PROGRAMMER'S REFERENCE

RUN-TIME LIBRARY
COMMAND-LINE COMPILER
ERROR MESSAGES
COMPILER DIRECTIVES

BORLAND
Turbo Pascal®
Version 7.0
Programmer’s Reference
Introduction
What's in this manual? ................. 1
How to contact Borland ............. 2

Chapter 1 Library reference
Sample procedure .................. 5
Abs function ...................... 6
Addr function .................... 6
Append procedure ................ 6
ArcCoordsType type .............. 7
Arc procedure ................... 8
ArcTan function ................ 8
Assign procedure ............... 9
AssignCrt procedure ........... 10
Assigned function ............. 10
Bar constants ................... 11
Bar procedure ................... 11
Bar3D procedure ................ 12
BitBlit operators ............... 12
BlockRead procedure .......... 13
BlockWrite procedure .......... 14
Break procedure ................ 15
ChDir procedure ............... 15
CheckBreak variable .......... 16
CheckEOF variable .......... 16
CheckSnow variable .......... 16
Chr function .................. 17
Circle procedure ............... 17
ClearDevice procedure ........ 18
ClearViewPort procedure .... 19
Clipping constants ............ 19
Close procedure ............... 20
CloseGraph procedure ........ 20
ClrEol procedure ............. 21
ClrScr procedure ............. 21
Color constants .............. 22
Color constants for SetRGBPalette .... 22
Concat function .......... 23
Continue procedure .......... 23
Copy function ............ 24
Cos function ............. 25
CreateDir procedure ........ 25
crt mode constants .... 25
Cseg function ................ 26
DateTime type ............ 26
Dec procedure ............ 27
Delay procedure ............ 27
Delete procedure ........... 27
DelLine procedure ............ 28
DetectGraph procedure ....... 28
DirectVideo variable .... 29
DiskFree function ....... 30
DiskSize function ....... 30
Dispose procedure ....... 30
DosError variable .......... 31
DosExitCode function ....; 32
DosVersion function ....; 32
DrawPoly procedure ....... 32
Driver and mode constants .... 33
DSeg function .......... 35
Ellipse procedure ........... 35
EnvCount function ....... 36
EnvStr function ........... 36
Eof function (text files) .... 36
Eof function (typed, untyped files) .... 37
Eoln function .......... 37
Erase procedure ............ 38
ErrorAddr variable ....... 39
Exclude procedure ....... 39
Exec procedure ............ 39
Exit procedure ............ 40
ExitCode variable ....... 41
InitGraph procedure ................ 90
InOutRes variable .................. 92
Input variable ...................... 92
Insert procedure ................... 93
InsLine procedure ................... 93
InstallUserDriver function ......... 94
InstallUserFont function .......... 96
Int function ....................... 97
Intr procedure ..................... 97
IOResult function ................... 98
Justification constants ............. 99
Keep procedure ..................... 99
Key Pressed function ............... 99
LastMode variable .................. 100
Length function ..................... 100
Line procedure ..................... 100
Line style constants ............... 101
LineRel procedure .................. 102
LineSettingsType type .............. 103
LineTo procedure ................... 103
Ln function ........................ 104
Lo function ........................ 104
Low function ....................... 104
LowVideo procedure ............... 105
Lst variable ....................... 106
MaxAvail function .................. 106
MaxColors constant ................. 107
MemAvail function .................. 107
MkDir procedure .................... 108
Move procedure ..................... 108
MoveRel procedure .................. 109
MoveTo procedure ................... 110
MsDos procedure .................... 110
New procedure ...................... 111
NormVideo procedure .............. 111
NoSound procedure ................. 112
Odd function ....................... 112
Ofs function ....................... 112
Ord function ....................... 112
Output variable .................... 113
OutText procedure .................. 113
OutTextXY procedure ............... 115
OvrClearBuf procedure ............. 116
OvrCodeList variable .............. 116
OvrDebugPtr variable .............. 117
OvrDosHandle variable ............. 117
OvrEmsHandle variable ............. 117
OvrFileMode variable .............. 118
OvrGetBuf function ............... 118
OvrGetRetry function .............. 118
OvrHeapEnd variable ............... 119
OvrHeapOrg variable ............... 119
OvrHeapPtr variable ............... 120
OvrHeapSize variable .............. 120
OvrInit procedure .................. 120
OvrInitEMS procedure .............. 121
OvrLoadCount variable ............. 122
OvrLoadList variable .............. 122
OvrReadBuf variable ............... 123
OvrResult variable ............... 123
OvrSetBuf procedure ............... 123
OvrSetRetry procedure ............. 124
OvrTrapCount variable ............. 125
ovrXXXX constants ................. 125
PackTime procedure ............... 125
PaletteType type ................... 126
ParamCount function ............... 126
ParamStr function ................. 127
Pi function ....................... 127
PieSlice procedure ............... 127
PointType type ..................... 128
Pos function ....................... 128
Pred function ...................... 129
PrefixSeg variable ................ 129
PutImage procedure ............... 130
PutPixel procedure ............... 132
Random function ................... 132
Randomize procedure ............... 133
RandSeed variable .................. 133
Read procedure (text files) ....... 133
Read procedure (typed files) ...... 135
ReadKey function .................. 135
Readln procedure ................... 136
Rectangle procedure ............... 136
RegisterBGIdriver function ...... 137
RegisterBGIfont function ......... 139
Registers type .................... 141
### Chapter 2 Compiler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align data</td>
<td>209</td>
</tr>
<tr>
<td>Boolean evaluation</td>
<td>209</td>
</tr>
<tr>
<td>Debug information</td>
<td>209</td>
</tr>
<tr>
<td>DEFINE directive</td>
<td>209</td>
</tr>
<tr>
<td>ELSE directive</td>
<td>209</td>
</tr>
<tr>
<td>Emulation</td>
<td>209</td>
</tr>
<tr>
<td>ENDIF directive</td>
<td>209</td>
</tr>
<tr>
<td>Extended syntax</td>
<td>209</td>
</tr>
<tr>
<td>Force far calls</td>
<td>209</td>
</tr>
<tr>
<td>Generate 80286 Code</td>
<td>209</td>
</tr>
<tr>
<td>IFDEF directive</td>
<td>209</td>
</tr>
<tr>
<td>IFNDEF directive</td>
<td>209</td>
</tr>
<tr>
<td>IFOPT directive</td>
<td>209</td>
</tr>
<tr>
<td>Include file</td>
<td>209</td>
</tr>
<tr>
<td>Input/output checking</td>
<td>209</td>
</tr>
<tr>
<td>Link object file</td>
<td>209</td>
</tr>
<tr>
<td>Local symbol information</td>
<td>209</td>
</tr>
<tr>
<td>Memory allocation sizes</td>
<td>209</td>
</tr>
<tr>
<td>Numeric coprocessor</td>
<td>209</td>
</tr>
<tr>
<td>Open string parameters</td>
<td>209</td>
</tr>
<tr>
<td>Overflow checking</td>
<td>209</td>
</tr>
<tr>
<td>Overlay code generation</td>
<td>209</td>
</tr>
<tr>
<td>Overlay unit name</td>
<td>209</td>
</tr>
<tr>
<td>Range checking</td>
<td>209</td>
</tr>
<tr>
<td>Stack-overflow checking</td>
<td>209</td>
</tr>
<tr>
<td>Symbol reference checking</td>
<td>209</td>
</tr>
<tr>
<td>Type-checked pointers</td>
<td>209</td>
</tr>
<tr>
<td>UNDEF directive</td>
<td>209</td>
</tr>
<tr>
<td>Var-string checking</td>
<td>209</td>
</tr>
<tr>
<td>Using conditional compilation directives</td>
<td>209</td>
</tr>
<tr>
<td>Conditional symbols</td>
<td>209</td>
</tr>
</tbody>
</table>

### Chapter 3 Command-line compiler

#### Command-line compiler options

<table>
<thead>
<tr>
<th>Option</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler directive options</td>
<td>217</td>
</tr>
<tr>
<td>The switch directive option</td>
<td>217</td>
</tr>
<tr>
<td>The conditional defines option</td>
<td>217</td>
</tr>
<tr>
<td>Compiler mode options</td>
<td>217</td>
</tr>
<tr>
<td>The make (/M) option</td>
<td>217</td>
</tr>
<tr>
<td>The build all (/B) option</td>
<td>217</td>
</tr>
<tr>
<td>The find error (/F) option</td>
<td>217</td>
</tr>
<tr>
<td>The link buffer (/L) option</td>
<td>217</td>
</tr>
<tr>
<td>The quiet (/Q) option</td>
<td>217</td>
</tr>
<tr>
<td>Directory options</td>
<td>217</td>
</tr>
<tr>
<td>The TPL &amp; CFG directory (/T) option</td>
<td>217</td>
</tr>
<tr>
<td>The EXE &amp; TPU directory (/E) option</td>
<td>217</td>
</tr>
<tr>
<td>The include directories (/E) option</td>
<td>217</td>
</tr>
<tr>
<td>The unit directories (/U) option</td>
<td>217</td>
</tr>
<tr>
<td>The object files directories (/O) option</td>
<td>217</td>
</tr>
<tr>
<td>Debug options</td>
<td>217</td>
</tr>
<tr>
<td>The map file (/G) option</td>
<td>217</td>
</tr>
<tr>
<td>The debugging (/V) option</td>
<td>217</td>
</tr>
<tr>
<td>The TPC.CFG file</td>
<td>217</td>
</tr>
</tbody>
</table>

### Chapter 4 Error messages

#### Compiler error messages

<table>
<thead>
<tr>
<th>Error</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time errors</td>
<td>220</td>
</tr>
<tr>
<td>DOS errors</td>
<td>220</td>
</tr>
<tr>
<td>I/O errors</td>
<td>220</td>
</tr>
<tr>
<td>Critical Errors</td>
<td>220</td>
</tr>
</tbody>
</table>
### TABLES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1: Graph unit driver constants</td>
<td>33</td>
</tr>
<tr>
<td>1.2: Graph unit mode constants</td>
<td>34</td>
</tr>
<tr>
<td>1.3: The components of the output string</td>
<td>205</td>
</tr>
<tr>
<td>1.4: The components of the fixed-point string</td>
<td>205</td>
</tr>
<tr>
<td>3.1: Command-line options</td>
<td>228</td>
</tr>
<tr>
<td>4.1: Error message types</td>
<td>237</td>
</tr>
<tr>
<td>A.1: Editing commands</td>
<td>266</td>
</tr>
<tr>
<td>A.2: Block commands in depth</td>
<td>268</td>
</tr>
<tr>
<td>A.3: Borland-style block commands</td>
<td>269</td>
</tr>
<tr>
<td>A.4: Other editor commands in depth</td>
<td>269</td>
</tr>
<tr>
<td>A.5: Regular expression wildcards</td>
<td>270</td>
</tr>
<tr>
<td>B.1: Compiler directives</td>
<td>271</td>
</tr>
<tr>
<td>C.1: Turbo Pascal reserved words</td>
<td>275</td>
</tr>
<tr>
<td>C.2: Turbo Pascal standard directives</td>
<td>276</td>
</tr>
<tr>
<td>D.1: ASCII table</td>
<td>278</td>
</tr>
</tbody>
</table>
This manual is a reference that you can keep nearby when you’re programming. Use it when you want to

- Look up the details of a particular run-time library procedure, function, variable, type, or constant and find out how to use it
- Understand what each compiler directive does, how it works, and how to use it
- Learn how to use the command-line compiler
- Find out what an error message means
- Look up editor commands
- Look up compiler directives in a quick reference table
- Review a list of reserved words and standard compiler directives
- Look up ASCII alphanumeric characters, symbols, and control instructions

What’s in this manual?

This manual has four reference chapters and four appendixes.

**Chapter 1: Library reference** is an alphabetized lookup of all the procedures, functions, variables, types, constants, and typed constants found in the units that make up the run-time library.

**Chapter 2: Compiler directives** explains how to use the three types of compiler directives and presents a detailed, alphabetized lookup of all the directives.

**Chapter 3: Command-line compiler** explains how to use the command-line compiler.

**Chapter 4: Error messages** lists in numerical order all the error messages you might encounter and explains what they mean.
Appendix A: Editor reference explains the key combinations and commands you can use while editing your code.

Appendix B: Compiler directives quick reference lists all the compiler directives, their command-line equivalents, and brief descriptions. To find more detailed explanations, read Chapter 2.

Appendix C: Reserved words and standard directives lists all the reserved words and standard directives in Turbo Pascal.

Appendix D: ASCII characters lists all the American Standard Code for Information Interchange (ASCII) characters.

How to contact Borland

Borland offers a variety of services to answer your questions about Turbo Pascal.

Be sure to send in the registration card; registered owners are entitled to technical support and may receive information on upgrades and supplementary products.

TechFax

800-822-4269 (voice)

TechFax is a 24-hour automated service that sends free technical information to your fax machine. You can use your touch-tone phone to request up to three documents per call.

Borland Download BBS

408-439-9096 (modem) up to 9600 Baud

The Borland Download BBS has sample files, applications, and technical information you can download with your modem. No special setup is required.

Online information services

Subscribers to the CompuServe, GEnie, or BIX information services can receive technical support by modem. Use the commands in the following table to contact Borland while accessing an information service.
<table>
<thead>
<tr>
<th>Service</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompuServe</td>
<td>GO BORLAND</td>
</tr>
<tr>
<td>BIX</td>
<td>JOIN BORLAND</td>
</tr>
<tr>
<td>GEnie</td>
<td>BORLAND</td>
</tr>
</tbody>
</table>

Address electronic messages to Sysop or All. Don’t include your serial number; messages are in public view unless sent by a service’s private mail system. Include as much information on the question as possible; the support staff will reply to the message within one working day.

**Borland Technical Support**

Borland Technical Support is available weekdays from 6:00 a.m. to 5:00 p.m. Pacific time to answer technical questions about Borland products. Please call from a telephone near your computer, with the program running and the following information available:

- Product name, serial number, and version number
- Brand and model of the hardware in your system
- Operating system and version number—use the operating system’s VER command to find the version number
- Contents of your AUTOEXEC.BAT and CONFIG.SYS files (located in the root directory (\) of your computer’s boot disk)
- Contents of your WIN.INI and SYSTEM.INI files (located in your Windows directory)
- Daytime phone number where you can be reached

If the call concerns a software problem, please be able to describe the steps that will reproduce the problem.

Borland Technical Support also publishes technical information sheets on a variety of topics.

**Borland Advisor Line**

The Borland Advisor Line is a service for users who need immediate access to advice on Turbo Pascal issues.

The Advisor Line operates weekdays from 6:00 a.m. to 5:00 p.m. Pacific time. The first minute is free; each subsequent minute is $2.00.
Borland Customer Service

Borland Customer Service is available weekdays from 7:00 a.m. to 5:00 p.m. Pacific time to answer nontechnical questions about Borland products, including pricing information, upgrades, and order status.
This chapter contains a detailed description of all Turbo Pascal procedures, functions, variables, types, and constants. At the beginning of each alphabetically listed entry is the name of the unit or units containing the data element or routine, followed by the purpose, the declaration format, and any remarks specifically related to that entry. If any special restrictions apply, these are also described. The cross-referenced entries and examples provide additional information about how to use the specified entry. The first sample procedure illustrates this format.

Sample procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Description of purpose.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>How the data element or routine is declared; user-defined entries are italicized. Tables instead of declarations are used to illustrate constants whose values cannot be changed.</td>
</tr>
<tr>
<td>Remarks</td>
<td>Specific information about this entry.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Special requirements that relate to this entry.</td>
</tr>
<tr>
<td>See also</td>
<td>Related variables, constants, types, procedures, and functions that are also described in this chapter.</td>
</tr>
</tbody>
</table>
| Example     | A sample program that illustrates how to use this entry. In cases where the same function (for example, DiskFree) is included in more than one
Abs function

unit (for example, the Dos and WinDos units), separate program examples are listed only if significant differences exist between the two versions.

### Abs function

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Returns the absolute value of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function Abs(X);</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><code>X</code> is an integer-type or real-type expression. The result, of the same type as <code>X</code>, is the absolute value of <code>X</code>.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code>var</code></td>
</tr>
<tr>
<td></td>
<td><code>r: Real;</code></td>
</tr>
<tr>
<td></td>
<td><code>i: Integer;</code></td>
</tr>
<tr>
<td></td>
<td><code>begin</code></td>
</tr>
<tr>
<td></td>
<td><code>r := Abs(-2.3);</code></td>
</tr>
<tr>
<td></td>
<td><code>i := Abs(-157);</code></td>
</tr>
<tr>
<td></td>
<td><code>end.</code></td>
</tr>
</tbody>
</table>

### Addr function

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Returns the address of a specified object.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function Addr(X): Pointer;</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><code>X</code> is any variable, or a procedure or function identifier. The result is a pointer that points to <code>X</code>. Like <code>nil</code>, the result of <code>Addr</code> is assignment compatible with all pointer types.</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td><code>Ofs, Ptr, Seg</code></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code>var P: Pointer;</code></td>
</tr>
<tr>
<td></td>
<td><code>begin</code></td>
</tr>
<tr>
<td></td>
<td><code>P := Addr(P);</code></td>
</tr>
<tr>
<td></td>
<td><code>{ Now points to itself }</code></td>
</tr>
</tbody>
</table>

### Append procedure

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Opens an existing file for appending.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>procedure Append(var F: Text);</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><code>F</code> is a text file variable that must have been associated with an external file using <code>Assign</code>.</td>
</tr>
</tbody>
</table>

---

**Programmer’s Reference**
Append procedure

Append opens the existing external file with the name assigned to \( F \). An error occurs if no external file of the given name exists. If \( F \) is already open, it is closed, then reopened. The current file position is set to the end of the file.

If a \( \text{Ctrl}+\text{Z} \) (ASCII 26) is present in the last 128-byte block of the file, the current file position is set to overwrite the first \( \text{Ctrl}+\text{Z} \) in the block. In this way, text can be appended to a file that terminates with a \( \text{Ctrl}+\text{Z} \).

If \( F \) was assigned an empty name, such as Assign\((F, ''\))\), then, after the call to Append, \( F \) refers to the standard output file (standard handle number 1).

After a call to Append, \( F \) becomes write-only, and the file pointer is at end-of-file.

With \{\$I-\}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
Assign, Close, Reset, Rewrite

Example

```
var F: Text;
begin
  Assign(F, 'TEST.TXT');
  Rewrite(F); { Create new file }
  Writeln(F, 'original text');
  Close(F); { Close file, save changes }
  Append(F); { Add more text onto end }
  Writeln(F, 'appended text');
  Close(F); { Close file, save changes }
end.
```

ArcCoordsType type

Purpose
Used by GetArcCoords to retrieve information about the last call to Arc or Ellipse.

Declaration
type
ArcCoordsType = record
  X, Y: Integer;
  Xstart, Ystart: Integer;
  Xend, Yend: Integer;
end;

See also
GetArcCoords
Arc procedure

Purpose
Draws a circular arc from a starting angle to an ending angle.

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

Remarks
Draws a circular arc around \((X, Y)\), with a radius of \(Radius\) from \(StAngle\) to \(EndAngle\) in the current drawing color.

Each graphics driver contains an aspect ratio used by \(Circle\), \(Arc\), and \(PieSlice\). A start angle of 0 and an end angle of 360 draws a complete circle. The angles for \(Arc\), \(Ellipse\), and \(PieSlice\) are counterclockwise with 0 degrees at 3 o’clock, 90 degrees at 12 o’clock, and so on. Information about the last call to \(Arc\) can be retrieved by \(GetArcCoords\).

Restrictions
Must be in graphics mode.

See also
\(Circle\), \(Ellipse\), \(FillEllipse\), \(GetArcCoords\), \(GetAspectRatio\), \(PieSlice\), \(Sector\), \(SetAspectRatio\)

Example
uses Graph;
var
  Gd, Gm: Integer;
  Radius: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  for Radius := 1 to 5 do
    Arc(100, 100, 0, 90, Radius * 10);
  Readln;
  CloseGraph;
end.

ArcTan function

Purpose
Returns the arctangent of the argument.

Declaration
function ArcTan(X: Real): Real;

Remarks
\(X\) is a real-type expression. The result is the principal value, in radians, of the arctangent of \(X\).

See also
\(Cos\), \(Sin\)
### ArcTan function

Example

```pascal
var R: Real;
begin
  R := ArcTan(Pi);
end.
```

### Assign procedure

**Purpose**
Assigns the name of an external file to a file variable.

**Declaration**

```pascal
procedure Assign(var F; Name);
```

**Remarks**

- `F` is a file variable of any file type, and `Name` is a string-type expression or an expression of type `PChar` if extended syntax is enabled. All further operations on `F` operate on the external file with the file name `Name`.
- After a call to `Assign`, the association between `F` and the external file continues to exist until another `Assign` is done on `F`.
- A file name consists of a path of zero or more directory names separated by backslashes, followed by the actual file name:
  
  ```pascal
  Drive:\DirName\...\DirName\FileName
  ```
  
  If the path begins with a backslash, it starts in the root directory; otherwise, it starts in the current directory.
  
  - `Drive` is a disk drive identifier (A–Z). If `Drive` and the colon are omitted, the default drive is used.`DirName\...\DirName` is the root directory and subdirectory path to the file name. `FileName` consists of a name of up to eight characters, optionally followed by a period and an extension of up to three characters. The maximum length of the entire file name is 79 characters.
- A special case arises when `Name` is an empty string, that is, when `Length(Name)` is zero. In that case, `F` becomes associated with the standard input or standard output file. These special files allow a program to utilize the I/O redirection feature of the DOS operating system. If assigned an empty name, then after a call to `Reset(F)`, `F` refers to the standard input file, and after a call to `Rewrite(F)`, `F` refers to the standard output file.

**Restrictions**

- Never use `Assign` on an open file.

**See also**

`Append, Close, Lst, Reset, Rewrite`

**Example**

```pascal
{ Try redirecting this program from DOS to PRN, disk file, etc. }
var F: Text;
begin
  Assign(F, '');
  { Standard output }
```
Assign procedure

```pascal
Rewrite(F);
Writeln(F, 'standard output...');
Close(F);
end.
```

AssignCrt procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Associates a text file with the CRT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>procedure AssignCrt(var F: Text);</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>AssignCrt</code> works exactly like the <code>Assign</code> standard procedure except that no file name is specified. Instead, the text file is associated with the CRT. This allows faster output (and input) than would normally be possible using standard output (or input).</td>
</tr>
</tbody>
</table>
| Example | `uses Crt;
var F: Text;
begin
Write('Output to screen or printer [S, P]? '); if UpCase(ReadKey) = 'P' then Assign(F, 'PRN') else AssignCrt(F); Rewrite(F);
Writeln(F, 'Fast output via CRT routines...'); Close(F);
end.` |

Assigned function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Tests to determine if a pointer or procedural variable is <code>nil</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>function Assigned(var P): Boolean;</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>P</code> must be a variable reference of a pointer or procedural type. <code>Assigned</code> returns <code>True</code> if <code>P</code> is not <code>nil</code>, or <code>False</code> if <code>nil</code>. <code>Assigned(P)</code> corresponds to the test <code>P &lt;&gt; nil</code> for a pointer variable, and <code>@P &lt;&gt; nil</code> for a procedural variable.</td>
</tr>
</tbody>
</table>
| Example | `var P: Pointer;
begin
P := nil;
if Assigned(P) then Writeln('You won’t see this');
P := @P;
if Assigned(P) then Writeln('You’ll see this');
end.` |
Bar constants

Purpose: Constants that control the drawing of a 3-D top on a bar.

Remarks: Used by the Bar3D procedure to control whether to draw a top on 3-D bars.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopOn</td>
<td>True</td>
</tr>
<tr>
<td>TopOff</td>
<td>False</td>
</tr>
</tbody>
</table>

See also: Bar3D

Bar procedure

Purpose: Draws a bar using the current fill style and color.

Declaration: procedure Bar(XL, YL, XL2, YL2: Integer);

Remarks: Draws a filled-in rectangle (used in bar charts, for example). Uses the pattern and color defined by SetFillStyle or SetFillPattern. To draw an outlined bar, call Bar3D with a depth of zero.

Restrictions: Must be in graphics mode.

See also: Bar3D, GraphResult, SetFillStyle, SetFillPattern, SetLineStyle

Example: uses Graph;
var
  Gd, Gm: Integer;
  I, Width: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  Width := 10;
  for I := 1 to 5 do
    Bar(I * Width, I * 10, Succ(I) * Width, 200);
  Readln;
  CloseGraph;
end.
Bar3D procedure

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

Declaration
procedure Bar3D(X1, Y1, X2, Y2: Integer; Depth: Word; Top: Boolean);

Remarks
Draws a filled-in, three-dimensional bar using the pattern and color defined by SetFillStyle or SetFillPattern. 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);
```

Restrictions
Must be in graphics mode.

See also
Bar, GraphResult, SetFillPattern, SetFillStyle, SetLineStyle

Example
```
uses Graph;
var
  Gd, Gm: Integer;
  Y0, Y1, Y2, X1, X2: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Y0 := 10;
  Y1 := 60;
  Y2 := 110;
  X1 := 10;
  X2 := 50;
  Bar3D(X1, Y0, X2, Y1, 10, TopOn);
  Bar3D(X1, Y1, X2, Y2, 10, TopOff);
  Readln;
  CloseGraph;
end.
```

BitBlt operators

Purpose
BitBlt operators used with PutImage and SetWriteMode.

Remarks
The following constant values represent the indicated logical operations.
BitBlt operators

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyPut</td>
<td>0 (mov)</td>
</tr>
<tr>
<td>XORPut</td>
<td>1 (xor)</td>
</tr>
</tbody>
</table>

These BitBlt constants are used by PutImage only:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrPut</td>
<td>2 (or)</td>
</tr>
<tr>
<td>AndPut</td>
<td>3 (and)</td>
</tr>
<tr>
<td>NotPut</td>
<td>4 (not)</td>
</tr>
</tbody>
</table>

BlockRead procedure

Purpose
Reads one or more records into a variable.

Declaration

```
procedure BlockRead(var F: file; var Buf; Count: Word [ ; var Result: Word ]);```

Remarks

- **F** is an untyped file variable, **Buf** is any variable, **Count** is an expression of type **Word**, and **Result** is a variable of type **Word**.

