on the runtime environment and the CPU of the target machine. If the target system has a Pentium CPU installed which supports the Read Time

Stamp Counter instruction (RDTSC), the Zen Timer library will use this to obtain the maximum timing precision available.

If the Pentium RDTSC instruction is not available, we then do all timing using the old style 8253 timer chip. The 8253 timer routines provide highly accurate timings results in pure DOS mode, however in a DOS box under Windows or other Operating Systems the virtualization of the timer can produce inaccurate results.

Because the Long Period Zen Timer stores the results in a 32-bit unsigned integer, you can only time periods of up to 2^{32} microseconds, or about 1hr 20mins. For timing longer periods use the Ultra Long Period Zen Timer.

LZTimerResolution function

Returns the resolution of the Ultra Long Period Zen Timer.

Declaration:

```
function LZTimerResolution: Real;
```

Remarks:

Returns the resolution of the Long Period Zen Timer as a floating point value measured in seconds per timer count. This function always returns 1e-6.

ULZDelay procedure

Delays a specified number of 0.054925 sec.

Declaration:

```
procedure ULZDelay (Value: DWord);
```

ULZElapsedTime function

Compute the elapsed time between two timer counts.

Declaration:

```
function ULZElapsedTime(start, finish: DWord): DWord
```

Remarks:

Returns the elapsed time for the Ultra Long Period Zen Timer in units of the timers resolution (1/18th of a second under DOS). This function correctly computes the difference even if a midnight boundary has been crossed during the timing period.

ULZReadTime function

Reads the current time from the Ultra Long Period Zen Timer.

Declaration:

```
function ULZReadTime: DWord;
```

Remarks:

Returned value is a current Ultra Long Period Zen Timer. Current count is returned. You can use the **ULZElapsedTime** function to find the elapsed time between two timer count readings.

ULZTimerCount function

Returns the current count for the Ultra Long Period Zen Timer.

Declaration:

```
function ULZTimerCount: DWord;
```

Remark:

The returned value is a current count that has elapsed between calls to **ULZTimerOn** and **ULZTimerOff** in resolution counts.

ULZTimerLap function

Returns the current count for the Ultra Long Period Zen Timer and keeps it running.

Declaration:

```
function ULZTimerLap: DWord;
```

Remarks:

The returned value is a current count that has elapsed since the last call to **ULZTimerOn** in microseconds. The time continues to run after this function is called so you can call this function repeatedly.

ULZTimerOff procedure

Stops the Long Period Zen Timer counting.

Declaration:

```
procedure ULZTimerOff;
```

Remarks:

Stops the Ultra Long Period Zen Timer counting and fixes the count. Once you have stopped the timer you can read the count with **ULZTimerCount**.

ULZTimerOn procedure

Starts the Ultra Long Period Zen Timer counting.

Declaration:

```
procedure ULZTimerOn;
```

Remarks:

Starts the Ultra Long Period Zen Timer counting. Once you have started the timer, you can stop it with ULZTimerOff or you can fix the current count with **ULZTimerLap**.

The Ultra Long Period Zen Timer uses the available operating system services to obtain accurate timings results with as much precision as the operating system provides, but with enough granularity to time longer periods of time than the Long Period Zen Timer. Note that the resolution of the timer ticks is not constant between different platforms, and you should use the **ULZTimerResolution** function to determine the number of seconds in a single tick of the timer, and use this to convert the timer counts to seconds.

Under 32-bit DOS, we use the system timer tick which runs at 18.2 times per second. Given that the timer count is returned as an unsigned 32-bit integer, this we can time intervals that are a maximum of 2^{32} * (1/18.2) in length (or about 65,550 hours or 2731 days!).

ULZTimerResolution function

Returns the resolution of the Ultra Long Period Zen Timer.

Declaration:

```
function ULZTimerResolution: Real;
```

Remarks:

Returns the resolution of the Ultra Long Period Zen Timer as a floating point value measured in seconds per timer count. This function always returns 0.054925.

ZTimerInit procedure

Initializes the Zen Timer library.

Declaration:

```
procedure ZTimerInit;
```

Remarks:

This function initializes the Zen Timer library, and must be called before any of the remaining Zen Timer library functions are called.