BlockRead reads **Count** or fewer records from the file **F** into memory, starting at the first byte occupied by **Buf**. The actual number of complete records read (less than or equal to **Count**) is returned in the optional parameter **Result**. If **Result** is not specified, an I/O error occurs if the number read is not equal to **Count**.

The entire transferred block occupies at most **Count** * **RecSize** bytes, where **RecSize** is the record size specified when the file was opened (or 128 if the record size was unspecified). An error occurs if **Count** * **RecSize** is greater than 65,535 (64K).

**Result** is an optional parameter. If the entire block was transferred, **Result** will be equal to **Count** on return. Otherwise, if **Result** is less than **Count**, the end of the file was reached before the transfer was completed. In that case, if the file's record size is greater than 1, **Result** returns the number of complete records read; that is, a possible last partial record is not included in **Result**.

The current file position is advanced by **Result** records as an effect of **BlockRead**.

With {$I-}, IOResult returns 0 if the operation succeeded; otherwise, it returns a nonzero error code.

Restrictions
File must be open.

See also
BlockWrite
**Example**

```pascal
program CopyFile;
{ Simple, fast file copy program with NO error-checking }
var
    FromF, ToF: file;
    NumRead, NumWritten: Word;
    Buf: array[1..2048] of Char;
begin
    Assign(FromF, ParamStr(1));       { Open input file }
    Reset(FromF, 1);                  { Record size = 1 }
    Assign(ToF, ParamStr(2));        { Open output file }
    Rewrite(ToF, 1);                  { Record size = 1 }
    Writeln('Copying ', FileSize(FromF), ' bytes...');
    repeat
        BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
        BlockWrite(ToF, Buf, NumRead, NumWritten);
    until (NumRead = 0) or (NumWritten <> NumRead);
    Close(FromF);
    Close(ToF);
end.
```

---

**BlockWrite procedure**

**Purpose**
Writes one or more records from a variable.

**Declaration**

```pascal
procedure BlockWrite(var F: file; var Buf; Count: Word
                     ; var Result: Word );
```

**Remarks**

`F` is an untyped file variable, `Buf` is any variable, `Count` is an expression of type `Word`, and `Result` is a variable of type `Word`.

`BlockWrite` writes `Count` or fewer records to the file `F` from memory, starting at the first byte occupied by `Buf`. The actual number of complete records written (less than or equal to `Count`) is returned in the optional parameter `Result`. If `Result` is not specified, an I/O error occurs if the number written is not equal to `Count`.

The entire block transferred occupies at most `Count * RecSize` bytes, where `RecSize` is the record size specified when the file was opened (or 128 if the record size was unspecified). An error occurs if `Count * RecSize` is greater than 65,535 (64K).

`Result` is an optional parameter. If the entire block was transferred, `Result` will be equal to `Count` on return. Otherwise, if `Result` is less than `Count`, the disk became full before the transfer was completed. In that case, if the file's record size is greater than 1, `Result` returns the number of complete records written.
The current file position is advanced by \textit{Result} records as an effect of the \textit{BlockWrite}.

With \{\$I+\}, \textit{IOResult} returns 0 if the operation succeeded; otherwise, it returns a nonzero error code.

\textbf{Restrictions} \hspace{1cm} File must be open.

\textbf{See also} \hspace{1cm} \textit{BlockRead}

\textbf{Example} \hspace{1cm} See example for \textit{BlockRead}.

\textbf{Break procedure} \hspace{1cm} \textbf{System}

\begin{itemize}
  \item \textbf{Purpose} \hspace{1cm} Terminates a \texttt{for}, \texttt{while}, or \texttt{repeat} statement.
  \item \textbf{Declaration} \hspace{1cm} \texttt{procedure Break;}
  \item \textbf{Remarks} \hspace{1cm} \textit{Break} exits the innermost enclosing \texttt{for}, \texttt{while}, or \texttt{repeat} statement immediately. \textit{Break} is analogous to a \texttt{goto} statement addressing a label just after the end of the innermost enclosing repetitive statement. The compiler reports an error if \textit{Break} is not enclosed by a \texttt{for}, \texttt{while}, or \texttt{repeat} statement.
  \item \textbf{See also} \hspace{1cm} \textit{Continue, Exit, Halt}
  \item \textbf{Example} \hspace{1cm} \begin{verbatim}
  var S: string;
  begin
    while True do
      begin
        Readln(S);
        if S = '' then Break;
        Writeln(S);
      end;
  end.
\end{verbatim}
\end{itemize}

\textbf{ChDir procedure} \hspace{1cm} \textbf{System}

\begin{itemize}
  \item \textbf{Purpose} \hspace{1cm} Changes the current directory.
  \item \textbf{Declaration} \hspace{1cm} \texttt{procedure ChDir(S: String);}
  \item \textbf{Remarks} \hspace{1cm} The string-type expression changes the current directory to the path specified by \texttt{S}. If \texttt{S} specifies a drive letter, the current drive is also changed.
\end{itemize}
ChDir procedure

With {$I-}, \textit{IOResult} returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also \textit{GetDir}, \textit{MkDir}, \textit{RmDir}. \textit{SetCurDir} performs the same function, but it takes a null-terminated string as an argument rather than a Pascal-style string.

Example

\begin{verbatim}
begin
  {$I-}
  { Get directory name from command line }
  ChDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Cannot find directory');
end.
\end{verbatim}

CheckBreak variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Enables and disables checks for \textit{Ctrl+Break}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>\texttt{var CheckBreak: Boolean;}</td>
</tr>
<tr>
<td>Remarks</td>
<td>When \textit{CheckBreak} is \textit{True}, pressing \textit{Ctrl+Break} aborts the program when it next writes to the display. When \textit{CheckBreak} is \textit{False}, pressing \textit{Ctrl+Break}, has no effect. \textit{CheckBreak} is \textit{True} by default. (At run time, \texttt{Crt} stores the old \textit{Ctrl+Break} interrupt vector, $1B, in a global pointer variable called \texttt{SaveInt1B}.)</td>
</tr>
<tr>
<td>See also</td>
<td>\textit{KeyPressed}, \textit{ReadKey}, \texttt{SaveInt1B}</td>
</tr>
</tbody>
</table>

CheckEOF variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Enables and disables the end-of-file character.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>\texttt{var CheckEOF: Boolean;}</td>
</tr>
<tr>
<td>Remarks</td>
<td>When \textit{CheckEOF} is \textit{True}, an end-of-file character is generated if you press \textit{Ctrl+Z} while reading from a file assigned to the screen. When \textit{CheckEOF} is \textit{False}, pressing \textit{Ctrl+Z} has no effect. \textit{CheckEOF} is \textit{False} by default.</td>
</tr>
</tbody>
</table>

CheckSnow variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Enables and disables “snow-checking” on CGA video adapters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>\texttt{var CheckSnow: Boolean;}</td>
</tr>
</tbody>
</table>
CheckSnow variable

Remarks
On most CGAs, interference will result if characters are stored in video memory outside the horizontal retrace intervals. This does not occur with monochrome adapters, EGAs, or VGAs.

When a color text mode is selected, CheckSnow is set to True, and direct video-memory writes will occur only during the horizontal retrace intervals. If you are running on a newer CGA, you might want to set CheckSnow to False at the beginning of your program and after each call to TextMode. This will disable snow-checking, resulting in significantly higher output speeds.

Restrictions
CheckSnow has no effect when DirectVideo is False.

See also
DirectVideo

Chr function

Purpose
Returns a character with a specified ordinal number.

Declaration
function Chr(X: Byte): Char;

Remarks
Returns the character with the ordinal value (ASCII value) of the byte-type expression, X.

See also
Ord

Example
var I: Integer;
begin
  for I := 32 to 255 do Write(Chr(I));
end.

Circle procedure

Purpose
Draws a circle using (X, Y) as the center point.

Declaration
procedure Circle(X, Y: Integer; Radius: Word);

Remarks
Draws a circle in the current color set by SetColor. Each graphics driver contains an aspect ratio used by Circle, Arc, and PieSlice.

Restrictions
Must be in graphics mode.

See also
Arc, Ellipse, FillEllipse, GetArcCoords, GetAspectRatio, PieSlice, Sector, SetAspectRatio
Circle procedure

Example
uses Graph;
var
  Gd, Gm: Integer;
  Radius: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  for Radius := 1 to 5 do
    Circle(100, 100, Radius * 10);
  Readln;
  CloseGraph;
end.

ClearDevice procedure

Purpose
Clears the graphics screen and prepares it for output.

Declaration
procedure ClearDevice;

Remarks
ClearDevice moves the current pointer to (0, 0), clears the screen using the background color set by SetBkColor, and prepares it for output.

Restrictions
Must be in graphics mode.

See also
ClearViewport, CloseGraph, GraphDefaults, InitGraph, RestoreCrtMode, SetGraphMode

Example
uses Crt, Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Randomize;
  repeat
    LineTo(Random(200), Random(200));
  until KeyPressed;
  ClearDevice;
  Readln;
  CloseGraph;
end.
ClearViewPort procedure

Purpose
Clears the current viewport.

Declaration
procedure ClearViewPort;

Remarks
Sets the fill color to the background color (Palette[0]) and moves the current pointer to (0, 0).

Restrictions
Must be in graphics mode.

See also
ClearDevice, GetViewSettings, SetViewPort

Example
uses Graph;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, ");
if GraphResult <> grOk then
   Halt(1);
Rectangle(19, 19, GetMaxX - 19, GetMaxY - 19);
SetViewPort(20, 20, GetMaxX - 20, GetMaxY - 20, ClipOn);
OutTextXY(0, 0, '<ENTER> clears viewport:');
Readln;
ClearViewPort;
OutTextXY(0, 0, '<ENTER> to quit:');
Readln;
CloseGraph;
end.

Clipping constants

Purpose
Constants that control clipping: used with SetViewPort.

Remarks
With clipping on, graphics output is clipped at the viewport boundaries:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClipOn</td>
<td>True</td>
</tr>
<tr>
<td>ClipOff</td>
<td>False</td>
</tr>
</tbody>
</table>

See also
SetViewPort
Close procedure

Purpose
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 DOS file handle for reuse.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
Append, Assign, Reset, Rewrite

Example
var F: file;
begin
Assign(F, 'AUTOEXEC.BAT');
Reset(F, 1);
Writeln('File size = ', FileSize(F));
Close(F);
end.

CloseGraph procedure

Purpose
Shuts down the graphics system.

Declaration
procedure CloseGraph;

Remarks
CloseGraph restores the original screen mode before graphics was initialized and frees the memory allocated on the heap for the graphics scan buffer. CloseGraph also deallocates driver and font memory buffers if they were allocated by calls to GraphGetMem and GraphFreeMem.

Restrictions
Must be in graphics mode.

See also
DetectGraph, GetGraphMode, InitGraph, RestoreCrtMode, SetGraphMode

Example
uses Graph;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, '');
if GraphResult <> grOk then
  Halt(1);
Line(0, 0, GetMaxX, GetMaxY);
CloseGraph procedure

Readln;
CloseGraph;
end.  { Shut down graphics }

ClrEol procedure

Purpose: Clears all characters from the cursor position to the end of the line without moving the cursor.

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. The following program lines define a text window and clear the current line from the cursor position (1, 1) to the right edge of the active window (60, 1).

Window(1, 1, 60, 20);
ClrEol;

See also: ClrScr, Window

Example: uses Crt;
begin
  TextBackground(LightGray);
  ClrEol;  { Changes cleared columns to LightGray background }
end.

ClrScr procedure

Purpose: Clears the active window and places the cursor in 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 procedure

ClrScr is window-relative. The following program lines define a text window and clear a 60x20 rectangle beginning at (1, 1).

```plaintext
Window(1, 1, 60, 20);
ClrScr;
```

See also  
ClrEol, Window

Example  
uses Crt;
begin
  TextBackground(LightGray);
  ClrScr;  { Changes entire window to LightGray background }
end.

Color constants

Color constants used by SetPalette and SetAllPalette.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>2</td>
</tr>
<tr>
<td>Cyan</td>
<td>3</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Magenta</td>
<td>5</td>
</tr>
<tr>
<td>Brown</td>
<td>6</td>
</tr>
<tr>
<td>LightGray</td>
<td>7</td>
</tr>
<tr>
<td>DarkGray</td>
<td>8</td>
</tr>
<tr>
<td>LightBlue</td>
<td>9</td>
</tr>
<tr>
<td>LightGreen</td>
<td>10</td>
</tr>
<tr>
<td>LightCyan</td>
<td>11</td>
</tr>
<tr>
<td>LightRed</td>
<td>12</td>
</tr>
<tr>
<td>LightMagenta</td>
<td>13</td>
</tr>
<tr>
<td>Yellow</td>
<td>14</td>
</tr>
<tr>
<td>White</td>
<td>15</td>
</tr>
</tbody>
</table>

See also  
SetAllPalette, SetPalette, SetColor

Color constants for SetRGBPalette

Constants that can be used with SetRGBPalette to select the standard EGA colors on an IBM 8514 graphics adapter.

Remarks  
The following EGA color constant values are defined:
Color constants for SetRGBPalette

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGABlack</td>
<td>0 (dark colors)</td>
</tr>
<tr>
<td>EGABlue</td>
<td>1</td>
</tr>
<tr>
<td>EGAGreen</td>
<td>2</td>
</tr>
<tr>
<td>EGACyan</td>
<td>3</td>
</tr>
<tr>
<td>EGARed</td>
<td>4</td>
</tr>
<tr>
<td>EGAMagenta</td>
<td>5</td>
</tr>
<tr>
<td>EGABrown</td>
<td>20</td>
</tr>
<tr>
<td>EGAGray</td>
<td>7</td>
</tr>
<tr>
<td>EGADarkGray</td>
<td>56 (light colors)</td>
</tr>
<tr>
<td>EGABlue</td>
<td>57</td>
</tr>
<tr>
<td>EGAGreen</td>
<td>58</td>
</tr>
<tr>
<td>EGA Cyan</td>
<td>59</td>
</tr>
<tr>
<td>EGA Red</td>
<td>60</td>
</tr>
<tr>
<td>EGA Magenta</td>
<td>61</td>
</tr>
<tr>
<td>EGA Yellow</td>
<td>62</td>
</tr>
<tr>
<td>EGA White</td>
<td>63</td>
</tr>
</tbody>
</table>

See also SetRGBPalette

Concat function

**Purpose**
Concatenates a sequence of strings.

**Declaration**

```pascal
function Concat(S1, S2, ..., SN: String): String;
```

**Remarks**
Each parameter is a string-type expression. The result is the concatenation of all the string parameters. If the resulting string is longer than 255 characters, it is truncated after the 255th character. Using the plus (+) operator returns the same result as using the Concat function:

S := 'ABC' + 'DEF';

**See also** Copy, Delete, Insert, Length, Pos

**Example**

```pascal
var S: String;
begin
  S := Concat('ABC', 'DEF');
  { 'ABCDEF' }
end.
```

Continue procedure

**Purpose**
Continues a **for**, **while**, or **repeat** statement.

**Declaration**

```pascal
procedure Continue;
```
Continue procedure

Remarks  
Continue 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.

See also  Break, Exit, Halt

Example  

```pascal
var
  I: Integer;
  Name: string[79];
  F: file;
begin
  for I := 1 to ParamCount do
  begin
    Name := ParamStr(I);
    Assign(F, Name);
    {$I-}
    Reset(F, 1);
    {$I+}
    if IOResult <> 0 then
    begin
      Writeln('File not found: ' , Name);
      Continue;
    end;
    Writeln(Name, ': ', FileSize(F), ' bytes');
    Close(F);
  end;
end.
```

Copy function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns a substring of a string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Copy(S: String; Index: Integer; Count: Integer): String;</td>
</tr>
<tr>
<td>Remarks</td>
<td>S is a string-type expression. Index and Count are integer-type expressions. Copy returns a string containing Count characters starting with the Indexth character in S. If Index is larger than the length of S, Copy returns an empty string. If Count specifies more characters than remain starting at the Indexth position, only the remainder of the string is returned.</td>
</tr>
<tr>
<td>See also</td>
<td>Concat, Delete, Insert, Length, Pos</td>
</tr>
</tbody>
</table>
Copy function

Example

```pascal
var S: string;
begin
  S := 'ABCDEF';
  S := Copy(S, 2, 3);
end.
```

Cos function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the cosine of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Cos(X: Real): Real;</td>
</tr>
<tr>
<td>Remarks</td>
<td>X is a real-type expression. The result is the cosine of ( X ) where ( X ) represents an angle in radians.</td>
</tr>
<tr>
<td>See also</td>
<td>ArcTan, Sin</td>
</tr>
</tbody>
</table>
| Example          | var R: Real;
                   begin
                   R := Cos(Pi);
                   end.                           |

CreateDir procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creates a new subdirectory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure CreateDir(Dir: PChar);</td>
</tr>
<tr>
<td>Remarks</td>
<td>The subdirectory to be created is specified in ( \text{Dir} ). Errors are reported in \text{DosError}. \text{MkDir} performs the same function as \text{CreateDir}, but it takes a Pascal-style string as an argument rather than a null-terminated string.</td>
</tr>
<tr>
<td>See also</td>
<td>\text{GetCurDir}, \text{SetCurDir}, \text{RemoveDir}</td>
</tr>
</tbody>
</table>

Crt mode constants

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Used to represent Crt text and line modes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remarks</td>
<td>( \text{BW40}, \text{C040}, \text{BW80}, ) and ( \text{C080} ) represent the four color text modes supported by the IBM PC Color/Graphics Adapter (CGA). The ( \text{Mono} ) constant represents the single black-and-white text mode supported by the IBM PC Monochrome Adapter. ( \text{Font8x8} ) represents EGA/VGA 43- and 50-line modes and is used with ( \text{C080} ) or ( \text{LastMode} ). ( \text{LastMode} ) returns to the last active text mode after using graphics.</td>
</tr>
</tbody>
</table>

Chapter 1, Library reference
Crt mode constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW40</td>
<td>0</td>
<td>40x25 B/W on color adapter</td>
</tr>
<tr>
<td>BW80</td>
<td>2</td>
<td>80x25 B/W on color adapter</td>
</tr>
<tr>
<td>Mono</td>
<td>7</td>
<td>80x25 B/W on monochrome adapter</td>
</tr>
<tr>
<td>C040</td>
<td>1</td>
<td>40x25 color on color adapter</td>
</tr>
<tr>
<td>C080</td>
<td>3</td>
<td>80x25 color on color adapter</td>
</tr>
<tr>
<td>Font8x8</td>
<td>256</td>
<td>For EGA/VGA 43 and 50 line</td>
</tr>
<tr>
<td>C40</td>
<td>C040</td>
<td>For Turbo Pascal 3.0 compatibility</td>
</tr>
<tr>
<td>C80</td>
<td>C080</td>
<td>For Turbo Pascal 3.0 compatibility</td>
</tr>
</tbody>
</table>

See also TextMode

CSeg function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the current value of the CS register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function CSeg: Word;</td>
</tr>
<tr>
<td>Remarks</td>
<td>The result of type Word is the segment address of the code segment within which CSeg was called.</td>
</tr>
<tr>
<td>See also</td>
<td>DSeg, SSeg</td>
</tr>
</tbody>
</table>

DateTime type

<table>
<thead>
<tr>
<th>Purpose</th>
<th>UnpackTime and PackTime use variables of DateTime type to examine and construct 4-byte, packed date-and-time values for the GetFTime, SetFTime, FindFirst, and FindNext procedures.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>type</td>
</tr>
<tr>
<td></td>
<td>DateTime = record</td>
</tr>
<tr>
<td></td>
<td>Year, Month, Day, Hour, Min, Sec: Word;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Valid ranges are Year 1980..2099, Month 1..12, Day 1..31, Hour 0..23, Min 0..59, and Sec 0..59.</td>
</tr>
<tr>
<td>See also</td>
<td>FindFirst, FindNext, GetFTime, SetFTime</td>
</tr>
</tbody>
</table>
Dec procedure

**Purpose**
Decrement a variable.

**Declaration**

```plaintext
procedure Dec(var X [; N: Longint]);
```

**Remarks**

X is an ordinal-type variable or a variable of type `PChar` if the extended syntax is enabled, and N is an integer-type expression. X is decremented by 1, or by N if N is specified; that is, `Dec(X)` corresponds to `X := X - 1`, and `Dec(X, N)` corresponds to `X := X - N`.

`Dec` generates optimized code and is especially useful in a tight loop.

**See also**

`Inc`, `Pred`, `Succ`

**Example**

```plaintext
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Dec(IntVar);                // IntVar := IntVar - 1
  Dec(LongintVar, 5);        // LongintVar := LongintVar - 5
end.
```

Delay procedure

**Purpose**
Delays a specified number of milliseconds.

**Declaration**

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

**Remarks**

Ms specifies the number of milliseconds to wait.

`Delay` is an approximation, so the delay period will not last exactly Ms milliseconds.

Delete procedure

**Purpose**
Deletes a substring from a string.

**Declaration**

```plaintext
procedure Delete(var S: String; Index: Integer; Count: Integer);
```

**Remarks**

S is a string-type variable. `Index` and `Count` are integer-type expressions. `Delete` deletes `Count` characters from S starting at the `Index`th position. If `Index` is larger than the length of S, no characters are deleted. If `Count`
Delete procedure

specifies more characters than remain starting at the Indexth position, the remainder of the string is deleted.

See also Concat, Copy, Insert, Length, Pos

DelLine procedure

Purpose Deletes the line containing the cursor.

Declaration procedure DelLine;

Remarks The line containing the cursor is deleted, and all lines below are moved one line up (using the BIOS scroll routine). A new line is added at the bottom.

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

Example DelLine is window-relative. The following example will delete the first line in the window, which is the tenth line on the screen.

Window(1, 10, 60, 20);
DelLine;

See also InsLine, Window

DetectGraph procedure

Purpose Checks the hardware and determines which graphics driver and mode to use.

Declaration procedure DetectGraph(var GraphDriver, GraphMode: Integer);

Remarks Returns the detected driver and mode value that can be passed to InitGraph, which will then load the correct driver. If no graphics hardware was detected, the GraphDriver parameter and GraphResult returns a value of grNotDetected. See page 33 for a list of driver and mode constants.

Unless instructed otherwise, InitGraph calls DetectGraph, finds and loads the correct driver, and initializes the graphics system. The only reason to call DetectGraph directly is to override the driver that DetectGraph recommends. The example that follows identifies the system as a 64K or 256K EGA, and loads the CGA driver instead. When you pass InitGraph a GraphDriver other than Detect, you must also pass in a valid GraphMode for the driver requested.
**DetectGraph procedure**

**Restrictions**  You should not use `DetectGraph` (or `Detect` with `InitGraph`) with the IBM 8514 unless you want the emulated VGA mode.

**See also**  `CloseGraph`, `Driver and mode`, `GraphResult`, `InitGraph`

**Example**

```pascal
uses Graph;
var GraphDriver, GraphMode: Integer;
begin
  DetectGraph(GraphDriver, GraphMode);
  if (GraphDriver = EGA) or
     (GraphDriver = EGA64) then
    begin
      GraphDriver := CGA;
      GraphMode := CGAHi;
    end;
  InitGraph(GraphDriver, GraphMode, '');
  if GraphResult <> grOk then
    Halt(1);
  Line(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.
```

**DirectVideo variable**

**Purpose**  Enables and disables direct memory access for `Write` and `Writeln` statements that output to the screen.

**Declaration**  `var DirectVideo: Boolean;`

**Remarks**  When `DirectVideo` is `True`, `Write` and `Writeln` to files associated with the CRT will store characters directly in video memory instead of calling the BIOS to display them. When `DirectVideo` is `False`, all characters are written through BIOS calls, which is a significantly slower process.

`DirectVideo` always defaults to `True`. If, for some reason, you want characters displayed through BIOS calls, set `DirectVideo` to `False` at the beginning of your program and after each call to `TextMode`.

**See also**  `CheckSnow`
DiskFree function  Dos, WinDos

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

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

Remarks: A Drive of 0 indicates the default drive, 1 indicates drive A, 2 indicates B, and so on. DiskFree returns -1 if the drive number is invalid.

See also: DiskSize, GetDir

Example:
```delphi
uses Dos;
begin
  Writeln(DiskFree(0) div 1024, ' Kbytes free');
end.
```

DiskSize function  Dos, WinDos

Purpose: Returns the total size in bytes on a specified disk drive.

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

Remarks: A Drive of 0 indicates the default drive, 1 indicates drive A, 2 indicates B, and so on. DiskSize returns -1 if the drive number is invalid.

See also: DiskFree, GetDir

Example:
```delphi
uses Dos;
begin
  Writeln(DiskSize(0) div 1024, ' Kbytes capacity');
end.
```

Dispose procedure  System

Purpose: Disposes of a dynamic variable.

Declaration: procedure Dispose(var P: Pointer [, Destructor ]); 

Remarks: P is a variable of any pointer type previously assigned by the New procedure or assigned a meaningful value by an assignment statement. Dispose destroys the variable referenced by P and returns its memory region to the heap. After a call to Dispose, the value of P becomes undefined and it is an error to subsequently reference P.

Dispose allows a destructor call as a second parameter, for disposing a dynamic object type variable. In this case, P is a pointer variable pointing...
Dispose procedure

to an object type, and Destructor is a call to the destructor of that object type.

**Restrictions**

If \( P \) does not point to a memory region in the heap, a run-time error occurs.

For a complete discussion of this topic, see “The heap manager” in Chapter 19 of the Language Guide.

**See also**

FreeMem, GetMem, New

**Example**

```pascal
type Str18 = string[18];
var P: ^Str18;
begin
  New(P);
  P := 'Now you see it...';
  Dispose(P);
end.
```

---

DosError variable

**Dos, WinDos**

**Purpose**

Used by many Dos and WinDos routines to report errors.

**Declaration**

```pascal
var DosError: Integer;
```

**Remarks**

The values stored in DosError are DOS error codes. A value of 0 indicates no error; other possible error codes include the following:

<table>
<thead>
<tr>
<th>DOS error code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>File not found</td>
</tr>
<tr>
<td>3</td>
<td>Path not found</td>
</tr>
<tr>
<td>5</td>
<td>Access denied</td>
</tr>
<tr>
<td>6</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>8</td>
<td>Not enough memory</td>
</tr>
<tr>
<td>10</td>
<td>Invalid environment</td>
</tr>
<tr>
<td>11</td>
<td>Invalid format</td>
</tr>
<tr>
<td>18</td>
<td>No more files</td>
</tr>
</tbody>
</table>

See Chapter 4, “Error messages,” for a detailed description of DOS error messages.

**See also**

CreateDir, Exec, FindFirst, FindNext, GetCurDir, GetFAttr, GetFTime, RemoveDir, SetCurDir, SetFAttr, SetFTime
DosExitCode function

**Purpose**
Returns the exit code of a subprocess.

**Declaration**
function DosExitCode: Word;

**Remarks**
The low byte is the code sent by the terminating process. The high byte is set to
- 0 for normal termination
- 1 if terminated by Ctrl+C
- 2 if terminated due to a device error
- 3 if terminated by the Keep procedure

**See also**
Exec, Keep

DosVersion function

**Purpose**
Returns the DOS version number.

**Declaration**
function DosVersion: Word;

**Remarks**
DosVersion returns the DOS version number. The low byte of the result is the major version number, and the high byte is the minor version number. For example, DOS 3.20 returns 3 in the low byte, and 20 in the high byte.

**Example**
uses Dos;
var Ver: Word;
begin
  Ver := DosVersion;
  Writeln('This is DOS version ', Lo(Ver), '.', Hi(Ver));
end.

**See also**
Hi, Lo

DrawPoly procedure

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

**Declaration**
procedure DrawPoly(NumPoints: Word; var PolyPoints);

**Remarks**
PolyPoints is an untyped parameter that contains the coordinates of each intersection in the polygon. NumPoints specifies the number of coordinates in PolyPoints. A coordinate consists of two words, an X and a Y value.
DrawPoly procedure

DrawPoly uses the current line style and color. Use SetWriteMode to determine whether the polygon is copied to or XORed to the screen.

Note that in order to draw a closed figure with \(N\) vertices, you must pass \(N + 1\) coordinates to DrawPoly, where

\[\text{PolyPoints}[N + 1] = \text{PolyPoints}[1]\]

In order to draw a triangle, for example, four coordinates must be passed to DrawPoly.

**Restrictions**
Must be in graphics mode.

**See also**
FillPoly, GetLineSettings, GraphResult, SetColor, SetLineStyle, SetWriteMode

**Example**

```pascal
uses Graph;
const
  Triangle: array[1..4] of PointType = ((X: 50; Y: 100), (X: 100; Y: 100),
    (X: 150; Y: 150), (X: 50; Y: 100));
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  DrawPoly(SizeOf(Triangle) div SizeOf(PointType), Triangle); { 4 }
  Readln;
  CloseGraph;
end.
```

**Driver and mode constants**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Used with routines that call graphics drivers and color palettes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remarks</td>
<td>The following tables list graphics drivers and color palettes.</td>
</tr>
</tbody>
</table>

### Table 1.1
**Graph unit driver constants**

<table>
<thead>
<tr>
<th>Driver Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect</td>
<td>0</td>
<td>Requests autodetection</td>
</tr>
<tr>
<td>CGA</td>
<td>1</td>
<td>CGA mode</td>
</tr>
<tr>
<td>MCGA</td>
<td>2</td>
<td>MCGA mode</td>
</tr>
<tr>
<td>EGA</td>
<td>3</td>
<td>EGA mode</td>
</tr>
<tr>
<td>EGA64</td>
<td>4</td>
<td>EGA64 mode</td>
</tr>
<tr>
<td>EGAMono</td>
<td>5</td>
<td>EGAMono mode</td>
</tr>
<tr>
<td>IBM8514</td>
<td>6</td>
<td>IBM8514 mode</td>
</tr>
<tr>
<td>HercMono</td>
<td>7</td>
<td>HercMono mode</td>
</tr>
</tbody>
</table>
Driver and mode constants

Table 1.1: Graph unit driver constants (continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Column x Row</th>
<th>Colors</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT400</td>
<td>8</td>
<td></td>
<td>ATT400 mode</td>
<td></td>
</tr>
<tr>
<td>VGA</td>
<td>9</td>
<td></td>
<td>VGA mode</td>
<td></td>
</tr>
<tr>
<td>PC3270</td>
<td>10</td>
<td></td>
<td>PC3270 mode</td>
<td></td>
</tr>
<tr>
<td>CurrentDriver</td>
<td>-128</td>
<td></td>
<td>Passed to GetModeRange</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.2: Graph unit mode constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
<th>Column x Row</th>
<th>Palette</th>
<th>Colors</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT400C0</td>
<td>0</td>
<td>320x200</td>
<td>0</td>
<td>LightGreen, LightRed, Yellow</td>
<td>1</td>
</tr>
<tr>
<td>ATT400C1</td>
<td>1</td>
<td>320x200</td>
<td>1</td>
<td>LightCyan, LightMagenta, White</td>
<td>1</td>
</tr>
<tr>
<td>ATT400C2</td>
<td>2</td>
<td>320x200</td>
<td>2</td>
<td>Green, Red, Brown</td>
<td>1</td>
</tr>
<tr>
<td>ATT400C3</td>
<td>3</td>
<td>320x200</td>
<td>3</td>
<td>Cyan, Magenta, LightGray</td>
<td>1</td>
</tr>
<tr>
<td>ATT400Med</td>
<td>4</td>
<td>640x200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATT400Hi</td>
<td>5</td>
<td>640x400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGAC0</td>
<td>0</td>
<td>320x200</td>
<td>0</td>
<td>LightGreen, LightRed, Yellow</td>
<td>1</td>
</tr>
<tr>
<td>CGAC1</td>
<td>1</td>
<td>320x200</td>
<td>1</td>
<td>LightCyan, LightMagenta, White</td>
<td>1</td>
</tr>
<tr>
<td>CGAC2</td>
<td>2</td>
<td>320x200</td>
<td>2</td>
<td>Green, Red, Brown</td>
<td>1</td>
</tr>
<tr>
<td>CGAC3</td>
<td>3</td>
<td>320x200</td>
<td>3</td>
<td>Cyan, Magenta, LightGray</td>
<td>1</td>
</tr>
<tr>
<td>CGAHi</td>
<td>4</td>
<td>640x200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGAHi</td>
<td>5</td>
<td>640x400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EGALo</td>
<td>0</td>
<td>640x200</td>
<td></td>
<td>16 color</td>
<td>4</td>
</tr>
<tr>
<td>EGAHi</td>
<td>1</td>
<td>640x350</td>
<td></td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td>EGA64Lo</td>
<td>0</td>
<td>640x200</td>
<td></td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA64Hi</td>
<td>1</td>
<td>640x350</td>
<td></td>
<td>4 color</td>
<td>1</td>
</tr>
<tr>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td></td>
<td>64K on card, 256K on card</td>
<td>1</td>
</tr>
<tr>
<td>HercMonoHi</td>
<td>0</td>
<td>720x348</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>IBM8514Lo</td>
<td>0</td>
<td>640x480</td>
<td></td>
<td>256 colors</td>
<td>4</td>
</tr>
<tr>
<td>IBM8514Hi</td>
<td>1</td>
<td>1024x768</td>
<td></td>
<td>256 colors</td>
<td>2</td>
</tr>
<tr>
<td>MCGAC0</td>
<td>0</td>
<td>320x200</td>
<td>0</td>
<td>LightGreen, LightRed, Yellow</td>
<td>1</td>
</tr>
<tr>
<td>MCGAC1</td>
<td>1</td>
<td>320x200</td>
<td>1</td>
<td>LightCyan, LightMagenta, White</td>
<td>1</td>
</tr>
<tr>
<td>MCGAC2</td>
<td>2</td>
<td>320x200</td>
<td>2</td>
<td>Green, Red, Brown</td>
<td>1</td>
</tr>
<tr>
<td>MCGAC3</td>
<td>3</td>
<td>320x200</td>
<td>3</td>
<td>Cyan, Magenta, LightGray</td>
<td>1</td>
</tr>
<tr>
<td>MCGAMed</td>
<td>4</td>
<td>640x200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCGAHi</td>
<td>5</td>
<td>640x480</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC3270Hi</td>
<td>0</td>
<td>720x350</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGALo</td>
<td>0</td>
<td>640x200</td>
<td></td>
<td>16 color</td>
<td>4</td>
</tr>
<tr>
<td>VGAMed</td>
<td>1</td>
<td>640x350</td>
<td></td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td>VGAHi</td>
<td>2</td>
<td>640x480</td>
<td></td>
<td>16 color</td>
<td>1</td>
</tr>
</tbody>
</table>

See also DetectGraph, GetModeRange, InitGraph
DSeg function

Purpose
Returns the current value of the DS register.

Declaration
function DSeg: Word;

Remarks
The result of type Word is the segment address of the data segment.

See also CSeg, SSeg

Ellipse procedure

Purpose
Draws an elliptical arc from start angle to end angle, using (X, Y) as the center point.

Declaration
procedure Ellipse(X, Y: Integer; StAngle, EndAngle: Word;
                    XRadii, YRadii: Word);

Remarks
Draws an elliptical arc in the current color using (X, Y) as a center point, and XRadii and YRadii as the horizontal and vertical axes travelling from StAngle to EndAngle.

A start angle of 0 and an end angle of 360 draws a complete oval. The angles for Arc, Ellipse, and PieSlice are counterclockwise with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on. Information about the last call to Ellipse can be retrieved by GetArcCoords.

Restrictions
Must be in graphics mode.

See also Arc, Circle, FillEllipse, GetArcCoords, GetAspectRatio, PieSlice, Sector, SetAspectRatio

Example
uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Ellipse(100, 100, 0, 360, 30, 50);
  Ellipse(100, 100, 0, 180, 50, 30);
  Readln;
  CloseGraph;
end.
EnvCount function

**Purpose**
Returns the number of strings contained in the DOS environment.

**Declaration**
```pascal
function EnvCount: Integer;
```

**Remarks**
`EnvCount` returns the number of strings contained in the DOS environment. Each environment string is of the form `VAR=VALUE`. The strings can be examined with the `EnvStr` function.

For more information about the DOS environment, see your DOS manuals.

**See also**
`EnvStr, GetEnv`

**Example**
```pascal
uses Dos;
var I: Integer;
begin
  for I := 1 to EnvCount do
    Writeln(EnvStr(I));
end.
```

EnvStr function

**Purpose**
Returns a specified environment string.

**Declaration**
```pascal
function EnvStr(Index: Integer): String;
```

**Remarks**
`EnvStr` returns a specified string from the DOS environment. The string `EnvStr` returns is of the form `VAR=VALUE`. The index of the first string is one. If `Index` is less than one or greater than `EnvCount`, `EnvStr` returns an empty string.

For more information about the DOS environment, see your DOS manuals.

**See also**
`EnvCount, GetEnv`

Eof function (text files)

**Purpose**
Returns the end-of-file status of a text file.

**Declaration**
```pascal
function Eof [ (var F: Text) ]: Boolean;
```

**Remarks**
`F`, if specified, is a text file variable. If `F` is omitted, the standard file variable `Input` is assumed. `Eof(F)` returns `True` if the current file position is
Eof function (text files)

beyond the last character of the file or if the file contains no components; otherwise, Eof(F) returns False.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
\textit{Eoln, SeekEof}

**Example**

```pascal
var
  F: Text;
  Ch: Char;
begin
  { Get file to read from command line }
  Assign(F, ParamStr(1));
  Reset(F);
  while not Eof(F) do
    begin
      Read(F, Ch);
      Write(Ch);
    end;
  { dump text file }
end.
```

### Eof function (typed, untyped files)

**System**

**Purpose**

Returns the end-of-file status of a typed or untyped file.

**Declaration**

\texttt{function Eof(var F): Boolean;}

**Remarks**

\(F\) is a file variable. \textit{Eof(F)} returns \texttt{True} if the current file position is beyond the last component of the file or if the file contains no components; otherwise, \textit{Eof(F)} returns \texttt{False}.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

### Eoln function

**System**

**Purpose**

Returns the end-of-line status of a text file.

**Declaration**

\texttt{function Eoln [(var F: Text)]: Boolean;}

**Remarks**

\(F\), if specified, is a text file variable. If \(F\) is omitted, the standard file variable \textit{Input} is assumed. \textit{Eoln(F)} returns \texttt{True} if the current file position is at an end-of-line marker or if \textit{Eof(F)} is \texttt{True}; otherwise, \textit{Eoln(F)} returns \texttt{False}.
Eoln function

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also Eof, SeekEoln

Erase procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Erases an external file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure Erase(var F);</td>
</tr>
</tbody>
</table>
| Remarks | $F$ is a file variable of any file type. The external file associated with $F$ is erased.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

| Restrictions | Never use Erase on an open file. |
| See also | Rename |
| Example | var F: file; Ch: Char; begin { Get file to delete from command line } Assign(F, ParamStr(1)); {$I-} Reset(F); {$I+} if IOResult <> 0 then Writeln('Cannot find ', ParamStr(1)) else begin Close(F); Write('Erase ', ParamStr(1), '?'); Readln(Ch); if UpCase(Ch) = 'Y' then Erase(F); end; end. |
### ErrorAddr variable

**Purpose**
Contains the address of the statement causing a run-time error.

**Declaration**
```pascal
var ErrorAddr: Pointer;
```

**Remarks**
If a program terminates normally or stops due to a call to `Halt`, `ErrorAddr` is `nil`. If a program ends because of a run-time error, `ErrorAddr` contains the address of the statement in error. For additional information, see “Exit procedures” in Chapter 20 in the Language Guide.

**See also**
`ExitCode`, `ExitProc`

### Exclude procedure

**Purpose**
Excludes an element from a set.

**Declaration**
```pascal
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 from the set given by S. The construct

```
Exclude(S, I)
```

corresponds to

```
S := S - [I]
```

but the `Exclude` procedure generates more efficient code.

**See also**
`Include`

### Exec procedure

**Purpose**
Executes a specified program with a specified command line.

**Declaration**
```pascal
procedure Exec(Path, CmdLine: String);
```

**Remarks**
The program name is given by the `Path` parameter, and the command line is given by `CmdLine`. To execute a DOS internal command, run COMMAND.COM; for instance,

```
Exec(‘\COMMAND.COM’, ‘/C DIR *.PAS’);
```
Exec procedure

The /C in front of the command is a requirement of COMMAND.COM (but not of other applications). Errors are reported in DosError. The exit code of any child process is reported by the DosExitCode function.

It is recommended that SwapVectors be called just before and just after the call to Exec. SwapVectors swaps the contents of the SaveIntXX pointers in the System unit with the current contents of the interrupt vectors. This ensures that the Exec'd process does not use any interrupt handlers installed by the current process, and vice versa.

Exec does not change the memory allocation state before executing the program. Therefore, when compiling a program that uses Exec, be sure to reduce the maximum heap size using a $M compiler directive; otherwise, there won't be enough memory (DosError = 8).

See also DosError, DosExitCode, SaveIntXX, SwapVectors

Example

```pascal
{$M $4000,0,0}                         { 16K stack, no heap required or reserved }
uses Dos;
var ProgramName, CmdLine: String;
begn
  Write('Program to Exec (include full path): ');
  Readln(ProgramName);
  Write('Command line to pass to ', ProgramName, ': ');
  Readln(CmdLine);
  WriteIn('About to Exec...');
  SwapVectors;
  Exec(ProgramName, CmdLine);
  SwapVectors;
  WriteIn('...back from Exec');
  if DosError <> 0 then                { Error? }
    WriteIn('Dos error #', DosError)
  else
    WriteIn('Exec successful. Child process exit code = ', DosExitCode);
end.
```

Exit procedure

Purpose

Exits immediately from the current block.

Declaration

```pascal
procedure Exit;
```

Remarks

Executed in a subroutine (procedure or function), Exit causes the subroutine to return. Executed in the statement part of a program, Exit causes the program to terminate. A call to Exit is analogous to a goto statement addressing a label just before the end of a block.
Exit procedure

See also

*Halt*

Example

```pascal
procedure WasteTime;
begin
  repeat
    if KeyPressed then Exit;
    Write('Xx');
  until False;
end;
begin
  WasteTime;
end.
```

ExitCode variable

**System**

**Purpose**
Contains the application's exit code.

**Declaration**
```
var ExitCode: Integer;
```

**Remarks**
An exit procedure can learn the cause of termination by examining `ExitCode`. If a program terminates normally, `ExitCode` is zero. If a program stops through a call to `Halt`, `ExitCode` contains the value passed to `Halt`. If a program ends due to a run-time error, `ExitCode` contains the error code.

**See also**

ExitProc variable

**System**

**Purpose**
Implements an application's exit procedure list.

**Declaration**
```
var ExitProc: Pointer;
```

**Remarks**
`ExitProc` lets you install an exit procedure to be called as part of a program's termination, whether it is a normal termination, a termination through a call to `Halt`, or a termination due to a run-time error.

**See also**
Exp function

Purpose
Returns the exponential of the argument.

Declaration
function Exp(X: Real): Real;

Remarks
X is a real-type expression. The result is the exponential of X; that is, the value \( e \) raised to the power of \( X \), where \( e \) is the base of the natural logarithms.

See also Ln

fcXXXX flag constants

Purpose
Return flags used by the function FileSplit.

Remarks
The following fcXXXX constants are defined:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcExtension</td>
<td>$0001</td>
</tr>
<tr>
<td>fcFileName</td>
<td>$0002</td>
</tr>
<tr>
<td>fcDirectory</td>
<td>$0004</td>
</tr>
<tr>
<td>fcWildcards</td>
<td>$0008</td>
</tr>
</tbody>
</table>

See also FileSplit

FExpand function

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

Declaration
function FExpand(Path: PathStr): PathStr;

Remarks
Expands the file name in Path into a fully qualified file name. The resulting name is converted to uppercase and consists of a drive letter, a colon, a root relative directory path, and a file name. Embedded '.' and '..' directory references are removed.

The PathStr type is defined in the Dos unit as string[79].

Assuming that the current drive and directory is C:\SOURCE\PAS, the following FExpand calls would produce these values:

- FExpand('test.pas') = 'C:\SOURCE\PAS\TEST.PAS'
- FExpand('..\*\Tpf') = 'C:\SOURCE\*\Tpf'
- FExpand('c:\bin\turbo.exe') = 'C:\BIN\TURBO.EXE'
**FExpand function**

`FSplit` can separate the result of `FExpand` into a drive/directory string, a file-name string, and an extension string.

**See also** FileExpand, FindFirst, FindNext, FSplit, File-handling string types

---

### File attribute constants

**Dos, WinDos**

**Purpose**
Used to construct file attributes in connection with the `GetFAttr`, `SetFAttr`, `FindFirst`, and `FindNext` procedures.

**Remarks**
These are the file attribute constants defined in the `Dos` and `WinDos` units.

<table>
<thead>
<tr>
<th>Dos Constant</th>
<th>WinDos Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ReadOnly</code></td>
<td><code>faReadOnly</code></td>
<td>$01</td>
</tr>
<tr>
<td><code>Hidden</code></td>
<td><code>faHidden</code></td>
<td>$02</td>
</tr>
<tr>
<td><code>SysFile</code></td>
<td><code>faSysFile</code></td>
<td>$04</td>
</tr>
<tr>
<td><code>VolumeID</code></td>
<td><code>faVolumeID</code></td>
<td>$08</td>
</tr>
<tr>
<td><code>Directory</code></td>
<td><code>faDirectory</code></td>
<td>$10</td>
</tr>
<tr>
<td><code>Archive</code></td>
<td><code>faArchive</code></td>
<td>$20</td>
</tr>
<tr>
<td><code>AnyFile</code></td>
<td><code>faAnyFile</code></td>
<td>$3F</td>
</tr>
</tbody>
</table>

The constants are additive, that is, the statement

```
FindFirst('*.*', ReadOnly + Directory, S);  \{ Dos \}
```

or

```
FindFirst('*.*', faReadOnly + faDirectory, S);  \{ WinDos \}
```

will locate all normal files as well as read-only files and subdirectories in the current directory. The `AnyFile` (or `faAnyFile`) constant is simply the sum of all attributes.

**See also** FindFirst, FindNext, GetFAttr, SetFAttr

---

### File name length constants

**WinDos**

**Purpose**
Contain the maximum file name component string lengths used by the functions `FileSearch` and `FileExpand`.

**Remarks**
The following **file name length** constants are defined:
File name length constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsPathName</td>
<td>79</td>
</tr>
<tr>
<td>fsDirectory</td>
<td>67</td>
</tr>
<tr>
<td>fsFileName</td>
<td>8</td>
</tr>
<tr>
<td>fsExtension</td>
<td>4</td>
</tr>
</tbody>
</table>

See also FileSearch, FileSplit

FileExpand function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Expands a file name into a fully qualified file name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function FileExpand(Dest, Name: PChar): PChar;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Expands the file name in Name into a fully qualified file name. The resulting name is converted to uppercase and consists of a drive letter, a colon, a root relative directory path, and a file name. Embedded ‘.’ and ‘..’ directory references are removed, and all name and extension components are truncated to 8 and 3 characters respectively. The returned value is Dest. Dest and Name can refer to the same location.</td>
</tr>
</tbody>
</table>

Assuming that the current drive and directory is C:\SOURCE\PAS, the following FileExpand calls would produce these values:

- FileExpand(S, 'test.pas') = 'C:\SOURCE\PAS\TEST.PAS'
- FileExpand(S, '..\.TPW') = 'C:\SOURCE\.TPW'
- FileExpand(S, 'c:\bin\turbo.exe') = 'C:\BIN\TURBO.EXE'

The FileSplit function can be used to split the result of FileExpand into a drive/directory string, a file-name string, and an extension string.

See also FExpand, FindFirst, File name lengths, FindNext, FileSplit

File-handling string types

<table>
<thead>
<tr>
<th>Purpose</th>
<th>String types are used by various procedures and functions in the Dos unit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remarks</td>
<td>The following string types are defined:</td>
</tr>
<tr>
<td></td>
<td>ComStr = string[127];  { Command-line string }</td>
</tr>
<tr>
<td></td>
<td>PathStr = string[79];  { Full file path string }</td>
</tr>
<tr>
<td></td>
<td>DirStr = string[67];  { Drive and directory string }</td>
</tr>
<tr>
<td></td>
<td>NameStr = string[8];  { File-name string }</td>
</tr>
<tr>
<td></td>
<td>ExtStr = string[4];  { File-extension string }</td>
</tr>
</tbody>
</table>

See also FExpand, FSplit
**FileMode variable**

**Purpose**
Determines the access code to pass to DOS when typed and untyped files are opened using the *Reset* procedure.

**Declaration**
```
var FileMode: Byte;
```

**Remarks**
The range of valid FileMode values depends on the version of DOS in use. For all versions, however, the following modes are defined:

- 0  Read only
- 1  Write only
- 2  Read/Write

The default value, 2, allows both reading and writing. Assigning another value to FileMode causes all subsequent Resets to use that mode. New files using Rewrite are always opened in read/write mode (that is, FileMode = 2).

DOS version 3.x and higher defines additional modes, which are primarily concerned with file-sharing on networks. For more details, see your DOS programmer's reference manual.

**See also** Rewrite

---

**FilePos function**

**Purpose**
Returns the current file position of a file.

**Declaration**
```
function FilePos(var F): Longint;
```

**Remarks**

- F is a file variable. If the current file position is at the beginning of the file, FilePos(F) returns 0. If the current file position is at the end of the file—that is, if Eof(F) is True—FilePos(F) is equal to FileSize(F).

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

**Restrictions**
Cannot be used on a text file. File must be open.

**See also** FileSize, Seek
FileRec type

Purpose  Record definition used internally by Turbo Pascal and also declared in the Dos unit.

Declaration  type
FileRec = record
  Handle: Word;
  Mode: Word;
  RecSize: Word;
  Private: array[1..26] of Byte;
  UserData: array[1..16] of Byte;
  Name: array[0..79] of Char;
end;

Remarks  FileRec defines the internal data format of both typed and untyped files.

See also  TextRec

FileSearch function

Purpose  Searches for a file in a list of directories.

Declaration  function FileSearch(Dest, Name, List: PChar): PChar;

Remarks  Searches for the file given by Name in the list of directories given by List. The directories in List must be separated by semicolons, just like the directories specified in a PATH command in DOS. The search always starts with the current directory of the current drive. If the file is found, FileSearch stores a concatenation of the directory path and the file name in Dest. Otherwise, FileSearch stores an empty string in Dest. The returned value is Dest. Dest and Name must not refer to the same location.

The maximum length of the result is defined by the fsPathName constant, which is 79.

To search the PATH used by DOS to locate executable files, call GetEnvVar('PATH') and pass the result to FileSearch as the List parameter.

The result of FileSearch can be passed to FileExpand to convert it into a fully qualified file name; that is, an uppercase file name that includes both a drive letter and a root-relative directory path. In addition, you can use FileSplit to split the file name into a drive/directory string, a file-name string, and an extension string.
See also  *File name lengths, FileExpand, FileSplit, FSearch*

**Example**

```pascal
uses WinDos;
var
  S: array[0..fsPathName] of Char;
begins
  FileSearch(S, 'TURBO.EXE', GetEnvVar('PATH'));
  if S[0] = #0 then
    Writeln('TURBO.EXE not found')
  else
    Writeln('Found as ', FileExpand(S, S));
end.
```

**FileSize function**

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Returns the current size of a file.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function FileSize(var F): Longint;</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><em>F</em> is a file variable. <em>FileSize(F)</em> returns the number of components in <em>F</em>. If the file is empty, <em>FileSize(F)</em> returns 0. With {$I-}, <em>IOResult</em> returns 0 if the operation was successful; otherwise, it returns a nonzero error code.</td>
</tr>
<tr>
<td><strong>Restrictions</strong></td>
<td>Cannot be used on a text file. File must be open.</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td><em>FilePos</em></td>
</tr>
</tbody>
</table>
| **Example**       | **var** *F*: file of Byte;
  **begin**
  { Get file name from command line }
  Assign(*F*, ParamStr(1));
  Reset(*F*);
  Writeln('File size in bytes: ', *FileSize(F)*);
  Close(*F*);
  end. |

**FileSplit function**

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Splits a file name into its three components.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function FileSplit(Path, Dir, Name, Ext: PChar): Word;</code></td>
</tr>
</tbody>
</table>
| **Remarks**       | Splits the file name specified by *Path* into its three components. *Dir* is set to the drive and directory path with any leading and trailing backslashes, *Name* is set to the file name, and *Ext* is set to the extension with a
preceding period. If a component string parameter is nil, the corresponding part of the path is not stored. If the path does not contain a given component, the returned component string is empty. The maximum string lengths returned in Dir, Name, and Ext are defined by the fsDirectory, fsFileName, and fsExtension constants.

The returned value is a combination of the fcDirectory, fcFileName, and fcExtension bit masks, indicating which components were present in the path. If the name or extension contains any wildcard characters (* or ?), the fcWildcards flag is set in the returned value.

See page 42 for a list of fcXXXX flag constants and page 43 for a list of fsXXXX file name length constants.

See also FileExpand, FindFirst, FindNext, FSplit

Example

uses Strings, WinDos;
var
Path: array[0..fsPathName] of Char;
Dir: array[0..fsDirectory] of Char;
Name: array[0..fsFileName] of Char;
Ext: array[0..fsExtension] of Char;
begin
Write('Filename (WORK. PAS): ');
Readln(Path);
FileSplit(Path, Dir, Name, Ext);
if Name[0] = #0 then StrCopy(Name, 'WORK');
if Ext[0] = #0 then StrCopy(Ext, '.PAS');
StrECopy(StrECopy(StrECopy(Path, Dir), Name), Ext);
Writeln('Resulting name is ', Path);
end.

Fill pattern constants

Purpose

Constants that determine the pattern used to fill an area.

Remarks

Use SetFillPattern to define your own fill pattern, then call SetFillStyle(UserFill, SomeColor) and make your fill pattern the active style.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyFill</td>
<td>0</td>
<td>Fills area in background color</td>
</tr>
<tr>
<td>SolidFill</td>
<td>1</td>
<td>Fills area in solid fill color</td>
</tr>
<tr>
<td>LineFill</td>
<td>2</td>
<td>———— fill</td>
</tr>
<tr>
<td>LtSlashFill</td>
<td>3</td>
<td>/// fill</td>
</tr>
<tr>
<td>SlashFill</td>
<td>4</td>
<td>/// fill with thick lines</td>
</tr>
<tr>
<td>BkSlashFill</td>
<td>5</td>
<td>\ fill with thick lines</td>
</tr>
<tr>
<td>LtBkSlashFill</td>
<td>6</td>
<td>\ fill</td>
</tr>
</tbody>
</table>
Fill pattern constants

| HatchFill  | 7 | Light hatch fill |
| XHatchFill | 8 | Heavy cross hatch fill |
| InterleaveFill | 9 | Interleaving line fill |
| WideDotFill | 10 | Widely spaced dot fill |
| CloseDotFill | 11 | Closely spaced dot fill |
| UserFill | 12 | User-defined fill |

See also FillPatternType, GetFillSettings, SetFillStyle

FillChar procedure

| Purpose | Fills a specified number of contiguous bytes with a specified value. |
| Declaration | procedure FillChar(var X; Count: Word; Value); |
| Remarks | X is a variable reference of any type. Count is an expression of type Word. Value is any ordinal-type expression. FillChar writes Count contiguous bytes of memory into Value, starting at the first byte occupied by X. No range-checking is performed, so be careful to specify the correct number of bytes. Whenever possible, use the SizeOf function to specify the count parameter. When using FillChar on strings, remember to set the length byte afterward. |
| See also | Move |
| Example | var S: string[80]; begin { Set a string to all spaces } FillChar(S, SizeOf(S), ' '); S[0] := #80; { Set length byte } end. |

FillEllipse procedure

| Purpose | Draws a filled ellipse. |
| Declaration | procedure FillEllipse(X, Y: Integer; XRadius, YRadius: Word); |
| Remarks | Draws a filled ellipse using (X, Y) as a center point, and XRadius and YRadius as the horizontal and vertical axes. The ellipse is filled with the current fill color and fill style, and is bordered with the current color. |
| Restrictions | Must be in graphics mode. |

**FillEllipse procedure**

See also  
*Arc, Circle, Ellipse, GetArcCoords, GetAspectRatio, PieSlice, Sector, SetAspectRatio*

Example  
```pascal
uses Graph;
const R = 30;
var
  Driver, Mode: Integer;
  Xasp, Yasp: Word;
begin
  Driver := Detect;
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  { Draw ellipse }
  FillEllipse(GetMaxX div 2, GetMaxY div 2, 50, 50);
  GetAspectRatio(Xasp, Yasp);
  { Circular ellipse }
  FillEllipse(R, R, R, R * Longint(Xasp) div Yasp);
  Readln;
  CloseGraph;
end.
```

**FillPatternType type**

Purpose  
Record that defines a user-defined fill pattern.

Declaration  
```pascal
FillPatternType = array[1..8] of Byte;
```

See also  
*Fill pattern constants, GetFillPattern, SetFillPattern*

**FillPoly procedure**

Purpose  
Draws and fills a polygon, using the scan converter.

Declaration  
```pascal
procedure FillPoly(NumPoints: Word; var PolyPoints);
```

Remarks  
*PolyPoints* is an untyped parameter that contains the coordinates of each intersection in the polygon. *NumPoints* specifies the number of coordinates in *PolyPoints*. A coordinate consists of two words, an X and a Y value.

*FillPoly* calculates all the horizontal intersections, and then fills the polygon using the current fill style and color defined by *SetFillStyle* or *SetFillPattern*. The outline of the polygon is drawn in the current line style and color as set by *SetLineStyle*. 
If an error occurs while filling the polygon, `GraphResult` returns a value of \(-6\) (`grNoScanMem`).

**Restrictions**
Must be in graphics mode.

**See also**
`DrawPoly`, `GetFillSettings`, `GetLineSettings`, `GraphResult`, `SetFillPattern`, `SetFillStyle`, `SetLineStyle`

**Example**
```pascal
uses Graph;
const
    Triangle: array[1..3] of PointType = ((X: 50; Y: 100),
        (X: 100; Y: 100), (X: 150; Y: 150));
var Gd, Gm: Integer;
begin
    Gd := Detect;
    InitGraph(Gd, Gm, ");
    if GraphResult <> grOk then
        Halt(1);
    FillPoly(SizeOf(Triangle) div SizeOf(PointType), Triangle);
    Readln;
    CloseGraph;
end.
```

**FillSettingsType**

**Purpose**
The record that defines the pattern and color used to fill an area.

**Declaration**
```pascal
type
    FillSettingsType = record
        Pattern: Word;
        Color: Word;
    end;
```

**See also**
`GetFillSettings`

**FindFirst procedure**

**Purpose**
Searches the specified (or current) directory for the first entry matching
the specified file name and set of attributes.

**Declaration**
```pascal
procedure FindFirst(Path: String; Attr: Word; var S: SearchRec); \{ Dos \}
procedure FindFirst(Path: PChar; Attr: Word; var S: TSearchRec); \{ WinDos \}
```

**Remarks**
`Path` is the directory mask (for example, `*.*`). The `Attr` parameter specifies
the special files to include (in addition to all normal files). See page 43 for
a list of `Dos` and `WinDos` file attribute constants.
FindFirst procedure

The result of the directory search is returned in the specified search record. See page 147 for a declaration of SearchRec and page 199 for a declaration of TSearchRec.

Errors are reported in DosError; possible error codes are 3 (Path not found) and 18 (No more files).

See also DosError, FExpand, File attribute constants, FileExpand, FindNext, SearchRec, TSearchRec

Example

uses Dos;
var DirInfo: SearchRec;
begin
  FindFirst('*.PAS', Archive, DirInfo);
  while DosError = 0 do
    begin
      Writeln(DirInfo.Name);
      FindNext(DirInfo);
    end;
end.

FindNext procedure

Returns the next entry that matches the name and attributes specified in a previous call to FindFirst.

Dos, WinDos

Purpose

Declaration

procedure FindNext(var S: SearchRec); { Dos }

procedure FindNext(var S: TSearchRec); { WinDos }

Remarks

The search record must be the same search record passed to FindFirst. Errors are reported in DosError; the only possible error code is 18 (No more files).

See also DosError, File attribute, FExpand, FileExpand, FindFirst, SearchRec, TSearchRec

Example

See the example for FindFirst.
Flag constants

**Flag constants**

**Dos, WinDos**

**Purpose**

Used to test individual flag bits in the Flags register after a call to Intr or MsDos.

**Remarks**

<table>
<thead>
<tr>
<th>Constants</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCarry</td>
<td>$0001</td>
</tr>
<tr>
<td>FParity</td>
<td>$0004</td>
</tr>
<tr>
<td>FAuxiliary</td>
<td>$0010</td>
</tr>
<tr>
<td>FZero</td>
<td>$0040</td>
</tr>
<tr>
<td>FSign</td>
<td>$0080</td>
</tr>
<tr>
<td>FOverflow</td>
<td>$0800</td>
</tr>
</tbody>
</table>

For instance, if \( R \) is a register record, the tests

\[
R.\text{Flags} \text{ and } F\text{Carry} <\> 0 \\
R.\text{Flags} \text{ and } F\text{Zero} = 0
\]

are True respectively if the Carry flag is set and if the Zero flag is clear.

**See also** Intr, MsDos

---

**FloodFill procedure**

Fills a bounded region with the current fill pattern.

**Declaration**

```pascal
procedure FloodFill(X: Integer; 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 pattern, as set by SetFillStyle or SetFillPattern, 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.

If an error occurs while flooding a region, GraphResult returns a value of grNoFloodMem.

Note that FloodFill stops after two blank lines have been output. This can occur with a sparse fill pattern and a small polygon. In the following program, the rectangle is not completely filled:

```pascal
program StopFill;
uses Graph;
var Driver, Mode: Integer;
```

---

Chapter 1, Library reference 53
FloodFill procedure

begin
  Driver := Detect;
  InitGraph(Driver, Mode, 'c:\bgi');
  if GraphResult <> grOk then
    Halt(1);
    SetFillStyle(LtSlashFill, GetMaxColor);
    Rectangle(0, 0, 8, 20);
    FloodFill(1, 1, GetMaxColor);
    Readln;
    CloseGraph;
end.

In this case, using a denser fill pattern like SlashFill will completely fill the figure.

Restrictions

Use FillPoly instead of FloodFill whenever possible so you can maintain code compatibility with future versions. Must be in graphics mode. This procedure is not available when using the IBM 8514 graphics driver (IBM8514.BGI).

See also

FillPoly, GraphResult, SetFillPattern, SetFillStyle

Example

uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' '); 
  if GraphResult <> grOk then
    Halt(1);
    SetColor(GetMaxColor);
    Circle(50, 50, 20);
    FloodFill(50, 50, GetMaxColor);
    Readln;
    CloseGraph;
end.

Flush procedure

Purpose
Flushes the buffer of a text file open for output.

Declaration
procedure Flush(var F: Text);

Remarks
F is a text file variable.

When a text file has been opened for output using Rewrite or Append, a call to Flush will empty the file's buffer. This guarantees that all characters written to the file at that time have actually been written to the external file. Flush has no effect on files opened for input.
With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

**fmXXXX constants**  
**Dos, WinDos**

**Purpose**  
Defines the allowable values for the *Mode* field of a `TextRec` and `TFileRec` text file record.

**Remarks**  
The *Mode* fields of Turbo Pascal's file variables contain one of the values specified here:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmClosed</code></td>
<td>$D7B0</td>
</tr>
<tr>
<td><code>fmInput</code></td>
<td>$D7B1</td>
</tr>
<tr>
<td><code>fmOutput</code></td>
<td>$D7B2</td>
</tr>
<tr>
<td><code>fmInOut</code></td>
<td>$D7B3</td>
</tr>
</tbody>
</table>

**See also**  
`TextRec`, `TFileRec`

**Font control constants**  
**Graph**

**Purpose**  
Constants that identify fonts.

**Remarks**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DefaultFont</code></td>
<td>0 (8x8 bit mapped font)</td>
</tr>
<tr>
<td><code>TriplexFont</code></td>
<td>1 (&quot;stroked&quot; fonts)</td>
</tr>
<tr>
<td><code>SmallFont</code></td>
<td>2</td>
</tr>
<tr>
<td><code>SansSerifFont</code></td>
<td>3</td>
</tr>
<tr>
<td><code>GothicFont</code></td>
<td>4</td>
</tr>
<tr>
<td><code>HorizDir</code></td>
<td>0 (left to right)</td>
</tr>
<tr>
<td><code>VertDir</code></td>
<td>1 (bottom to top)</td>
</tr>
<tr>
<td><code>UserCharSize</code></td>
<td>0 (user-defined Char size)</td>
</tr>
</tbody>
</table>

**See also**  
`GetTextSettings`, `SetTextStyle`, `TextSettingsType`

**Frac function**  
**System**

**Purpose**  
Returns the fractional part of the argument.

**Declaration**  
```plaintext```
```python
function Frac(X: Real): Real;
```
Frac function

Remarks  
X is a real-type expression. The result is the fractional part of X; that is,  
\( \text{Frac}(X) = X - \text{Int}(X) \).

See also  
\( \text{Int} \)

Example  
\begin{verbatim}
var R: Real;
begin
  R := Frac(123.456);  \{ 0.456 \}
  R := Frac(-123.456); \{ -0.456 \}
end.
\end{verbatim}

FreeList variable

Purpose  
Points to the first free block in the heap.

Declaration  
\begin{verbatim}
var FreeList: Pointer;
\end{verbatim}

Remarks  
The \texttt{FreeList} variable points to the first free block in the heap. This block 
contains a pointer to the next free block, which contains a pointer to the 
next free block, and so forth. The last free block contains a pointer to the 
top of the heap. If there are no free blocks on the free list, \texttt{FreeList} will be 
equal to \texttt{HeapPtr}. See Chapter 12, "Standard procedures and functions," 
in the Language Guide for more information.

See also  
\texttt{Dispose, FreeMem, HeapPtr}

FreeMem procedure

Purpose  
Disposes of a dynamic variable of a given size.

Declaration  
\begin{verbatim}
procedure FreeMem(var P: Pointer; Size: Word);
\end{verbatim}

Remarks  
\( P \) is a variable of any pointer type previously assigned by the \texttt{GetMem} 
procedure or assigned a meaningful value by an assignment statement. 
\( Size \) is an expression specifying the size in bytes of the dynamic variable to 
dispose of; it must be \textit{exactly} the number of bytes previously allocated to 
that variable by \texttt{GetMem}. \texttt{FreeMem} destroys the variable referenced by \( P \) 
and returns its memory region to the heap. If \( P \) does not point to a 
memory region in the heap, a run-time error occurs. After a call to 
\texttt{FreeMem}, the value of \( P \) becomes undefined, and an error occurs if you 
subsequently reference \( P \).

See also  
\texttt{Dispose, FreeMem, HeapError, New}
FSearch function

Purpose
Searches for a file in a list of directories.

Declaration
function FSearch(Path: PathStri; DirList: String): PathStr;

Remarks
Searches for the file given by Path in the list of directories given by DirList. The directories in DirList must be separated by semicolons, just like the directories specified in a PATH command in DOS. The search always starts with the current directory of the current drive. The returned value is a concatenation of one of the directory paths and the file name, or an empty string if the file could not be located.

To search the PATH used by DOS to locate executable files, call GetEnv('PATH') and pass the result to FSearch as the DirList parameter.

The result of FSearch can be passed to FExpand to convert it into a fully qualified file name, that is, an uppercase file name that includes both a drive letter and a root-relative directory path. In addition, you can use FSplit to split the file name into a drive/directory string, a file-name string, and an extension string.

See also
FExpand, FSplit, GetEnv

Example
uses Dos;
var S: PathStri;
begin
S := FSearch('TURBO.EXE', GetEnv('PATH'));
if S = '' then
  Writeln('TURBO.EXE not found')
else
  Writeln('Found as ', FExpand(S));
end.

FSplit procedure

Purpose
Splits a file name into its three components.

Declaration
procedure FSplit(Path: PathStri; var Dir: DirStri; var Name: NameStri;
  var Ext: ExtStri);

Remarks
Splits the file name specified by Path into its three components. Dir is set to the drive and directory path with any leading and trailing backslashes, Name is set to the file name, and Ext is set to the extension with a
**FSplit procedure**

preceding dot. Each of the component strings might possibly be empty, if Path contains no such component.

FSplit never adds or removes characters when it splits the file name, and the concatenation of the resulting Dir, Name, and Ext will always equal the specified Path.

See page 44 for a list of File-handling string types.

**See also** FExpand, File-handling string types

**Example** uses Dos;

```pascal
var
    P: PathStr;
    D: DirStr;
    N: NameStr;
    E: ExtStr;
begin
    Write('Filename (WORK.PAS): ');
    Readln(P);
    FSplit(P, D, N, E);
    if N = '' then N := 'WORK';
    if E = '' then E := '.PAS';
    P := D + N + E;
    Writeln('Resulting name is ', P);
end.
```

**GetArcCoords procedure**

**Graph**

**Purpose** Lets the user inquire about the coordinates of the last Arc command.

**Declaration** procedure GetArcCoords(var ArcCoords: ArcCoordsType);

**Remarks** GetArcCoords returns a variable of type ArcCoordsType. GetArcCoords returns a variable containing the center point (X, Y), the starting position (Xstart, Ystart), and the ending position (Xend, Yend) of the last Arc or Ellipse command. These values are useful if you need to connect a line to the end of an ellipse.

**Restrictions** Must be in graphics mode.

**See also** Arc, Circle, Ellipse, FillEllipse, PieSlice, PieSliceXY, Sector

**Example** uses Graph;

```pascal
var
    Gd, Gm: Integer;
```

Programmer's Reference
GetArcCoords procedure

ArcCoords: ArcCoordsType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Arc(100, 100, 0, 270, 30);
  GetArcCoords(ArcCoords);
  with ArcCoords do
    Line(Xstart, Ystart, Xend, Yend);
  Readln;
  CloseGraph;
end.

GetArgCount function

Purpose Returns the number of parameters passed to the program on the command line.

Declaration function GetArgCount: Integer

See also GetArgStr, ParamCount, ParamStr

GetArgStr function

Purpose Returns the command-line parameter specified by Index.

Declaration function GetArgStr(Dest: PChar; Index: Integer; MaxLen: Word): PChar;

Remarks If Index is less than zero or greater than GetArgCount, GetArgStr returns an empty string. If Index is zero, GetArgStr returns the file name of the current module. Dest is the returned value. The maximum length of the returned string is specified by the MaxLen parameter.

See also GetArgCount, ParamCount, ParamStr

GetAspectRatio procedure

Purpose Returns the effective resolution of the graphics screen from which the aspect ratio (Xasp:Yasp) can be computed.

Declaration procedure GetAspectRatio(var Xasp, Yasp: Word);
GetAspectRatio procedure

Remarks Each driver and graphics mode has an aspect ratio associated with it (maximum Y resolution divided by maximum X resolution). This ratio can be computed by making a call to GetAspectRatio and then dividing the Xasp parameter by the Yasp parameter. This ratio is used to make circles, arcs, and pie slices round.

Restrictions Must be in graphics mode.

See also Arc, Circle, Ellipse, GetMaxX, GetMaxY, PieSlice, SetAspectRatio

Example uses Graph;
var Gd, Gm: Integer;
Xasp, Yasp: Word;
XSideLength, YSideLength: Integer;
begin Gd := Detect;
InitGraph(Gd, Gm, '');
if GraphResult <> grOk then
  Halt(1);
GetAspectRatio(Xasp, Yasp);
XSideLength := 20;
{ Adjust Y length for aspect ratio }
YSideLength := Round((Xasp / Yasp) * XSideLength);
{ Draw a 'square' rectangle on the screen }
Rectangle(0, 0, XSideLength, YSideLength);
Readln;
CloseGraph;
end.

GetBkColor function

Purpose Returns the index into the palette of the current background color.

Declaration function GetBkColor: Word;

Remarks Background colors range from 0 to 15, depending on the current graphics driver and current graphics mode.

GetBkColor returns 0 if the 0th palette entry is changed by a call to SetPalette or SetAllPalette.

Restrictions Must be in graphics mode.

See also GetColor, GetPalette, InitGraph, SetAllPalette, SetBkColor, SetColor, SetPalette
Example uses Crt, Graph;
var
  Gd, Gm: Integer;
  Color: Word;
  Pal: PaletteType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ");
  if GraphResult <> grOk then
    Halt(1);
  Randomize;
  GetPalette(Pal);
  if Pal.Size <> 1 then
    begin
      repeat
        Color := Succ(GetBkColor);
        if Color > Pal.Size-1 then
          Color := 0;
        SetBkColor(Color);
        LineTo(Random(GetMaxX), Random(GetMaxY));
      until KeyPressed;
    end
  else
    Line(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.

GetCBreak procedure

Purpose Gets Ctrl+Break checking.

Declaration procedure GetCBreak(var Break: Boolean);

Remarks GetCBreak sets the value of Break depending on the state of Ctrl+Break checking in DOS. When off (False), DOS only checks for Ctrl+Break during I/O to console, printer, or communication devices. When on (True), checks are made at every system call.

See also SetCBreak
GetColor function

**Purpose**
Returns the color value passed to the previous successful call to `SetColor`.

**Declaration**
```pascal
function GetColor: Word;
```

**Remarks**
Drawing colors range from 0 to 15, depending on the current graphics driver and current graphics mode.

**Restrictions**
Must be in graphics mode.

**See also**
`GetBkColor`, `GetPalette`, `InitGraph`, `SetAllPalette`, `SetColor`, `SetPalette`

**Example**
```pascal
uses Graph;
var
  Gd, Gm: Integer;
  Color: Word;
  Pal: PaletteType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Randomize;
  GetPalette(Pal);
  repeat
    Color := Succ(GetColor);
    if Color > Pal.Size - 1 then
      Color := 0;
    SetColor(Color);
    LineTo(Random(GetMaxX), Random(GetMaxY));
  until KeYPressed;
  CloseGraph;
end.
```

GetCurDir function

**Purpose**
Returns the current directory of a specified drive.

**Declaration**
```pascal
function GetCurDir(Dir: PChar; Drive: Byte): PChar
```

**Remarks**
The string returned in `Dir` always starts with a drive letter, a colon, and a backslash. `Drive = 0` indicates the current drive, 1 indicates A, 2 indicates B, and so on. The returned value is `Dir`. Errors are reported in `DosError`.

If the drive specified by `Drive` is invalid, `Dir` returns 'X:\', as if it were the root directory of the invalid drive.
GetCurDir function

The maximum length of the resulting string is defined by the fsDirectory constant.

See also  SetCurDir, CreateDir, RemoveDir. GetDir returns the current directory of a specified drive as a Pascal-style string.

GetDate procedure

Purpose  Returns the current date set in the operating system.

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

Remarks  Ranges of the values returned are Year 1980..2099, Month 1..12, Day 1..31, and DayOfWeek 0..6 (where 0 corresponds to Sunday).

See also  GetTime, SetDate, SetTime

GetDefaultPalette function

Purpose  Returns the palette definition record.

Declaration  function GetDefaultPalette(var Palette: PaletteType): PaletteType;

Remarks  GetDefaultPalette returns a PaletteType record, which contains the palette as the driver initialized it during InitGraph.

Restrictions  Must be in graphics mode.

See also  InitGraph, GetPalette, SetAllPalette, SetPalette

Example  uses Crt, Graph;
  var
    Driver, Mode, I: Integer;
    MyPal, OldPal: PaletteType;
  begin
    DirectVideo := False;
    Randomize;
    Driver := Detect;
    InitGraph(Driver, Mode, '');
    if GraphResult < 0 then
      Halt(1);
    GetDefaultPalette(OldPal);
    MyPal := OldPal;  { Duplicate and modify }
    MyPal := MyPal;   { Preserve old one }
    MyPal := MyPal;   { Display something }

GetDefaultPalette function

begin
  for I := 0 to MyPal.Size - 1 do
  begin
    SetColor(I);
    OutTextXY(10, I * 10, '...Press any key...');
  end;
  repeat
    with MyPal do
    begin
      Colors[Random(Size)] := Random(Size + 1);
      SetAllPalette(MyPal);
    until KeyPressed;
    SetAllPalette(OldPal);  \{ Restore original palette \}
    ClearDevice;
    OutTextXY(10, 10, 'Press <Return>...');
    Readln;
    CloseGraph;
  end;

GetDir procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the current directory of a specified drive.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure GetDir(D: Byte; var S: String);</td>
</tr>
<tr>
<td>Remarks</td>
<td>( D ) is an integer-type expression, and ( S ) is a string-type variable. The current directory of the drive specified by ( D ) is returned in ( S ). ( D = 0 ) indicates the current drive, ( 1 ) indicates drive ( A ), ( 2 ) indicates drive ( B ), and so on. ( GetDir ) performs no error-checking. If the drive specified by ( D ) is invalid, ( S ) returns 'X:', as if it were the root directory of the invalid drive.</td>
</tr>
<tr>
<td>See also</td>
<td>( ChDir, MkDir, RmDir ). ( GetCurDir ) performs the same function as ( GetDir ), but it takes a null-terminated string as an argument instead of a Pascal-style string.</td>
</tr>
</tbody>
</table>

GetDriverName function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns a string containing the name of the current driver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function GetDriverName: String;</td>
</tr>
<tr>
<td>Remarks</td>
<td>After a call to ( InitGraph ), returns the name of the active driver.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Must be in graphics mode.</td>
</tr>
<tr>
<td>See also</td>
<td>( GetModeName, InitGraph )</td>
</tr>
</tbody>
</table>
GetDriverName function

Example

uses Graph;
var Driver, Mode: Integer;
begin
  Driver := Detect; { Put in graphics mode }
  InitGraph(Driver, Mode, ",
  if GraphResult < 0 then
    Halt(1);
  OutText('Using driver ' + GetDriverName);
  Readln;
  CloseGraph;
end.

GetEnv function

DOS

Purpose
Returns the value of a specified environment variable.

Declaration
function GetEnv(EnvVar: String): String;

Remarks
GetEnv returns the value of a specified variable. The variable name can be either uppercase or lowercase, but it must not include the equal sign (=) character. If the specified environment variable does not exist, GetEnv returns an empty string.

For more information about the DOS environment, see your DOS manuals.

See also
EnvCount, EnvStr

Example

{$M 8192,0,0}
uses Dos;
var Command: string[79];
begin
  Write('Enter DOS command: ');
  Readln(Command);
  if Command <> ", then
    Command := '/C ' + Command;
  SwapVectors;
  Exec(GetEnv('COMSPEC'), Command);
  SwapVectors;
  if DosError <> 0 then
    Writeln('Could not execute COMMAND.COM');
end.
GetEnvVar function

**Purpose**
Returns a pointer to the value of a specified environment variable.

**Declaration**
```
function GetEnvVar(VarName: PChar): PChar
```

**Remarks**
`GetEnvVar` returns a pointer to the value of a specified variable; for example, a pointer to the first character after the equals sign (=) in the environment entry given by `VarName`. The variable name can be in either uppercase or lowercase, but it must not include the equal sign (=) character. If the specified environment variable does not exist, `GetEnvVar` returns `nil`.

**Example**
```
uses WinDos;
begin
  Writeln('The current PATH is ', GetEnvVar('PATH'));
end.
```

GetFAttr procedure

**Purpose**
Returns the attributes of a file.

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

**Remarks**
`F` must be a file variable (typed, untyped, or text file) that has been assigned but not opened. The attributes are examined by `anding` them with the file attribute masks defined as constants in the `Dos` unit. See page 43 for a list of file attribute constants for `Dos` and `WinDos` units.

Errors are reported in `DosError`; possible error codes are

- 3 (Invalid path)
- 5 (File access denied)

**Restrictions**
`F` cannot be open.

**See also**
`DosError`, File attribute constants, `GetFTime`, `SetFAttr`, `SetFTime`

**Example**
```
uses Dos; { or WinDos }
var
  F: file;
  Attr: Word;
begin
  { Get file name from command line }
  Assign(F, ParamStr(1));
  GetFAttr(F, Attr);
  Writeln(ParamStr(1));
```
GetFAttr procedure

```pascal
if DosError <> 0 then
  Writeln('DOS error code = ', DosError)
else
begin
  Write('Attribute = ', Attr);
  { Determine attribute type using File attribute constants in Dos or WinDos unit }
  if Attr and ReadOnly <> 0 then
    Writeln('Read only file');
  if Attr and Hidden <> 0 then
    Writeln('Hidden file');
  if Attr and SysFile <> 0 then
    Writeln('System file');
  if Attr and VolumeID <> 0 then
    Writeln('Volume ID');
  if Attr and Directory <> 0 then
    Writeln('Directory name');
  if Attr and Archive <> 0 then
    Writeln('Archive (normal file)');
end; { else }
end.
```

GetFillPattern procedure

**Purpose**
Returns the last fill pattern set by a previous call to `SetFillPattern`.

**Declaration**
```pascal
procedure GetFillPattern(var FillPattern: FillPatternType);
```

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

**Restrictions**
Must be in graphics mode.

**See also**
`GetFillSettings`, `SetFillPattern`, `SetFillStyle`

GetFillSettings procedure

**Purpose**
Returns the last fill pattern and color set by a previous call to `SetFillPattern` or `SetFillStyle`.

**Declaration**
```pascal
procedure GetFillSettings(var Filllnfo: FillSettingsType);
```

**Remarks**
The `Pattern` field reports the current fill pattern selected. The `Color` field reports the current fill color selected. Both the fill pattern and color can be changed by calling the `SetFillStyle` or `SetFillPattern` procedure. If `Pattern` is
GetFillSettings procedure

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

Restrictions
Must be in graphics mode.

See also
FillPoly, GetFillPattern, SetFillPattern, SetFillStyle

Example

uses Graph;
var
Gd, Gm: Integer;
FillInfo: FillSettingsType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  GetFillSettings(FillInfo);
  { Save fill style and color }
  Bar(0, 0, 50, 50);
  SetFillStyle(XHatchFill, GetMaxColor);
  { New style }
  Bar(50, 0, 100, 50);
  with FillInfo do
    { Restore old fill style }
    SetFillStyle(Pattern, Color);
  Bar(100, 0, 150, 50);
  Readln;
  CloseGraph;
end.

GetFTime procedure

Purpose
Returns the date and time a file was last written.

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

Remarks
F must be a file variable (typed, untyped, or text file) that has been
assigned and opened. The time returned in the Time parameter can be
unpacked through a call to UnpackTime. Errors are reported in DosError;
the only possible error code is 6 (Invalid file handle).

Restrictions
F must be open.

See also
DosError, PackTime, SetFTime, UnpackTime
GetGraphMode function

**Purpose**
Returns the current graphics mode.

**Declaration**
```pascal
function GetGraphMode: Integer;
```

**Remarks**
GetGraphMode returns the current graphics mode set by `InitGraph` or `SetGraphMode`. The Mode value is an integer from 0 to 5, depending on the current driver.

The following mode constants are defined:

<table>
<thead>
<tr>
<th>Graphics driver</th>
<th>Constant name</th>
<th>Value</th>
<th>Column x row</th>
<th>Palette</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGA</td>
<td>CGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAHl</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>MCGA</td>
<td>MCGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAMed</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAHi</td>
<td>5</td>
<td>640x480</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA</td>
<td>EGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>EGAHi</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td>EGA64</td>
<td>EGA64Lo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EGA64Hi</td>
<td>1</td>
<td>640x350</td>
<td>4 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA-MONO</td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>1*</td>
</tr>
<tr>
<td></td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>2**</td>
</tr>
<tr>
<td>HERC</td>
<td>HercMonoHi</td>
<td>0</td>
<td>720x348</td>
<td>2 color</td>
<td>2</td>
</tr>
<tr>
<td>ATT400</td>
<td>ATT400C0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400Med</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400Hi</td>
<td>5</td>
<td>640x400</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>VGA</td>
<td>VGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAMed</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAHl</td>
<td>2</td>
<td>640x480</td>
<td>16 color</td>
<td>1</td>
</tr>
</tbody>
</table>
GetGraphMode function

<table>
<thead>
<tr>
<th></th>
<th>PC3270</th>
<th>PC3270Hi</th>
<th>0</th>
<th>720x350</th>
<th>2 color</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM8514</td>
<td>IBM8514Lo</td>
<td>0</td>
<td>640x480</td>
<td>256 color</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IBM8514</td>
<td>IBM8514Hi</td>
<td>0</td>
<td>1024x768</td>
<td>256 color</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

* 64K on EGAMono card
** 256K on EGAMono card

Restrictions
Must be in graphics mode.

See also
ClearDevice, DetectGraph, InitGraph, RestoreCrtMode, SetGraphMode

Example

```pascal
uses Graph;

var
  Gd, Gm: Integer;
  Mode: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  OutText('<ENTER> to leave graphics:');
  Readln;
  RestoreCrtMode;
  Writeln('Now in text mode');
  Write('<ENTER> to enter graphics mode:');
  Readln;
  SetGraphMode(GetGraphMode);
  OutTextXY(0, 0, 'Back in graphics mode');
  OutTextXY(0, TextHeight('H'), '<ENTER> to quit:');
  Readln;
  CloseGraph;
end.
```

GetImage procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Saves a bit image of the specified region into a buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure GetImage(X1, Y1, X2, Y2: Integer; var BitMap);</td>
</tr>
<tr>
<td>Remarks</td>
<td>X1, Y1, X2, and Y2 define a rectangular region on the screen. BitMap is an untyped parameter that must be greater than or equal to 6 plus the amount of area defined by the region. The first two words of BitMap store the width and height of the region. The third word is reserved. The remaining part of BitMap is used to save the bit image itself. Use the ImageSize function to determine the size requirements of BitMap.</td>
</tr>
</tbody>
</table>
GetImage procedure

Restrictions  Must be in graphics mode. The memory required to save the region must be less than 64K.

See also  ImageSize, PutImage

Example  uses Graph;
  var
    Gd, Gm: Integer;
    P: Pointer;
    Size: Word;
  begin
    Gd := Detect;
    InitGraph(Gd, Gm, '');
    if GraphResult <> grOk then
      Halt(1);
    Bar(0, 0, GetMaxX, GetMaxY);
    Size := ImageSize(10, 20, 30, 40);
    GetMem(P, Size); { Allocate memory on heap }
    GetImage(10, 20, 30, 40, P);
    Readln;
    ClearDevice;
    PutImage(100, 100, P, NormalPut);
    Readln;
    CloseGraph;
  end.

GetIntVec procedure

Purpose  Returns the address stored in a specified interrupt vector.

Declaration  procedure GetIntVec(IntNo: Byte; var Vector: Pointer);

Remarks  IntNo specifies the interrupt vector number (0..255), and the address is returned in Vector.

See also  SetIntVec

GetLineSettings procedure

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

Declaration  procedure GetLineSettings(var LineInfo: LineSettingsType);

Remarks  See page 103 for the declaration of LineSettingsType.

Restrictions  Must be in graphics mode.
GetLineSettings procedure

See also  *DrawPoly, LineSettingsType, Line Style, SetLineStyle*

Example

uses Graph;
var
  Gd, Gm: Integer;
  OldStyle: LineSettingsType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ";
  if GraphResult <> grOk then
    Halt(1);
  Line(0, 0, 100, 0);
  GetLineSettings(OldStyle);
  SetLineStyle(DottedLn, A, ThickWidth);
  Line(0, 0, 100, 0);
  with OldStyle do
    SetLineStyle(LineStyle, Pattern, Thickness);
  Line(0, 20, 100, 20);
  Readln;
  CloseGraph;
end.

GetMaxColor function

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

Declaration  function GetMaxColor: Word;

Remarks  As an example, on a 256K EGA, *GetMaxColor* always returns 15, which means that any call to *SetColor* with a value from 0..15 is valid. On a CGA in high-resolution mode or on a Hercules monochrome adapter, *GetMaxColor* returns a value of 1 because these adapters support only draw colors of 0 or 1.

Restrictions  Must be in graphics mode.

See also  *SetColor*

GetMaxMode function

Purpose  Returns the maximum mode number for the currently loaded driver.

Declaration  function GetMaxMode: Word;

Remarks  *GetMaxMode* lets you find out the maximum mode number for the current driver, directly from the driver. (Formerly, *GetModeRange* was the only way
you could get this number; GetModeRange is still supported, but only for the Borland drivers.)

The value returned by GetMaxMode is the maximum value that can be passed to SetGraphMode. Every driver supports modes 0..GetMaxMode.

**Restrictions**
Must be in graphics mode.

**See also**
GetModeRange, SetGraphMode

**Example**
uses Graph;
var  
  Driver, Mode: Integer;
  I: Integer;
begin  
  Driver := Detect;  
  InitGraph(Driver, Mode, '');  
  if GraphResult < 0 then  
    Halt(I);  
  for I := 0 to GetMaxMode do  
    OutTextXY(10, 10 * Succ(I), GetModeName(I));  
  Readln;  
  CloseGraph;
end.

---

**GetMaxX function**

**Purpose**
Returns the rightmost column (x resolution) of the current graphics driver and mode.

**Declaration**
function GetMaxX: Integer;

**Remarks**
Returns the maximum X value for the current graphics driver and mode. On a CGA in 320x200 mode, for example, GetMaxX returns 319.

GetMaxX and GetMaxY are invaluable for centering, determining the boundaries of a region on the screen, and so on.

**Restrictions**
Must be in graphics mode.

**See also**
GetMaxY, GetX, GetY, MoveTo

**Example**
uses Graph;
var  
  Gd, Gm: Integer;
begin  
  Gd := Detect;  
  InitGraph(Gd, Gm, '');  
  if GraphResult <> grOk then  
    Halt(I);
GetMaxX function

Rectangle(0, 0, GetMaxX, GetMaxY);
Readln;
CloseGraph;
end.

GetMaxY function

Purpose
Returns the bottommost row (y resolution) of the current graphics driver and mode.

Declaration
function GetMaxY: Integer;

Remarks
Returns the maximum y value for the current graphics driver and mode. On a CGA in 320x200 mode, for example, GetMaxY returns 199.

GetMaxX and GetMaxY are invaluable for centering, determining the boundaries of a region on the screen, and so on.

Restrictions
Must be in graphics mode.

See also
GetMaxX, GetX, GetY, MoveTo

Example
uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ");
  if GraphResult <> grOk then
    Halt(1);
  Rectangle(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.

GetMem procedure

Purpose
Allocates a block of memory of a specified size.

Declaration
procedure GetMem(var P: Pointer; Size: Word);

Remarks
P is a variable of any pointer type. Size is an expression specifying the size in bytes of the dynamic variable to allocate. The newly created variable can be referenced as P^.

If there isn’t enough free space in the heap to allocate the new variable, a run-time error occurs. (It is possible to avoid a run-time error; see “The HeapError variable” in Chapter 19 of the Language Guide.)
**GetMem procedure**

**Restrictions**  
The largest block that can be safely allocated on the heap at one time is 65,528 bytes (64K-$8$).

**See also**  
Dispose, FreeMem, HeapError, New

### GetModeName function

**Purpose**  
Returns a string containing the name of the specified graphics mode.

**Declaration**  
function GetModeName (ModeNumber: Integer): String;

**Remarks**  
The mode names are embedded in each driver. The return values (320×200 CGA P1, 640×200 CGA, and so on) are useful for building menus, display status, and so forth.

**Restrictions**  
Must be in graphics mode.

**See also**  
GetDriverName, GetMaxMode, GetModeRange

**Example**

```pascal
uses Graph;
var
  Driver, Mode: Integer;
  I: Integer;
begin
  Driver := Detect;  
  { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  for I := 0 to GetMaxMode do
    { Display all mode names }
    OutTextXY(10, 10 * Succ(I), GetModeName(I));
  Readln;
  CloseGraph;
end.
```

### GetModeRange procedure

**Purpose**  
Returns the lowest and highest valid graphics mode for a given driver.

**Declaration**  
procedure GetModeRange (GraphDriver: Integer; var LoMode, HiMode: Integer);

**Remarks**  
The output from the following program will be $Lowest = 0$ and $Highest = 1$:

```pascal
uses Graph;
var
  Lowest, Highest: Integer;
begin
  GetModeRange(EGA64, Lowest, Highest);
```
GetModeRange procedure

GetModeRange procedure

Write('Lowest = ', Lowest);
Write(' Highest = ', Highest);
end.

If the value of GraphDriver is invalid, the LoMode and HiMode are set to –1.

See also  DetectGraph, GetGraphMode, InitGraph, SetGraphMode

GetPalette procedure

Purpose

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 type PaletteType.

Restrictions

Must be in graphics mode, and can only be used with EGA, EGA 64, or VGA (not the IBM 8514 or the VGA in 256-color mode).

See also  GetDefaultPalette, GetPaletteSize, SetAllPalette, SetPalette

Example

uses Graph;
var
  Gd, Gm: Integer;
  Color: Word;
  Palette: PaletteType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ');
  if GraphResult <> grOk then
    Halt(1);
  GetPalette(Palette);
  if Palette.Size <> 1 then
    for Color := 0 to Pred(Palette.Size) do
      begin
        SetColor(Color);
        Line(0, Color * 5, 100, Color * 5);
      end
  else
    Line(0, 0, 100, 0);
  Readln;
  CloseGraph;
end.
GetPaletteSize function

Purpose
Returns the size of the palette color lookup table.

Declaration
function GetPaletteSize: Integer;

Remarks
GetPaletteSize reports how many palette entries can be set for the current graphics mode; for example, the EGA in color mode returns a value of 16.

Restrictions
Must be in graphics mode.

See also
GetDefaultPalette, GetMaxColor, GetPalette, SetPalette

GetPixel function

Purpose
Gets the pixel value at X, Y.

Declaration
function GetPixel(X, Y: Integer): Word;

Remarks
Gets the color of the pixel at (X, Y).

Restrictions
Must be in graphics mode.

See also
GetImage, PutImage, PutPixel, SetWriteMode

Example
uses Graph;
var
  Gd, Gm: Integer;
  PixelColor: Word;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  PixelColor := GetPixel(10, 10);
  if PixelColor = 0 then
    PutPixel(10, 10, GetMaxColor);
  Readln;
  CloseGraph;
end.
GetTextSettings procedure

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

Declaration
procedure GetTextSettings(var TextInfo: TextSettingsType);

Remarks
See page 55 for the declaration of the Font control constants and page 196
for a declaration of the TextSettingsType record.

Restrictions
Must be in graphics mode.

See also
Font control constants, InitGraph, SetTextJustify, SetTextStyle, TextHeight,
TextSettingsType, TextWidth

Example
uses Graph;
var
   Gd, Gm: Integer;
   OldStyle: TextSettingsType;
begin
   Gd := Detect;
   InitGraph(Gd, Gm, ');
   if GraphResult <> grOk then
      Halt(1);
   GetTextSettings(OldStyle);
   OutTextXY(0, 0, 'Old text style');
   SetTextJustify(LeftText, CenterText);
   SetTextStyle(TriplexFont, VertDir, 4);
   OutTextXY(GetMaxX div 2, GetMaxY div 2, 'New Style');
   with OldStyle do
      begin { Restore old text style }
         SetTextJustify(Horiz, Vert);
         SetTextStyle(Font, Direction, CharSize);
      end;
   OutTextXY(0, TextHeight('H'), 'Old style again');
   Readln;
   CloseGraph;
end.

GetTime procedure

Purpose
Returns the current time set in the operating system.

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

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

See also
GetDate, SetDate, SetTime, UnpackTime

GetVerify procedure

Purpose
Returns the state of the verify flag in DOS.

Declaration
procedure GetVerify(var Verify: Boolean);

Remarks
GetVerify returns the state of the verify flag in DOS. When off (False), disk writes are not verified. When on (True), all disk writes are verified to ensure proper writing.

See also
SetVerify

GetViewSettings procedure

Purpose
Returns the current viewport and clipping parameters, as set by SetViewPort.

Declaration
procedure GetViewSettings(var ViewPort: ViewPortType);

Remarks
GetViewSettings returns a variable of ViewPortType. See page 202 for a declaration of the record ViewPortType.

Restrictions
Must be in graphics mode.

See also
ClearViewPort, SetViewPort, ViewPortType

Example
uses Graph;
var
  Gd, Gm: Integer;
  ViewPort: ViewPortType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  GetViewSettings(ViewPort);
  with ViewPort do
    begin
      Rectangle(0, 0, X2 - X1, Y2 - Y1);
      if Clip then
        OutText('Clipping is active.')
GetViewSettings procedure

```pascal
else
    OutText('No clipping today.');
end;
Readln;
CloseGraph;
end.
```

GetX function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the X coordinate of the current position (CP).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>function GetX: Integer;</code></td>
</tr>
<tr>
<td>Remarks</td>
<td>The value of GetX is relative to the dimensions of the active viewport, as the following examples illustrate.</td>
</tr>
<tr>
<td></td>
<td>SetViewport(0, 0, GetMaxX, GetMaxY, True);</td>
</tr>
<tr>
<td></td>
<td>Moves CP to absolute (0, 0), and GetX returns a value of 0.</td>
</tr>
<tr>
<td></td>
<td>MoveTo(5, 5);</td>
</tr>
<tr>
<td></td>
<td>Moves CP to absolute (5, 5), and GetX returns a value of 5.</td>
</tr>
<tr>
<td></td>
<td>SetViewport(10, 10, 100, 100, True);</td>
</tr>
<tr>
<td></td>
<td>Moves CP to absolute (10, 10), but GetX returns a value of 0.</td>
</tr>
<tr>
<td></td>
<td>MoveTo(5, 5);</td>
</tr>
<tr>
<td></td>
<td>Moves CP to absolute (15, 15), but GetX returns a value of 5.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Must be in graphics mode.</td>
</tr>
<tr>
<td>See also</td>
<td>GetViewSettings, GetY, InitGraph, MoveTo, SetViewport</td>
</tr>
</tbody>
</table>
| Example | uses Graph;
var
   Gd, Gm: Integer;
   X, Y: Integer;
begin
   Gd := Detect;
   InitGraph(Gd, Gm, '');
   if GraphResult <> grOk then
      Halt(1);
   OutText('Starting here. ');
   X := GetX;
   Y := GetY;
   OutTextXY(20, 10, 'Now over here...');
   OutTextXY(X, Y, 'Now back over here.');  
   Readln;
   CloseGraph;
end.
**GetY function**

**Purpose**

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

**Declaration**

function GetY: Integer;

**Remarks**

The value of GetX is relative to the dimensions of the active viewport as the following examples illustrate.

- SetViewport(0, 0, GetMaxX, GetMaxY, True);
  - Moves CP to absolute (0, 0), and GetY returns a value of 0.
- MoveTo(5, 5);
  - Moves CP to absolute (5, 5), and GetY returns a value of 5.
- SetViewport(10, 10, 100, 100, True);
  - Moves CP to absolute (10, 10), but GetY returns a value of 0.
- MoveTo(5, 5);
  - Moves CP to absolute (15, 15), but GetY returns a value of 5.

**Restrictions**

Must be in graphics mode.

**See also**

GetViewSettings, GetX, InitGraph, MoveTo, SetViewport

**Example**

uses Graph;

var
  Gd, Gm: Integer;
  X, Y: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  OutText('Starting here. ');
  X := GetX;
  Y := GetY;
  OutTextXY(20, 10, 'Now over here...');
  OutTextXY(X, Y, 'Now back over here. ');
  Readln;
  CloseGraph;
end.

**GotoXY procedure**

**Purpose**

Moves the cursor to the given coordinates.

**Declaration**

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

Remarks
Moves the cursor to the position within the current window specified by X and Y (X is the column, Y is the row). The upper left corner is (1, 1).

This procedure is window-relative. The following example moves the cursor to the upper left corner of the active window (absolute coordinates (1, 10)):

```pascal
Window(1, 10, 60, 20);
GotoXY(1, 1);
```

Restrictions
If the coordinates are in any way invalid, the call to GotoXY is ignored.

See also
WhereX, WhereY, Window

GraphDefaults procedure

Purpose
Resets the graphics settings.

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 style, fill color, and fill pattern
- Active font, text style, text justification, and user Char size

Restrictions
Must be in graphics mode.

See also
InitGraph

GraphErrorMsg function

Purpose
Returns an error message string for the specified ErrorCode.

Declaration
```
function GraphErrorMsg(ErrorCode: Integer): String;
```

Remarks
This function returns a string containing an error message that corresponds with the error codes in the graphics system. This makes it easy for a user program to display a descriptive error message ("Device driver not found" instead of "error code -3").
GraphErrorMsg function

See also  *DetectGraph*, *GraphResult*, *InitGraph*

Example

```pascal
uses Graph;
var
  GraphDriver, GraphMode: Integer;
  ErrorCode: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '');
  ErrorCode := GraphResult;
  if ErrorCode <> grOk then
    begin
      Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
      Readln;
      Halt(1);
    end;
  Line(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.
```

GraphFreeMemPtr variable

**Purpose**
Holds the address of the heap deallocation routine.

**Declaration**
```pascal
var GraphFreeMemPtr: Pointer;
```

**Remarks**
Initially *GraphFreeMemPtr* points to the *Graph* unit’s heap deallocation routine. If your program does its own heap management, assign the address of your deallocation routine to this variable. See Chapter 17, “Using the Borland Graphics Interface,” in the *Language Guide* for additional information about this routine.

GraphGetMemPtr variable

**Purpose**
Holds the address of the heap allocation routine.

**Declaration**
```pascal
var GraphGetMemPtr: Pointer;
```

**Remarks**
Initially *GraphGetMemPtr* points to the *Graph* unit’s heap allocation routine. If your program does its own heap management, assign the address of your allocation routine to this variable. See 17, “Using the Borland Graphics Interface,” in the *Language Guide* for additional information about this routine.
GraphResult function

**Purpose**
Returns an error code for the last graphics operation.

**Declaration**
```pascal
function GraphResult: Integer;
```

**Remarks**
See page 85 for a list of the grXXXX constant values.

The following routines set `GraphResult`:
- Bar
- Bar3D
- ClearViewPort
- CloseGraph
- DetectGraph
- DrawPoly
- FillPoly
- FloodFill
- GetGraphMode
- ImageSize
- InitGraph
- InstallUserDriver
- InstallUserFont
- PieSlice
- RegisterBGIdriver
- RegisterBGIfont
- SetAllPalette
- SetFillPattern
- SetFillStyle
- SetGraphBufSize
- SetGraphMode
- SetLineStyle
- SetPalette
- SetTextJustify
- SetTextStyle

Note that `GraphResult` is reset to zero after it has been called (similar to `IOResult`). Therefore, the user should store the value of `GraphResult` into a temporary variable and then test it.

A string function, `GraphErrorMsg`, is provided to return a string that corresponds with each error code.

**See also**
`GraphErrorMsg`, `grXXXX constants`

**Example**
```pascal
uses Graph;
var
  ErrorCode: Integer;
  GrDriver, GrMode: Integer;
begin
  GrDriver := Detect;
  InitGraph(GrDriver, GrMode, '');
  ErrorCode := GraphResult; { Check for errors }
  if ErrorCode <> grOk then
  begin
    Writeln('Graphics error:');
    Writeln(GraphErrorMsg(ErrorCode));
    Writeln('Program aborted...');
    Halt(1);
  end;
end;
```
{ Do some graphics... }  
ClearDevice;  
Rectangle(0, 0, GetMaxX, GetMaxY);  
Readln;  
CloseGraph;  
end.

**grXXXX constants**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Used by the <strong>GraphResult</strong> function to indicate the type of error that occurred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remarks</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grOk</td>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>grNoInitGraph</td>
<td>-1</td>
<td>(BGI) graphics not installed (use InitGraph).</td>
</tr>
<tr>
<td>grNoDetected</td>
<td>-2</td>
<td>Graphics hardware not detected.</td>
</tr>
<tr>
<td>grFileNotFound</td>
<td>-3</td>
<td>Device driver file not found.</td>
</tr>
<tr>
<td>grInvalidDriver</td>
<td>-4</td>
<td>Invalid device driver file.</td>
</tr>
<tr>
<td>grNoLoadMem</td>
<td>-5</td>
<td>Not enough memory to load driver.</td>
</tr>
<tr>
<td>grNoScanMem</td>
<td>-6</td>
<td>Out of memory in scan fill.</td>
</tr>
<tr>
<td>grNoFloodMem</td>
<td>-7</td>
<td>Out of memory in flood fill.</td>
</tr>
<tr>
<td>grFontNotFound</td>
<td>-8</td>
<td>Font file not found.</td>
</tr>
<tr>
<td>grNoFontMem</td>
<td>-9</td>
<td>Not enough memory to load font.</td>
</tr>
<tr>
<td>grInvalidMode</td>
<td>-10</td>
<td>Invalid graphics mode for selected driver.</td>
</tr>
<tr>
<td>grError</td>
<td>-11</td>
<td>Graphics error (generic error); there is no room in the font table to register another font. (The font table holds up to 10 fonts, and only 4 are provided, so this error should not occur.)</td>
</tr>
<tr>
<td>grIOerror</td>
<td>-12</td>
<td>Graphics I/O error.</td>
</tr>
<tr>
<td>grInvalidFont</td>
<td>-13</td>
<td>Invalid font file; the font header isn't recognized.</td>
</tr>
<tr>
<td>grInvalidFontNum</td>
<td>-14</td>
<td>Invalid font number; the font number in the font header is not recognized.</td>
</tr>
</tbody>
</table>

**See also** **GraphResult**

**Halt procedure**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stops program execution and returns to the operating system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><strong>procedure Halt</strong> [ ( ExitCode: Word ) ];</td>
</tr>
<tr>
<td>Remarks</td>
<td><strong>ExitCode</strong> is an optional expression of type <strong>Word</strong> that specifies the program's exit code. <strong>Halt</strong> without a parameter corresponds to <strong>Halt(0)</strong>.</td>
</tr>
</tbody>
</table>
Halt procedure


See also Exit, RunError

HeapEnd variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Points to the end of DOS memory used by programs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var HeapEnd: Pointer;</td>
</tr>
<tr>
<td>Remarks</td>
<td>HeapEnd is initialized by the system unit when your program begins. See Chapter 19, “Memory issues,” in the Language Guide for more information.</td>
</tr>
<tr>
<td>See also</td>
<td>HeapOrg, HeapPtr</td>
</tr>
</tbody>
</table>

HeapError variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Points to the heap error function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var HeapError: Pointer;</td>
</tr>
</tbody>
</table>
| Remarks          | HeapError contains the address of a heap error function that is called whenever the heap manager can’t complete an allocation request. Install a heap error function by assigning its address to HeapError:  

HeapError := @HeapFunc;

| See also         | GetMem, New                        |

HeapOrg variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Points to the bottom of the heap.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var HeapOrg: Pointer;</td>
</tr>
<tr>
<td>Remarks</td>
<td>HeapOrg contains the address of the bottom of the heap. See Chapter 19, “Memory issues,” in the Language Guide for more information.</td>
</tr>
<tr>
<td>See also</td>
<td>HeapEnd, HeapPtr</td>
</tr>
</tbody>
</table>
## HeapPtr variable

**Purpose** Points to the top of the heap.

**Declaration** `var HeapPtr: Pointer;`

**Remarks** `HeapPtr` contains the address of the top of the heap, that is, the bottom of free memory. Each time a dynamic variable is allocated on the heap, the heap manager moves `HeapPtr` upward by the size of the variable. See Chapter 19, “Memory issues,” in the Language Guide for more information.

**See also** `HeapOrg, HeapEnd`

## Hi function

**Purpose** Returns the high-order byte of the argument.

**Declaration** `function Hi(X): Byte;`

**Remarks** `X` is an expression of type `Integer` or `Word`. `Hi` returns the high-order byte of `X` as an unsigned value.

**See also** `Lo, Swap`

**Example**
```pascal
var B: Byte;
beg
  B := Hi($1234);  { $12 }
end.
```

## High function

**Purpose** Returns the highest value in the range of the argument.

**Declaration** `function High(X);`

**Result type** `X, or the index type of X.`

**Remarks** `X` is either a type identifier or a variable reference. The type denoted by `X`, or the type of the variable denoted by `X`, must be an ordinal type, an array type, or a string type. For an ordinal type, `High` returns the highest value in the range of the type. For an array type, `High` returns the highest value within the range of the index type of the array. For a string type, `High` returns the declared size of the string. For an open array or string
High function

parameter, *High* returns a value of type *Word*, giving the number of elements in the actual parameter minus one element.

**See also** *Low*

**Example**

```pascal
function Sum(var X: array of Real): Real;
var
  I: Word;
  S: Real;
begin
  S := 0;
  for I := 0 to High(X) do S := S + X[I];
  Sum := S;
end;
```

HighVideo procedure

**Purpose** Selects high-intensity characters.

**Declaration**

```pascal
procedure HighVideo;
```

**Remarks**

There is a *Byte* variable in *Crt—TextAttr*—that is used to hold the current video attribute. *HighVideo* sets the high intensity bit of *TextAttr*'s foreground color, thus mapping colors 0–7 onto colors 8–15.

**See also** *LowVideo, NormVideo, TextBackground, TextColor*

**Example**

```pascal
uses Crt;
begin
  TextAttr := LightGray;
  HighVideo;
  { Color is now white }
end.
```

ImageSize function

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

**Declaration**

```pascal
function ImageSize(X1, Y1, X2, Y2: Integer): Word;
```

**Remarks**

*X1, Y1, X2, and Y2* define 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 several words. The first stores the width of the region, and the second stores the height. The next words store the attributes of the image itself. The last word is reserved.
If the memory required to save the region is greater than or equal to 64K, a value of 0 is returned and GraphResult returns -11 (grError).

Restrictions
Must be in graphics mode.

See also
GetImage, PutImage

Example
uses Graph;
var
  Gd, Gm: Integer;
  P: Pointer;
  Size: Word;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Bar(0, 0, GetMaxX, GetMaxY);
  Size := ImageSize(10, 20, 30, 40);
  GetMem(P, Size);
  GetImage(10, 20, 30, 40, P);
  Readln;
  ClearDevice;
  PutImage(100, 100, P, NormalPut);
  Readln;
  CloseGraph;
end.

Inc procedure

Purpose
Increments a variable.

Declaration
procedure Inc(var X [ ; N: Longint ]);

Remarks
X is an ordinal-type variable or a variable of type PChar if the extended syntax is enabled and N is an integer-type expression. X is incremented by 1, or by N if N is specified; that is, Inc(X) corresponds to X := X + 1, and Inc(X, N) corresponds to X := X + N.

Inc generates optimized code and is especially useful in tight loops.

See also
Dec, Pred, Succ

Example
var
  IntVar: Integer;
  LongIntVar: Longint;
Inc procedure

begin
  Inc(IntVar);
  Inc(IntVar, 1);  \{ IntVar := IntVar + 1 \}
  Inc(LongIntVar, 5);
  Inc(LongIntVar, 5); \{ LongIntVar := LongIntVar + 5 \}
end.

Include procedure

Purpose  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.

See also  Exclude

InitGraph procedure

Purpose  Initializes the graphics system and puts the hardware into graphics mode.

Declaration procedure InitGraph(var GraphDriver: Integer; var GraphMode: Integer;
                                 PathToDriver: String);

Remarks  If GraphDriver is equal to Detect, a call is made to any user-defined autodetect routines (see InstallUserDriver) and then DetectGraph. If graphics hardware is detected, the appropriate graphics driver is initialized, and a graphics mode is selected.

If GraphDriver is not equal to 0, the value of GraphDriver is assumed to be a driver number; that driver is selected, and the system is put into the mode specified by GraphMode. If you override autodetection in this manner, you must supply a valid GraphMode parameter for the driver requested.

PathToDriver specifies the directory path where the graphics drivers can be found. If PathToDriver is null, the driver files must be in the current directory.
Normally, `InitGraph` loads a graphics driver by allocating memory for the driver (through `GraphGetMem`), then loads the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BINOBJ utility), then placing calls to `RegisterBGIdriver` in your source code (before the call to `InitGraph`) to register the graphics driver(s). When you build your program, you must link the .OBJ files for the registered drivers. You can also load a BGI driver onto the heap and then register it using `RegisterBGIdriver`.

If memory for the graphics driver is allocated on the heap using `GraphGetMem`, that memory is released when a call is made to `CloseGraph`.

After calling `InitGraph`, `GraphDriver` is set to the current graphics driver, and `GraphMode` is set to the current graphics mode.

If an error occurs, both `GraphDriver` and `GraphResult` (a function) return one of the following `grXXXX` constant values: `grNotDetected`, `grFileNotFound`, `grInvalidDriver`, `grNoLoadMem`, `grInvalidMode`. See page 85 for a complete list of graphics error constants.

`InitGraph` resets all graphics settings to their defaults (current pointer, palette, color, viewport, and so on).

You can use `InstallDriver` to install a vendor-supplied graphics driver (see `InstallUserDriver` for more information).

**Restrictions**  
Must be in graphics mode. If you use the Borland Graphics Interface (BGI) on a Zenith Z-449 card, Turbo Pascal's autodetection code will always select the 640×480 enhanced EGA mode. If this mode isn't compatible with your monitor, select a different mode in the `InitGraph` call. Also, Turbo Pascal cannot autodetect the IBM 8514 graphics card (the autodetection logic recognizes it as VGA). Therefore, to use the IBM 8514 card, the `GraphDriver` variable must be assigned the value IBM8514 (which is defined in the `Graph` unit) when `InitGraph` is called. You should not use `DetectGraph` (or `Detect` with `InitGraph`) with the IBM 8514 unless you want the emulated VGA mode.

**See also**  
`CloseGraph`, `DetectGraph`, `GraphDefaults`, `GraphResult`, `grXXXX` constants, `InstallUserDriver`, `RegisterBGIdriver`, `RegisterBGIfont`, `RestoreCrtMode`, `SetGraphBufSize`, `SetGraphMode`

**Example**  
uses Graph;
var
  grDriver: Integer;
  grMode: Integer;
InitGraph procedure

ErrCode: Integer;
begin
  grDriver := Detect;
  InitGraph(grDriver, grMode,'');
  ErrCode := GraphResult;
  if ErrCode = grOk then
    begin
      Line(0, 0, GetMaxX, GetMaxY);
      Readln;
      CloseGraph;
    end
  else
    Writeln('Graphics error:', GraphErrorMsg(ErrCode));
end.

InOutRes variable

**Purpose**
Stores the value that the next call to IOResult returns.

**Declaration**
var InOutRes: Integer;

**Remarks**
InOutRes is used by the built-in I/O functions.

See also IOResult

Input variable

**Purpose**
Standard input file.

**Declaration**
var Input: Text;

**Remarks**
Input is a read-only file associated with the operating system's standard input file; usually this is the keyboard.

A number of Turbo Pascal's standard file handling procedures and functions allow the file variable parameter to be omitted, in which case, the procedure or function will instead operate on the Input or Output file variable. For example, Read(X) corresponds to Read(Input, X), and Write(X) corresponds to Write(Output, X). The following standard file handling procedures and functions operate on the Input file when no file parameter is specified: Eof, Eoln, Read, Readln, SeekEof, and SeekEoln.

See also  

Output

Insert procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Inserts a substring into a string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure Insert(Source: String; var S: String; Index: Integer);</td>
</tr>
<tr>
<td>Remarks</td>
<td>Source is a string-type expression. S is a string-type variable of any length. Index is an integer-type expression. Insert inserts Source into S at the Indexth position. If the resulting string is longer than 255 characters, it is truncated after the 255th character.</td>
</tr>
<tr>
<td>See also</td>
<td>Concat, Copy, Delete, Length, Pos</td>
</tr>
</tbody>
</table>
| Example | var S: string; begin 
  S := 'Honest Lincoln'; 
  Insert('Abe', S, 8); 
  { 'Honest Abe Lincoln' } |

InsLine procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Inserts an empty line at the cursor position.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure InsLine;</td>
</tr>
<tr>
<td>Remarks</td>
<td>All lines below the inserted line are moved down one line, and the bottom line scrolls off the screen (using the BIOS scroll routine). All character positions are set to blanks with the currently defined text attributes. Thus, if TextBackground is not black, the new line becomes the background color.</td>
</tr>
<tr>
<td>Example</td>
<td>InsLine is window-relative. The following example inserts a line 60 columns wide at absolute coordinates (1, 10): Window(1, 10, 60, 20); InsLine;</td>
</tr>
<tr>
<td>See also</td>
<td>DelLine, Window</td>
</tr>
</tbody>
</table>
### InstallUserDriver function

#### Purpose
Installs a vendor-added device driver to the BGI device driver table.

#### Declaration
```
function InstallUserDriver(Name: String; AutoDetectPtr: Pointer): Integer;
```

#### Remarks
`InstallUserDriver` lets you use a vendor-added device driver. The `Name` parameter is the file name of the new device driver. `AutoDetectPtr` is a pointer to an optional autodetect function that can accompany the new driver. This autodetect function takes no parameters and returns an integer value.

If the internal driver table is full, `InstallUserDriver` returns a value of -11 (`grError`); otherwise `InstallUserDriver` assigns and returns a driver number for the new device driver.

There are two ways to use this vendor-supplied driver. Let's assume you have a new video card called the Spiffy Graphics Array (SGA) and that the SGA manufacturer provided you with a BGI device driver (SGA.BGI). The easiest way to use this driver is to install it by calling `InstallUserDriver` and then passing the return value (the assigned driver number) directly to `InitGraph`:

```pascal
var Driver, Mode: Integer;
begin
  Driver := InstallUserDriver('SGA', nil);  { Table full? }
  if Driver = grError then
  begin
    Halt(1);  { Every driver supports mode of 0 }
    Mode := 0;
    InitGraph(Driver, Mode, '');  { Override autodetection }
    { Do graphics ... }
  end.
```

The `nil` value for the `AutoDetectPtr` parameter in the `InstallUserDriver` call indicates there isn't an autodetect function for the SGA.

The other, more general way to use this driver is to link in an autodetect function that will be called by `InitGraph` as part of its hardware-detection logic. Presumably, the manufacturer of the SGA gave you an autodetect function that looks something like this:

```pascal
{$F+}
function DetectSGA: Integer;
var Found: Boolean;
begin
  DetectSGA := grError;  { Assume it’s not there }
  Found := ...;  { Look for the hardware }
end.
```
if not Found then
    Exit;  { Returns -11 }  
DetectSGA := 3;  { Return recommended default video mode }
end;
{ $F- }

DetectSGA's job is to look for the SGA hardware at run time. If an SGA
isn't detected, DetectSGA returns a value of -11 (grError); otherwise, the
return value is the default video mode for the SGA (usually the best mix
of color and resolution available on this hardware).

Note that this function takes no parameters, returns a signed, integer-type
value, and must be a far call. When you install the driver (by calling
InstallUserDriver), you pass the address of DetectSGA along with the
device driver's file name:

    var Driver, Mode: Integer;
    begin
        Driver := InstallUserDriver('SGA', @DetectSGA);
        if Driver = grError then  { Table full? }
            Halt(1);
        Driver := Detect;
        InitGraph(Driver, Mode, ' ');
        { Discard SGA driver #; trust autodetection }
    end.

After you install the device driver file name and the SGA autodetect
function, you call InitGraph and let it go through its normal autodetection
process. Before InitGraph calls its built-in autodetection function
(DetectGraph), it first calls DetectSGA. If DetectSGA doesn't find the SGA
hardware, it returns a value of -11 (grError) and InitGraph proceeds with
its normal hardware detection logic (which might include calling any
other vendor-supplied autodetection functions in the order in which they
were "installed"). If, however, DetectSGA determines that an SGA is
present, it returns a nonnegative mode number, and InitGraph locates and
loads SGA.BGI, puts the hardware into the default graphics mode recom-
mended by DetectSGA, and finally returns control to your program.

See also
    GraphResult, InitGraph, InstallUserFont, RegisterBGIdriver, RegisterBGIfont

Example
    uses Graph;
    var
        Driver, Mode,
        TestDriver,
        ErrCode: Integer;
        {$F+}
function TestDetect: Integer;
{ Autodetect function: assume hardware is always present; return value =
  recommended default mode }
begin
  TestDetect := 1; { Default mode = 1 }
end;
{$F-}

begin
  { Install the driver }
  TestDriver := InstallUserDriver('TEST', @TestDetect);
  if GraphResult <> grOk then
  begin
    Writeln('Error installing TestDriver');
    Halt(1);
  end;
  Driver := Detect; { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  ErrCode := GraphResult;
  if ErrCode <> grOk then
  begin
    Writeln('Error during Init: ', ErrCode);
    Halt(1);
  end;
  OutText('Installable drivers supported...
  Readln;
  CloseGraph;
end.

InstallUserFont function

Purpose
Installs a new font not built into the BGI system.

Declaration
function InstallUserFont(FontFileName: String): Integer;

Remarks
FontFileName is the file name of a stroked font. InstallUserFont returns the
font ID number that can be passed to SetTextStyle to select this font. If the
internal font table is full, a value of DefaultFont will be returned.

See also
InstallUserDriver, RegisterBGIdriver, RegisterBGIfont, SetTextStyle

Example
uses Graph;
var
  Driver, Mode: Integer;
  TestFont: Integer;
begin
  TestFont := InstallUserFont('TEST'); { Install the font }
  if GraphResult <> grOk then
    begin
      Writeln('Error installing TestFont (using DefaultFont)');
      Readln;
    end;
  Driver := Detect; { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult <> grOk then
    Halt(1);
  SetTextStyle(TestFont, HorizDir, 2); { Use new font }
  OutText('Installable fonts supported...');
  Readln;
  CloseGraph;
end.

Int function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the integer part of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Int(X: Real): Real;</td>
</tr>
<tr>
<td>Remarks</td>
<td>X is a real-type expression. The result is the integer part of X; that is, X rounded toward zero.</td>
</tr>
<tr>
<td>See also</td>
<td>Frac, Round, Trunc</td>
</tr>
</tbody>
</table>
| Example     | var R: Real; begin
|             | R := Int(123.456); { 123.0 }            |
|             | R := Int(-123.456); { -123.0 }          |
|             | end.                                     |

Intr procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Executes a specified software interrupt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure Intr(IntNo: Byte; var Regs: Registers); {Dos}</td>
</tr>
<tr>
<td></td>
<td>procedure Intr(IntNo: Byte; var Regs: TRegisters); {WinDos}</td>
</tr>
<tr>
<td>Remarks</td>
<td>IntNo is the software interrupt number (0..255). Registers is a record defined in the Dos unit; TRegisters is a record defined in the WinDos unit. See page 141 for the declaration of Registers and page 198 for the declaration of TRegisters.</td>
</tr>
</tbody>
</table>
Before executing the specified software interrupt, *Intr* loads the 8086 CPU's AX, BX, CX, DX, BP, SI, DI, DS, and ES registers from the *Regs* record. When the interrupt completes, the contents of the AX, BX, CX, DX, BP, SI, DI, DS, ES, and Flags registers are stored back into the *Regs* record.

For details on writing interrupt procedures, see the section “Interrupt handling” in Chapter 20 of the *Language Guide*.

### Restrictions
Software interrupts that depend on specific values in SP or SS on entry, or modify SP and SS on exit, cannot be executed using this procedure.

### See also
- *Flag constants, MsDos, Register, TRegister*

---

### IOResult function

#### Purpose
Returns the status of the last I/O operation performed.

#### Declaration

```
function IOResult: Integer;
```

#### Remarks
I/O-checking must be off—{$I-}$—in order 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.

The codes returned are summarized in Chapter 4. A value of 0 reflects a successful I/O operation.

#### Example

```
var F: file of Byte;
begin
  ( Get file name command line )
  Assign(F, ParamStr(l));
  {$I-}
  Reset(F);
  {$I+}
  if IOResult = 0 then
    Writeln('File size in bytes: ', FileSize(F))
  else
    Writeln('File not found');
end.
```

#### See also
- *InOutRes*
Justification constants

Purpose
Constants that control horizontal and vertical justification.

Remarks

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftText</td>
<td>0</td>
</tr>
<tr>
<td>CenterText</td>
<td>1</td>
</tr>
<tr>
<td>RightText</td>
<td>2</td>
</tr>
<tr>
<td>BottomText</td>
<td>0</td>
</tr>
<tr>
<td>CenterText</td>
<td>1</td>
</tr>
<tr>
<td>TopText</td>
<td>2</td>
</tr>
</tbody>
</table>

See also SetTextJustify

Keep procedure

Purpose
Keep (or terminate and stay resident) terminates the program and makes it stay in memory.

Declaration
procedure Keep(ExitCode: Word);

Remarks
The entire program stays in memory—including data segment, stack segment, and heap—so be sure to specify a maximum size for the heap using the $M compiler directive. The ExitCode corresponds to the one passed to the Halt standard procedure.

Restrictions
Use with care! Terminate-and-stay-resident (TSR) programs are complex and no other support for them is provided. See the MS-DOS technical documentation for more information.

See also DosExitCode

KeyPressed function

Purpose
Returns True if a key has been pressed on the keyboard; False otherwise.

Declaration
function KeyPressed: Boolean;

Remarks
The character (or characters) is left in the keyboard buffer. KeyPressed does not detect shift keys like Shift, Alt, NumLock, and so on.

See also ReadKey

Chapter 1, Library reference
Example

uses Crt;
begin
  repeat
    Write('Xx');
  until KeyPressed;
end.

{ Fill the screen until a key is typed }

LastMode variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stores current video mode each time TextMode is called.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var LastMode: Word;</td>
</tr>
<tr>
<td>Remarks</td>
<td>At program startup, LastMode is initialized to the then-active video mode.</td>
</tr>
<tr>
<td>See also</td>
<td>TextMode</td>
</tr>
</tbody>
</table>

Length function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the dynamic length of a string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Length(S: String): Integer;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Returns the length of the String S.</td>
</tr>
<tr>
<td>See also</td>
<td>Concat, Copy, Delete, Insert, Pos</td>
</tr>
<tr>
<td>Example</td>
<td>var S: String;</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>Readln(S);</td>
</tr>
<tr>
<td></td>
<td>Writeln('&quot;&quot;, S, '&quot;');</td>
</tr>
<tr>
<td></td>
<td>Writeln('length = ', Length(S));</td>
</tr>
<tr>
<td></td>
<td>end.</td>
</tr>
</tbody>
</table>

Line procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Draws a line from the (X1, Y1) to (X2, Y2).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure Line(X1, Y1, X2, Y2: Integer);</td>
</tr>
<tr>
<td>Remarks</td>
<td>Draws a line in the style and thickness defined by SetLineStyle and uses the color set by SetColor. Use SetWriteMode to determine whether the line is copied or XORed to the screen.</td>
</tr>
</tbody>
</table>
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 is at one endpoint of the line. If you want the current pointer updated automatically when the line is drawn, use LineRel to draw a line a relative distance from the CP. Note that Line doesn’t update the current pointer.

Restrictions
Must be in graphics mode. Also, for drawing a horizontal line, Bar is faster than Line.

See also
GetLineStyle, LineRel, LineTo, MoveTo, Rectangle, SetColor, SetLineStyle, SetWriteMode

Example
uses Crt, Graph;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, ‘’);
if GraphResult <> grOk then
  Halt(1);
Randomize;
repeat
  Line(Random(200), Random(200), Random(200), Random(200));
until KeyPressed;
ReadLn;
CloseGraph;
end.

Line style constants

Purpose
Constants used to determine a line style and thickness; used with GetLineStyle and SetLineStyle.

Remarks
The following Line style constants are defined:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidLn</td>
<td>0</td>
</tr>
<tr>
<td>DottedLn</td>
<td>1</td>
</tr>
<tr>
<td>CenterLn</td>
<td>2</td>
</tr>
<tr>
<td>DashedLn</td>
<td>3</td>
</tr>
</tbody>
</table>
Line style constants

<table>
<thead>
<tr>
<th>UserBitLn</th>
<th>4 (user-defined line style)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormWidth</td>
<td>1</td>
</tr>
<tr>
<td>ThickWidth</td>
<td>3</td>
</tr>
</tbody>
</table>

See also LineSettingsType

LineRel procedure

**Purpose**

Draws a line to a point that is a relative distance from the current pointer (CP).

**Declaration**

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

**Remarks**

LineRel will draw a line from the current pointer to a point that is a relative (Dx, Dy) distance from the current pointer. The current line style and pattern, as set by SetLineStyle, are used for drawing the line and uses the color set by SetColor. Relative move and line commands are useful for drawing a shape on the screen whose starting point can be changed to draw the same shape in a different location on the screen. Use SetWriteMode to determine whether the line is copied or XORed to the screen.

The current pointer is set to the last point drawn by LineRel.

**Restrictions**

Must be in graphics mode.

**See also**

GetLineStyle, Line, LineTo, MoveRel, MoveTo, SetLineStyle, SetWriteMode

**Example**

```
uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');  // Use ' ' so that the default settings are used.
  if GraphResult <> grOk then
    Halt(1);
  MoveTo(1, 2);
  LineRel(100, 100);        // Draw to the point (101, 102)
  Readln;
  CloseGraph;
end.
```
LineSettingsType type

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

Declaration
```pascal
type
  LineSettingsType = record
   LineStyle: Word;
    Pattern: Word;
    Thickness: Word;
  end;
```

Remarks
See Line style constants for a list of defined line styles and thickness values.

See also
GetLineSettings, SetLineStyle

LineTo procedure

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

Declaration
```pascal
procedure LineTo(X, Y: Integer);
```

Remarks
Draws a line in the style and thickness defined by SetLineStyle and uses the color set by SetColor. Use SetWriteMode to determine whether the line is copied or XORed to the screen.

Note that
```pascal
  MoveTo(100, 100);
  LineTo(200, 200);
```
is equivalent to
```pascal
  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 relative distance from the CP. Note that the second method doesn't change the value of the current pointer.

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

Restrictions
Must be in graphics mode.

See also
GetLineStyle, Line, LineRel, MoveRel, MoveTo, SetLineStyle, SetWriteMode

Example
```pascal
  uses Crt, Graph;
  var Gd, Gm: Integer;
  ```
LineTo procedure

begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
    Randomize;
    repeat
      LineTo(Random(200), Random(200));
    until KeyPressed;
  Readln;
  CloseGraph;
end.

Ln function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the natural logarithm of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Ln(X: Real): Real;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Returns the natural logarithm of the real-type expression X.</td>
</tr>
<tr>
<td>See also</td>
<td>Exp</td>
</tr>
</tbody>
</table>

Lo function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the low-order byte of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Lo(X): Byte;</td>
</tr>
<tr>
<td>Remarks</td>
<td>X is an expression of type Integer or Word. Lo returns the low-order byte of X as an unsigned value.</td>
</tr>
<tr>
<td>See also</td>
<td>Hi, Swap</td>
</tr>
</tbody>
</table>
| Example | var B: Byte;
begin
  B := Lo($1234);  { $34 } 
end. |

Low function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the lowest value in the range of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Low(X);</td>
</tr>
<tr>
<td>Result type</td>
<td>X, or the index type of X.</td>
</tr>
</tbody>
</table>
Remarks

X is either a type identifier or a variable reference. The type denoted by X, or the type of the variable denoted by X, must be an ordinal type, an array type, or a string type. For an ordinal type, Low returns the lowest value in the range of the type. For an array type, Low returns the lowest value within the range of the index type of the array. For a string type, Low returns 0. For an open array or string parameter, Low returns 0.

See also

High

Example

```pascal
var
  A: array[1..100] of Integer;
  I: Integer;
begin
  for I := Low(A) to High(A) do A[I] := 0;
end.
type
  TDay = (Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday);
const
var
  Day: TDay;
  Hours: array[TDay] of 0..24;
begin
  for Day := Low(TDay) to High(TDay) do
  begin
    Write('Hours worked on ', DayName[Day], ' ', ? ');
    Readln(Hours[Day]);
  end;
end.
```

LowVideo procedure

Purpose

Selects low-intensity characters.

Declaration

```
procedure LowVideo;
```

Remarks

There is a Byte variable in Crt—TextAttr—that holds the current video attribute. LowVideo clears the high-intensity bit of TextAttr's foreground color, thus mapping colors 8 to 15 onto colors 0 to 7.

See also

HighVideo, NormVideo, TextBackground, TextColor
LowVideo procedure

Example

```pascal
uses Crt;
begin
  TextAttr := White;
  LowVideo;
end.
```

Lst variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stores the standard output as a text file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var Lst: Text;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Use Lst to send the output of your program to the printer.</td>
</tr>
<tr>
<td>See also</td>
<td>Assign, Rewrite</td>
</tr>
</tbody>
</table>

Example

```pascal
program PrintIt;
var
  Lst: Text;
begin
  Assign(Lst, 'LPT1');
  Rewrite(Lst);
  Writeln(Lst, 'Hello printer.');
  Close(Lst)
end.
```

MaxAvail function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the size of the largest contiguous free block in the heap, corresponding to the size of the largest dynamic variable that can be allocated at that time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function MaxAvail: Longint;</td>
</tr>
<tr>
<td>Remarks</td>
<td>MaxAvail returns the size of the largest contiguous free block in the heap, corresponding to the size of the largest dynamic variable that can be allocated at that time using New or GetMem. To find the total amount of free memory in the heap, call MemAvail. MaxAvail compares the size of the largest free block below the heap pointer to the size of free memory above the heap pointer, and returns the larger of the two values. Your program can specify minimum and maximum heap requirements using the $M$ directive.</td>
</tr>
<tr>
<td>See also</td>
<td>MemAvail</td>
</tr>
</tbody>
</table>
Example

type
  PBuffer = 'TBuffer;
  TBuffer = array[0..16383] of Char;
var Buffer: PBuffer;
begin
  :
  if MaxAvail < SizeOf(TBuffer) then OutOfMemory else
  begin
    New(Buffer);
    :
  end;
  :
end.

MaxColors constant

<table>
<thead>
<tr>
<th>Purpose</th>
<th>The constant that determines the maximum number of colors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>const MaxColors = 15;</td>
</tr>
<tr>
<td>See also</td>
<td>GetDefaultPalette, GetPalette, SetAllPalette</td>
</tr>
</tbody>
</table>

MemAvail function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Returns the amount of free memory in the heap.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function MemAvail: Longint;</td>
</tr>
<tr>
<td>Remarks</td>
<td>MemAvail returns the sum of the sizes of all free blocks in the heap. Note that a contiguous block of storage the size of the returned value is unlikely to be available due to fragmentation of the heap. To find the largest free block, call MaxAvail. MemAvail is calculated by adding the sizes of all free blocks below the heap pointer to the size of free memory above the heap pointer. Your program can specify minimum and maximum heap requirements using the $SM directive.</td>
</tr>
<tr>
<td>See also</td>
<td>MaxAvail</td>
</tr>
</tbody>
</table>
MemAvail function

Example

begin
    Writeln(MemAvail, ' bytes available');
    Writeln('Largest free block is ', MaxAvail, ' bytes');
end.

MkDir procedure

Purpose
Creates a subdirectory.

Declaration
procedure MkDir(S: String);

Remarks
Creates a new subdirectory with the path specified by string S. The last item in the path cannot be an existing file name.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
ChDir, GetDir, RmDir. CreateDir performs the same function as MkDir, but it takes a null-terminated string rather than a Pascal-style string.

Example

begin
    {$I-}
    MkDir(ParamStr(1));
    if IOResult <> 0 then
        Writeln('Cannot create directory')
    else
        Writeln('New directory created');
end.

Move procedure

Purpose
Copies a specified number of contiguous bytes from a source range to a destination range.

Declaration
procedure Move(var Source, Dest; Count: Word);

Remarks
Source and Dest are variable references of any type. Count is an expression of type Word. Move copies a block of Count bytes from the first byte occupied by Source to the first byte occupied by Dest. No checking is performed, so be careful with this procedure.

When Source and Dest are in the same segment, that is, when the segment parts of their addresses are equal, Move automatically detects and compensates for any overlap. Intrasegment overlaps never occur on
statically and dynamically allocated variables (unless they are deliberately forced); therefore, such deliberately forced overlaps are not detected.

Whenever possible, use SizeOf to determine Count.

See also FillChar

Example

```pascal
var
    A: array[1..4] of Char;
    B: Longint;
begin
    Move(A, B, SizeOf(A));  { SizeOf = safety! }
end.
```

MoveRel procedure

Purpose Moves the current pointer (CP) a relative distance from its current location.

Declaration procedure MoveRel(Dx, Dy: Integer);

Remarks MoveRel moves the current pointer (CP) to a point that is a relative \((Dx, Dy)\) distance from the current pointer. Relative move and line commands are useful for drawing a shape on the screen whose starting point can be changed to draw the same shape in a different location on the screen.

Restrictions Must be in graphics mode.

See also GetMaxX, GetMaxY, GetX, GetY, LineRel, LineTo, MoveTo

Example uses Graph;

```pascal
var Gd, Gm: Integer;
begin
    Gd := Detect;
    InitGraph(Gd, Gm, ' ');  
    if GraphResult <> grOk then
        Halt(1);
    MoveTo(1, 2);
    MoveRel(10, 10);  { Move to the point (11, 12) }
    PutPixel(GetX, GetY, GetMaxColor);
    Readln;
    CloseGraph;
end.
```
MoveTo procedure

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

**Declaration**
```pascal
procedure MoveTo(X, Y: Integer);
```

**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`
- `InitGraph`
- `LineRel`
- `LineTo`
- `MoveRel`
- `MoveTo`
- `OutText`
- `SetGraphMode`
- `SetViewPort`

If a viewport is active, the CP will be viewport-relative (the \(X\) and \(Y\) values will be added to the viewport's \(X1\) and \(Y1\) values). The CP is never clipped at the current viewport's boundaries.

**See also**
- `GetMaxX`, `GetMaxY`, `GetX`, `GetY`, `MoveRel`

**Example**
```pascal
uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ');
  if GraphResult <> grOk then
    Halt(1);
  MoveTo(0, 0); { Upper left corner of viewport }
  LineTo(GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.
```

MsDos procedure

**Purpose**
Executes a DOS function call.

**Declaration**
```pascal
procedure MsDos(var Regs: Registers); {Dos}
procedure MsDos(var Regs: TRegisters); {WinDos}
```

**Remarks**
The effect of a call to `MsDos` is the same as a call to `Intr` with an `IntNo` of $21$. `Registers` is a record defined in the `Dos` unit. `TRegisters` is defined in the `WinDos` unit. See page 141 for the declaration of `Registers` and page 198 for the declaration of `TRegisters`. 
Restrictions
Software interrupts that depend on specific calls in SP or SS on entry or modify SP and SS on exit cannot be executed using this procedure.

See also
Intr, Registers, TRegisters

New procedure

Purpose
Creates a new dynamic variable and sets a pointer variable to point to it.

Declaration
procedure New(var P: Pointer [ , Init: Constructor ]); 

Remarks
P is a variable of any pointer type. The size of the allocated memory block corresponds to the size of the type that P points to. The newly created variable can be referenced as P^0. If there isn't enough free space in the heap to allocate the new variable, a run-time error occurs. (It is possible to avoid a run-time error in this case; see "The HeapError variable" in Chapter 19 in the Language Guide.)

New allows a constructor call as a second parameter for allocating a dynamic object type variable. P is a pointer variable, pointing to an object type, and Init refers to a constructor of that object type.

An additional extension allows New to be used as a function, which allocates and returns a dynamic variable of a specified type. If the call is of the form New(T), T can be any pointer type. If the call is of the form New(T, Init), T must be a pointer to an object type, and Init must refer to a constructor of that object type. In both cases, the type of the function result is T.

See also
Dispose, FreeMem, GetMem, HeapError

NormVideo procedure

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

Declaration
procedure NormVideo;

Remarks
There is a Byte variable in Crt—TextAttr—that holds the current video attribute. NormVideo restores TextAttr to the value it had when the program was started.

See also
HighVideo, LowVideo, TextBackground, TextColor
NoSound procedure

Purpose
Turns off the internal speaker.

Declaration
procedure NoSound;

Remarks
The following program fragment emits a 440-hertz tone for half a second:

```
Sound(440);
Delay(500);
NoSound;
```

See also  

Sound

Odd function

Purpose
Tests if the argument is an odd number.

Declaration
function Odd(X: Longint): Boolean;

Remarks
X is an integer-type expression. The result is True if X is an odd number, and False if X is an even number.

Ofs function

Purpose
Returns the offset of a specified object.

Declaration
function Ofs(X): Word;

Remarks
X is any variable, or a procedure or function identifier. The result of type Word is the offset part of the address of X.

Restrictions
In protected-mode programs, Ofs should be used only on valid pointer addresses; pointing to an invalid pointer address will generate a general protection fault error message.

See also Addr, Seg

Ord function

Purpose
Returns the ordinal value of an ordinal-type expression.

Declaration
function Ord(X): Longint;
Remarks  X is an ordinal-type expression. The result is of type Longint and its value is the ordinality of X.

See also  Chr

Output variable  System

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Standard output file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var Output: Text;</td>
</tr>
<tr>
<td>Remarks</td>
<td>Output is a write-only file associated with the operating system’s standard output file, which is usually the display. A number of Turbo Pascal’s standard file handling procedures and functions allow the file variable parameter to be omitted, in which case the procedure or function will instead operate on the Input or Output file variable. For example, Read(X) corresponds to Read(Input, X), and Write(X) corresponds to Write(Output, X). The following standard file handling procedures and functions operate on the Output file when no file parameter is specified: Write, Writeln. See Chapter 13 “Input and output,” in the Language Guide for details about I/O issues.</td>
</tr>
<tr>
<td>See also</td>
<td>Input</td>
</tr>
</tbody>
</table>

OutText procedure  Graph

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Sends a string to the output device at the current pointer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure OutText(TextString: String);</td>
</tr>
<tr>
<td>Remarks</td>
<td>Displays TextString at the current pointer using the current justification settings. TextString is truncated at the viewport border if it is too long. If one of the stroked fonts is active, TextString is truncated at the screen boundary if it is too long. If the default (bit-mapped) font is active and the string is too long to fit on the screen, no text is displayed. OutText uses the font set by SetTextStyle. In order 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, center, rotate 90 degrees, and so on). The current pointer (CP) is updated by OutText only if the direction is horizontal, and the horizontal justification is left. Text output direction is</td>
</tr>
</tbody>
</table>
set by `SetTextStyle` (horizontal or vertical); text justification is set by `SetTextJustify` (CP at the left of the string, centered around CP, or CP at the right of the string—written above CP, below CP, or centered around CP). In the following example, block #1 outputs `ABCDEF` and moves CP (text is both horizontally output and left-justified); block #2 outputs `ABC` with `DEF` written right on top of it because text is right-justified; similarly, block #3 outputs `ABC` with `DEF` written right on top of it because text is written vertically.

```pascal
program CPupdate;
uses Graph;
var Driver, Mode: Integer;
begin
  Driver := Detect;
  InitGraph(Driver, Mode, '"
  if GraphResult < 0 then
    Halt(1);
  { #1 }
  MoveTo(0, 0);
  SetTextStyle(DefaultFont, HorizDir, 1); { CharSize = 1 }
  SetTextJustify(LeftText, TopText);
  OutText('ABC');
  OutText('DEF');
  { #2 }
  MoveTo(100, 50);
  SetTextStyle(DefaultFont, HorizDir, 1); { CharSize = 1 }
  SetTextJustify(RightText, TopText);
  OutText('ABC');
  OutText('DEF');
  { #3 }
  MoveTo(100, 100);
  SetTextStyle(DefaultFont, VertDir, 1); { CharSize = 1 }
  SetTextJustify(LeftText, TopText);
  OutText('ABC');
  OutText('DEF');
  Readln;
  CloseGraph;
end.
```

The CP is never updated by `OutTextXY`.

The default font (8×8) is not clipped at the screen edge. Instead, if any part of the string would go off the screen, no text is output. For example, the following statements would have no effect:
OutText procedure

```pascal
SetViewport(0, 0, GetMaxX, GetMaxY, ClipOn);
SetTextJustify(LeftText, TopText);
OutTextXY(-5, 0);  { -5,0 not onscreen }
OutTextXY(GetMaxX - 1, 0, 'ABC');  { Part of 'A', }
{ All of 'BC' off screen }

The stroked fonts are clipped at the screen edge, however.

Restrictions  Must be in graphics mode.

See also  GetTextSettings, OutTextXY, SetTextJustify, SetTextStyle, SetUserCharSize, TextHeight, TextWidth

Example  uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  OutText('Easy to use');
  Readln;
  CloseGraph;
end.
```

OutTextXY procedure

Purpose  Sends a string to the output device.

Declaration  procedure OutTextXY(X, Y: Integer; TextString: String);

Remarks  Displays TextString at (X, Y). TextString is truncated at the viewport border if it is too long. If one of the stroked fonts is active, TextString is truncated at the screen boundary if it is too long. If the default (bit-mapped) font is active and the string is too long to fit on the screen, no text is displayed.

Use OutText to output text at the current pointer; use OutTextXY to output text elsewhere on the screen.

procedure, OutTextXY and In order 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, center, rotate 90 degrees, and so forth).

Restrictions  Must be in graphics mode.
OutTextXY procedure

See also  
GetTextSettings, OutText, SetTextJustify, SetTextStyle, SetUserCharSize, TextHeight, TextWidth

Example  
uses Graph;
var
 Gd, Gm: Integer;
begin
 Gd := Detect;
 InitGraph(Gd, Gm, ');
 if GraphResult <> grOk then
  Halt(1);
 MoveTo(0, 0);
 OutText('Inefficient');
 Readln;
 OutTextXY(GetX, GetY, 'Also inefficient');
 Readln;
 ClearDevice;
 OutTextXY(0, 0, 'Perfect!');  { Replaces above }
 Readln;
 CloseGraph;
end.

OvrClearBuf procedure

Purpose  
Clears the overlay buffer.

Declaration  
procedure OvrClearBuf;

Remarks  
Disposes of all currently loaded overlays from the overlay buffer. This forces subsequent calls to overlaid routines to reload the overlays from the overlay file (or from EMS). If OvrClearBuf is called from an overlay, that overlay will immediately be reloaded upon return from OvrClearBuf.

The overlay manager never requires you to call OvrClearBuf; in fact, doing so will decrease performance of your application, since it forces overlays to be reloaded. OvrClearBuf is solely intended for special use, such as temporarily reclaiming the memory occupied by the overlay buffer.

See also  
OvrGetBuf, OvrSetBuf

OvrCodeList variable

Purpose  
Overlay code segment list.

Declaration  
var OvrCodeList: Word;
**OvrCodeList variable**

**Remarks**  *OvrCodeList* is initialized at link time by Turbo Pascal's linker and used internally by the overlay manager. It is zero if the program contains no overlays, or nonzero otherwise. You should never modify *OvrCodeList*.

---

**OvrDebugPtr variable**

**Purpose** Overlay debugger hook.

**Declaration** var OvrDebugPtr: Pointer;

**Remarks** *OvrDebugPtr* is used by Turbo Pascal's integrated debugger and by Turbo Debugger to implement debugging of overlaid programs. You should never modify *OvrDebugPtr*.

---

**OvrDosHandle variable**

**Purpose** Overlay file handle.

**Declaration** var OvrDosHandle: Word;

**Remarks** *OvrDosHandle* stores the file handle of the program's overlay file. *OvrDosHandle* is initialized by the *OvrInit* routine in the *Overlay* unit. A value of zero in *OvrDosHandle* indicates that the overlay file is not currently open. You should never modify *OvrDosHandle*.

**See also** *OvrInit*

---

**OvrEmsHandle variable**

**Purpose** Overlay EMS handle.

**Declaration** var OvrEmsHandle: Word;

**Remarks** *OvrEmsHandle* stores the handle of the expanded memory block containing the program's overlays. *OvrEmsHandle* is initialized by the *OverlnitEMS* routine in the *Overlay* unit. A value of $FFFF in *OvrEmsHandle* indicates that no expanded memory block has been allocated for overlays. You should never modify *OvrEmsHandle*.

**See also** *OvrInit*, *OverlnitEMS*
### OvrFileMode variable

**Purpose**
Determines the access code to pass to DOS when the overlay file is opened.

**Declaration**
```pascal
var OvrFileMode: Byte;
```

**Remarks**
The default `OvrFileMode` is 0, corresponding to read-only access. By assigning a new value to `OvrFileMode` before calling `OvrInit`, you can change the access code. You might change it to allow shared access on a network system, for example. For further details on access code values, see your DOS programmer's reference manual.

**See also** `OvrInit`

### OvrGetBuf function

**Purpose**
Returns the current size of the overlay buffer.

**Declaration**
```pascal
function OvrGetBuf: Longint;
```

**Remarks**
The size of the overlay buffer is set through a call to `OvrSetBuf`. Initially, the overlay buffer is as small as possible, corresponding to the size of the largest overlay. When an overlaid program is executed, a buffer of this size is automatically allocated. Because it includes both code and fix-up information for the largest overlay, however, the initial buffer size could be larger than 64K.

**See also** `OvrInit`, `OvrInitEMS`, `OvrSetBuf`

**Example**
```pascal
{$M 16384,65536,655360}
uses Overlay;
const ExtraSize = 49152; {48K}
begin
  OvrInit('EDITOR.OVR');
  Writeln('Initial size of overlay buffer is ', OvrGetBuf, ' bytes.');
  OvrSetBuf(OvrGetBuf+ExtraSize);
  Writeln('Overlay buffer now increased to ', OvrGetBuf, ' bytes.');
end.
```

### OvrGetRetry function

**Purpose**
Returns the current size of the probation area.

**Declaration**
```pascal
function OvrGetRetry: Longint;
```
OvrGetRetry function

Remarks

OvrGetRetry returns the current size of the probation area which is the value last set with OvrSetRetry.

See also

OvrSetRetry

OvrHeapEnd variable

System

Purpose

Overlay buffer end.

Declaration

var OvrHeapEnd: Word;

Remarks

OvrHeapEnd stores the segment address of the end of the overlay buffer. Except as specified in the description of OvrHeapOrg, you should never modify OvrHeapEnd.

See also

OvrHeapOrg, OvrSetBuf

OvrHeapOrg variable

System

Purpose

Overlay buffer origin.

Declaration

var OvrHeapOrg: Word;

Remarks

OvrHeapOrg stores the segment address of the start of the overlay buffer. The run-time library’s start-up code initializes OvrHeapOrg, OvrHeapPtr, and OvrHeapEnd to point to an overlay buffer between the program’s stack segment and heap. The size of this initial overlay buffer (in 16-byte paragraphs) is given by the OvrHeapSize variable, and it corresponds to the size of the largest overlay in the program, including fixup information for the overlay.

It is possible for a program to move the overlay buffer to another location in memory by assigning new values to OvrHeapOrg, OvrHeapPtr, and OvrHeapEnd. Any such relocation should be done before the call to OvrInit or right after a call to OvrClearBuf to ensure that the overlay buffer is empty. To move the overlay buffer, assign the segment address of the start of the buffer to OvrHeapOrg and OvrHeapPtr, and assign the segment address of the end of the buffer to OvrHeapEnd. You must ensure that the size of the buffer (calculated by OvrHeapEnd – OvrHeapOrg) is greater than or equal to OvrHeapSize.

See also

OvrHeapEnd, OvrHeapPtr, OvrSetBuf
OvrHeapPtr variable

Purpose Overlay buffer pointer.
Declaration `var OvrHeapPtr: Word;`
Remarks `OvrHeapPtr` is used internally by the overlay manager. Except as specified in the description of `OvrHeapOrg`, you should never modify `OvrHeapPtr`.
See also `OvrHeapOrg`

OvrHeapSize variable

Purpose Minimum overlay heap size.
Declaration `var OvrHeapSize: Word;`
Remarks `OvrHeapSize` contains the minimum size of the overlay buffer in 16-byte paragraphs. `OvrHeapSize` is initialized at link time to contain the size of the largest overlay in the program, including fixup information for the overlay. It is zero if the program contains no overlays. You should never modify `OvrHeapSize`.
See also `OvrHeapOrg`

OvrInit procedure

Purpose Initializes the overlay manager and opens the overlay file.
Declaration `procedure OvrInit(FileName: String);`
Remarks If `FileName` does not specify a drive or a subdirectory, the overlay manager searches for the file in the current directory, in the directory that contains the .EXE file (if running under DOS 3.x or later), and in the directories specified in the PATH environment variable.

Errors are reported in the `OvrResult` variable. `ovrOk` indicates success. `ovrError` means that the overlay file is of an incorrect format, or that the program has no overlays. `ovrNotFound` means that the overlay file could not be located.

In case of error, the overlay manager remains uninstalled, and an attempt to call an overlaid routine will produce run-time error 208 ("Overlay manager not installed").
OvrInit procedure

OvrInit must be called before any of the other overlay manager procedures.

See also  
OvrGetBuf, OvrInitEMS, OvrSetBuf

Example uses Overlay;
begin
OvrInit('EDITOR.OVR');
if OvrResult <> ovrOk then
begin
  case OvrResult of
    ovrError: Writeln('Program has no overlays.');
    ovrNotFound: Writeln('Overlay file not found.');
  end;
  Halt(1);
end;
end.

OvrInitEMS procedure

Purpose Loads the overlay file into EMS if possible.

Declaration procedure OvrInitEMS;

Remarks If an EMS driver can be detected and if enough EMS memory is available, OvrInitEMS loads all overlays into EMS and closes the overlay file. Subsequent overlay loads are reduced to fast in-memory transfers. OvrInitEMS installs an exit procedure, which automatically deallocates EMS memory upon termination of the program.

Errors are reported in the OvrResult variable. ovrOk indicates success. ovrError means that OvrInit failed or was not called. ovrIOError means that an I/O error occurred while reading the overlay file. ovrNoEMSDriver means that an EMS driver could not be detected. ovrNoEMSMemory means that there is not enough free EMS memory available to load the overlay file.

In case of error, the overlay manager will continue to function, but overlays will be read from disk.

The EMS driver must conform to the Lotus/Intel/Microsoft Expanded Memory Specification (EMS). If you are using an EMS-based RAM disk, make sure that the command in the CONFIG.SYS file that loads the RAM-disk driver leaves some unallocated EMS memory for your overlaid applications.

See also  
OvrGetBuf, OvrInit, OvrResult, OvrSetBuf
Example

```pascal
uses Overlay;
begin
  OvrInit('EDITOR.OVR');
  if OvrResult <> ovrOk then begin
    Writeln('Overlay manager initialization failed.');
    Halt(1);
  end;
  OvrInitEMS;
  case OvrResult of
    ovrI0Error: Writeln('Overlay file I/O error.');
    ovrNoEMSDriver: Writeln('EMS driver not installed.');
    ovrNoEMSMemory: Writeln('Not enough EMS memory.');
    else Writeln('Using EMS for faster overlay swapping.');
  end;
end;
```

---

**OvrLoadCount variable**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Overlay load count.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var OvrLoadCount: Word;</td>
</tr>
<tr>
<td>Remarks</td>
<td>The initial value of <strong>OvrLoadCount</strong> is zero. The overlay manager increments <strong>OvrLoadCount</strong> each time an overlay is loaded. By examining <strong>OvrTrapCount</strong> and <strong>OvrLoadCount</strong> in the Debugger's Watch window during identical runs of an application, you can monitor the effect of different probation area sizes (set with <strong>OvrSetRetry</strong> to find the optimal size for your particular application.</td>
</tr>
<tr>
<td>See also</td>
<td><strong>OvrTrapCount</strong></td>
</tr>
</tbody>
</table>

**OvrLoadList variable**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Loaded overlays list.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var OvrLoadList: Word;</td>
</tr>
<tr>
<td>Remarks</td>
<td><strong>OvrLoadList</strong> is used internally by the overlay manager. You should never modify <strong>OvrLoadList</strong>.</td>
</tr>
</tbody>
</table>
### OvrReadBuf variable

| **Purpose** | Overlay read function pointer. |
| **Declaration** | `type OvrReadFunc = function(OvrSeg: Word): Integer;`  
`var OvrReadBuf: OvrReadFunc;` |
| **Remarks** | `OvrLoadList` lets you intercept overlay load operations to implement error handling, for example, or to check that a removable disk is present. Whenever the overlay manager needs to read an overlay, it calls the function whose address is stored in `OvrReadBuf`. If the function returns zero, the overlay manager assumes that the operation was successful; if the function result is nonzero, run-time error 209 is generated. The `OvrSeg` parameter indicates what overlay to load, but you'll never need to access this information. See Chapter 18, "Using overlays," in the *Language Guide* for details about installing your own overlay read function. |

### OvrResult variable

| **Purpose** | Result code for last overlay procedure call. |
| **Declaration** | `var OvrResult: Integer;` |
| **Remarks** | Before returning, each of the procedure in the *Overlay* unit stores a result code in the `ovrResult` variable. Possible `OvrXXXX` return codes are listed on page 125. In general, a value of zero indicates success. The `OvrResult` variable resembles the `IOResult` standard function except that `OvrResult` is not set to zero once it is accessed. Thus, there is no need to copy `OvrResult` into a local variable before it is examined.  
**See also** `OvrInit, OvrInitEMS, OvrSetBuf` |

### OvrSetBuf procedure

| **Purpose** | Sets the size of the overlay buffer. |
| **Declaration** | `procedure OvrSetBuf(BufSize: Longint);` |
| **Remarks** | `BufSize` must be larger than or equal to the initial size of the overlay buffer, and less than or equal to `MemAvail + OvrGetBuf`. The initial size of the overlay buffer is the size returned by `OvrGetBuf` before any calls to `OvrSetBuf`. |
OvrSetBuf procedure

If the specified size is larger than the current size, additional space is allocated from the beginning of the heap, thus decreasing the size of the heap. Likewise, if the specified size is less than the current size, excess space is returned to the heap.

OvrSetBuf requires that the heap be empty; an error is returned if dynamic variables have already been allocated using New or GetMem. For this reason, make sure to call OvrSetBuf before the Graph unit's InitGraph procedure; InitGraph allocates memory on the heap and—once it has done so—all calls to OvrSetBuf will be ignored.

If you are using OvrSetBuf to increase the size of the overlay buffer, you should also include a $M compiler directive in your program to increase the minimum size of the heap accordingly.

Errors are reported in the OvrResult variable. ovrOk indicates success. ovrError means that OvrInit failed or was not called, that BufSize is too small, or that the heap is not empty. ovrNoMemory means that there is not enough heap memory to increase the size of the overlay buffer.

See also OvrGetBuf, OvrInit, OvrInitEMS, OvrResult, ovrXXXX constants

Example

{$M 16384,65536,655360}
uses Overlay;
const ExtraSize = 49152; {48K}
begin
  OvrInit('EDITOR.OVR');
  OvrSetBuf(OvrGetBuf + ExtraSize);
end.

OvrSetRetry procedure

Purpose
Sets the size of the probation area in the overlay buffer.

Declaration
procedure OvrSetRetry(Size: Longint);

Remarks
If an overlay falls within the Size bytes before the overlay buffer tail, it is automatically put on probation. Any free space in the overlay buffer is considered part of the probation area. For reasons of compatibility with earlier versions of the overlay manager, the default probation area size is zero, which effectively disables the probation/reprieval mechanism.

There is no empirical formula for determining the optimal size of the probationary area; however, experiments have shown that values ranging from one-third to one-half of the overlay buffer size provide the best results.
See also *OvrGetRetry*

Example

Here's an example of how to use *OvrSetRetry*:

```pascal
OvrInit('MYPROG.OVR');
OvrSetBuf(BufferSize);
OvrSetRetry(BufferSize div 3);
```

---

**OvrTrapCount variable**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Overlay call interception count.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>var OvrTrapCount: Word;</code></td>
</tr>
<tr>
<td>Remarks</td>
<td>Each time a call to an overlaid routine is intercepted by the overlay manager, either because the overlay is not in memory or because the overlay is on probation, the <em>OvrTrapCount</em> variable is incremented. The initial value of <em>OvrTrapCount</em> is zero.</td>
</tr>
<tr>
<td>See also</td>
<td><em>OvrLoadCount</em></td>
</tr>
</tbody>
</table>

---

**ovrXXXX constants**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Return codes stored in the <em>OvrResult</em> variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remarks</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ovrOk</td>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>ovrError</td>
<td>-1</td>
<td>Overlay manager error</td>
</tr>
<tr>
<td>ovrNotFound</td>
<td>-2</td>
<td>Overlay file not found</td>
</tr>
<tr>
<td>ovrNoMemory</td>
<td>-3</td>
<td>Not enough memory for overlay buffer</td>
</tr>
<tr>
<td>ovrIOError</td>
<td>-4</td>
<td>Overlay file I/O error</td>
</tr>
<tr>
<td>ovrNoEMSDriver</td>
<td>-5</td>
<td>EMS driver not installed</td>
</tr>
<tr>
<td>ovrNoEMSMemory</td>
<td>-6</td>
<td>Not enough EMS memory</td>
</tr>
</tbody>
</table>

---

**PackTime procedure**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Converts a <em>DateTime</em> record into a 4-byte, packed date-and-time <em>Longint</em> used by <em>SetFTime</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>procedure PackTime(var DT: DateTime; var Time: Longint);</code> {Dos}</td>
</tr>
<tr>
<td></td>
<td><code>procedure PackTime(var DT: TDateTime; var Time: Longint);</code> {WinDos}</td>
</tr>
</tbody>
</table>
PackTime procedure

Remarks
The fields of the DateTime record are not range-checked. DateTime is a DOS record; use TDateTime if you are writing a program using WinDos. See page 26 for the declaration of DateTime or page 189 for the TDateTime declaration.

See also GetFTime, GetTime, SetFTime, SetTime, UnpackTime

PaletteType type

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

Declaration
type
  PaletteType = record
    Size: Byte;
    Colors: array[0..MaxColors] of Shortint;
  end;

PaletteType is defined as follows:

const
  MaxColors = 15;
type
  PaletteType = record
    Size: Byte;
    Colors: array[0..MaxColors] of Shortint;
  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.

ParamCount function

Purpose
Returns the number of parameters passed to the program on the command line.

Declaration
function ParamCount: Word;

Remarks
Blanks and tabs serve as separators.

See also ParamStr

System
ParamCount function

Example

begin
  if ParamCount = 0 then
    Writeln('No parameters on command line')
  else
    Writeln(ParamCount, ' parameter(s)');
end.

ParamStr function

Purpose
Returns a specified command-line parameter.

Declaration
function ParamStr(Index): String;

Remarks
Index is an expression of type Word. ParamStr returns the Indexth parameter from the command line, or an empty string if Index is greater than ParamCount. ParamStr(0) returns the path and file name of the executing program (for example, C:\TP\MYPROG.EXE).

See also
ParamCount

Example
var I: Word;
begin
  for I := 1 to ParamCount do
    Writeln(ParamStr(I));
end.

Pi function

Purpose
Returns the value of pi (3.1415926535897932385).

Declaration
function Pi: Real;

Remarks
Precision varies, depending on whether the compiler is in 80x87 or software-only mode.

PieSlice procedure

Purpose
Draws and fills a pie slice, using (X, Y) as the center point and drawing from start angle to end angle.

Declaration
procedure PieSlice(X, Y: Integer; StAngle, EndAngle, Radius: Word);

Remarks
The pie slice is outlined using the current color, and filled using the pattern and color defined by SetFillStyle or SetFillPattern.
PieSlice procedure

Each graphics driver contains an aspect ratio that is used by Circle, Arc, and PieSlice. A start angle of 0 and an end angle of 360 will draw and fill a complete circle. The angles for Arc, Ellipse, and PieSlice are counterclockwise with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on.

If an error occurs while filling the pie slice, GraphResult returns a value of grNoScanMem.

Restrictions
Must be in graphics mode.

See also
Arc, Circle, Ellipse, FillEllipse, GetArcCoords, GetAspectRatio, Sector, SetFillStyle, SetFillPattern, SetGraphBufSize

Example
uses Graph;
const Radius = 30;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, ");
if GraphResult <> grOk then
Halt(1);
PieSlice(100, 100, 0, 270, Radius);
Readln;
CloseGraph;
end.

PointType type

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

Déclaration
type
PointType = record
  X, Y: Integer;
end;

Pos function

Searches for a substring in a string.

Declaration
function Pos(Substr, S: String): Byte;

Remarks
Substr and S are string-type expressions. Pos searches for Substr within S, and returns an integer value that is the index of the first character of Substr within S. If Substr is not found, Pos returns zero.
Pos function

See also Concat, Copy, Delete, Insert, Length

Example

```pascal
var S: String;
begin
  S := ' 123.5';
  while Pos(' ', S) > 0 do
    S[Pos(' ', S)] := '0';
end. { Convert spaces to zeros }
```

Pred function

Purpose
Returns the predecessor of the argument.

Declaration
```pascal
function Pred(X);
```

Remarks
X is an ordinal-type expression. The result, of the same type as X, is the predecessor of X.

See also Dec, Inc, Succ

PrefixSeg variable

Purpose
Contains the segment address of the Program Segment Prefix (PSP) created by DOS when the application executes.

Declaration
```pascal
var PrefixSeg: Word;
```

Remarks
For a complete description of the Program Segment Prefix, see your DOS manuals.

Ptr function

Purpose
Converts a segment base and an offset address to a pointer-type value.

Declaration
```pascal
function Ptr(Seg, Ofs: Word): Pointer;
```

Remarks
Seg and Ofs are expressions of type Word. The result is a pointer that points to the address given by Seg and Ofs. Like nil, the result of Ptr is assignment compatible with all pointer types.

The function result can be dereferenced and typecast:
```pascal
if Byte(Ptr(Seg0040, $49)) = 7 then
  Writeln('Video mode = mono');
```

See also Addr, Ofs, Seg
Ptr function

Example

```pascal
var P: ^Byte;
begin
  P := Ptr(Seg0040, $49);
  Writeln('Current video mode is ', P);
end.
```

PutImage procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Puts a bit image onto the screen.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>procedure PutImage(X, Y: Integer; var BitMap; BitBlt: Word);</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>(X, Y)</code> is the upper left corner of a rectangular region on the screen. <code>BitMap</code> is an untyped parameter that contains the height and width of the region, and the bit image that will be put onto the screen. <code>BitBlt</code> specifies which binary operator will be used to put the bit image onto the screen. See page 12 for a list of <code>BitBlt</code> operators. Each constant corresponds to a binary operation. For example, <code>PutImage(X, Y, BitMap, NormalPut)</code> puts the image stored in <code>BitMap</code> at <code>(X, Y)</code> using the assembly language <code>MOV</code> instruction for each byte in the image. Similarly, <code>PutImage(X, Y, BitMap, XORPut)</code> puts the image stored in <code>BitMap</code> at <code>(X, Y)</code> using the assembly language <code>XOR</code> instruction for each byte in the image. This is an often-used animation technique for &quot;dragging&quot; an image around the screen. <code>PutImage(X, Y, BitMap, NotPut)</code> inverts the bits in <code>BitMap</code> and then puts the image stored in <code>BitMap</code> at <code>(X, Y)</code> using the assembly language <code>MOV</code> for each byte in the image. Thus, the image appears in inverse video of the original <code>BitMap</code>. Note that <code>PutImage</code> is never clipped to the viewport boundary. Moreover—with one exception—it is not actually clipped at the screen edge either. Instead, if any part of the image would go off the screen, no image is output. In the following example, the first image would be output, but the middle three <code>PutImage</code> statements would have no effect:</td>
</tr>
</tbody>
</table>

```pascal
program NoClip;
uses Graph;
var
  Driver, Mode: Integer;
P: Pointer;
```
begin
  Driver := Detect;
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  SetViewPort(0, 0, GetMaxX, GetMaxY, ClipOn);
  GetMem(p, ImageSize(0, 0, 99, 49));
  PieSlice(50, 25, 0, 360, 45);
  GetImage(0, 0, 99, 49, P^);
  ClearDevice;
  PutImage(GetMaxX - 99, 0, P^, NormalPut);
  PutImage(GetMaxX - 98, 0, P^, NormalPut);
  PutImage(-1, 0, P^, NormalPut);
  PutImage(0, -1, P^, NormalPut);
  PutImage(0, GetMaxY - 30, P^, NormalPut);
  Readln;
  CloseGraph;
end.

In the last PutImage statement, the height is clipped at the lower screen edge, and a partial image is displayed. This is the only time any clipping is performed on PutImage output.

Restrictions
Must be in graphics mode.

See also
BitBlt operators, GetImage, ImageSize

Example
uses Graph;
var
  Gd, Gm: Integer;
  P: Pointer;
  Size: Word;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  Bar(0, 0, GetMaxX, GetMaxY);
  Size := ImageSize(10, 20, 30, 40);
  GetMem(P, Size);
  GetImage(10, 20, 30, 40, P^);
  Readln;
  CloseDevice;
PutImage procedure

```
PutImage(100, 100, P^, NormalPut);
Readln;
CloseGraph;
end.
```

PutPixel procedure

### Purpose
Plots a pixel at X, Y.

### Declaration
```
procedure PutPixel(X, Y: Integer; Pixel: Word);
```

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

### Restrictions
Must be in graphics mode.

### See also
GetImage, GetPixel, PutImage

### Example
```
uses Crt, Graph;

var
  Gd, Gm: Integer;
  Color: Word;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
      Halt(1);
  Color := GetMaxColor;
  Randomize;
  repeat
      PutPixel(Random(100), Random(100), Color); { Plot "stars" }
      Delay(10);
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

Random function

### Purpose
Returns a random number.

### Declaration
```
function Random [ { Range: Word } ];
```

### Result type
Real or Word, depending on the parameter

### Remarks
If Range is not specified, the result is a Real-type random number within the range 0 <= X < 1. If Range is specified, it must be an expression of
type *Word*, and the result is a *Word-type* random number within the range
\(0 \leq X < \text{Range}\). If *Range* equals 0, a value of 0 is returned.

The random number generator should be initialized by making a call to
*Randize*, or by assigning a value to *RandSeed*.

### See also
*Randize*

#### Randomize procedure

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Initializes the built-in random generator with a random value.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>procedure Randomize;</code></td>
</tr>
</tbody>
</table>
| **Remarks** | The random value is obtained from the system clock. The random number
generator's seed is stored in a predeclared *Longint* variable called
*RandSeed*. |
| **See also** | *Random* |

#### RandSeed variable

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Stores the built-in random number generator's seed.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>var RandSeed: Longint;</code></td>
</tr>
</tbody>
</table>
| **Remarks** | By assigning a specific value to *RandSeed*, a specific sequence of random
numbers can be generated over and over. This is particularly useful in
applications that deal with data encryption, statistics, and simulations. |
| **See also** | *Random, Randize* |

#### Read procedure (text files)

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Reads one or more values from a text file into one or more variables.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>procedure Read( [ var F: Text; ] V1 [, V2, ..., VN ] );</code></td>
</tr>
</tbody>
</table>
| **Remarks** | *F*, if specified, is a text file variable. If *F* is omitted, the standard file
variable *Input* is assumed. Each *V* is a variable of type *Char, Integer, Real,*
or *String*. |

- With a type *Char* variable, *Read* reads one character from the file and
assigns that character to the variable. If *Eof(F)* was *True* before *Read* was
executed, the value *Chr(26)* (a Ctrl+Z character) is assigned to the
variable. If *Eoln(F)* was *True*, the value *Chr(13)* (a carriage-return

---

Chapter 1, Library reference
Read procedure (text files)

character) is assigned to the variable. The next Read starts with the next character in the file.

- With a type integer variable, Read expects a sequence of characters that form a signed whole number according to the syntax illustrated in section "Numbers" in Chapter 2 of the Language Guide. Any blanks, tabs, or end-of-line markers preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line marker following the numeric string or if Eof(F) becomes True. If the numeric string does not conform to the expected format, an I/O error occurs; otherwise, the value is assigned to the variable. If Eof(F) was True before Read was executed or if Eof(F) becomes True while skipping initial blanks, tabs, and end-of-line markers, the value 0 is assigned to the variable. The next Read will start with the blank, tab, or end-of-line marker that terminated the numeric string.

- With a type real variable, Read expects a sequence of characters that form a signed whole number (except that hexadecimal notation is not allowed). Any blanks, tabs, or end-of-line markers preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line marker following the numeric string or if Eof(F) becomes True. If the numeric string does not conform to the expected format, an I/O error occurs; otherwise, the value is assigned to the variable. If Eof(F) was True before Read was executed, or if Eof(F) becomes True while skipping initial blanks, tabs, and end-of-line markers, the value 0 is assigned to the variable. The next Read will start with the blank, tab, or end-of-line marker that terminated the numeric string.

- With a type string variable, Read reads all characters up to, but not including, the next end-of-line marker or until Eof(F) becomes True. The resulting character string is assigned to the variable. If the resulting string is longer than the maximum length of the string variable, it is truncated. The next Read will start with the end-of-line marker that terminated the string.

- When the extended syntax is enabled, Read can also be used to read null-terminated strings into zero-based character arrays. With a character array of the form array[0..N] of Char, Read reads up to N characters, or until Eoln(F) or Eof(F) become True, and then appends a NULL (#0) terminator to the string.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Read with a type string variable does not skip to the next line after reading. For this reason, you cannot use successive Read calls to read a sequence of strings because you’ll never get past the first line; after the
Read procedure (text files)

First `Read`, each subsequent `Read` will see the end-of-line marker and return a zero-length string. Instead, use multiple `Readln` calls to read successive string values.

**See also** `Readln`, `Write`, `Writeln`

Read procedure (typed files)

**Purpose** Reads a file component into a variable.

**Declaration**

```pascal
procedure Read(F, V1 [, V2, ..., VN ]);
```

**Remarks**

`F` is a file variable of any type except text, and each `V` is a variable of the same type as the component type of `F`. For each variable read, the current file position is advanced to the next component. An error occurs if you attempt to read from a file when the current file position is at the end of the file; that is, when `Eof(F)` is `True`.

With `{SI-}`, `IOResult` returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

**Restrictions**

File must be open.

**See also** `Write`

ReadKey function

**Purpose** Reads a character from the keyboard.

**Declaration**

```pascal
function ReadKey: Char;
```

**Remarks**

The character read is not echoed to the screen. If `KeyPressed` was `True` before the call to `ReadKey`, the character is returned immediately. Otherwise, `ReadKey` waits for a key to be typed.

The special keys on the PC keyboard generate extended scan codes. Special keys are the function keys, the cursor control keys, `Alt` keys, and so on. When a special key is pressed, `ReadKey` first returns a null character (`#0`), and then returns the extended scan code. Null characters cannot be generated in any other way, so you are guaranteed the next character will be an extended scan code.

The following program fragment reads a character or an extended scan code into a variable called `Ch` and sets a Boolean variable called `FuncKey` to `True` if the character is a special key:
ReadKey function

```pascal
Ch := ReadKey;
if Ch <> #0 then funcKey := False else
begin
    funcKey := True;
    Ch := ReadKey;
end;
```

The `CheckBreak` variable controls whether `Ctrl+Break` should abort the program or be returned like any other key. When `CheckBreak` is False, `ReadKey` returns a `Ctrl+C` (#3) for `Ctrl+Break`.

See also KeyPressed

Readln procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Executes the <code>Read</code> procedure then skips to the next line of the file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>procedure Readln( [ var F: Text; ] V_1[, V_2, ..., V_N ] );</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>Readln</code> is an extension to <code>Read</code>, as it is defined on text files. After executing the <code>Read</code>, <code>Readln</code> skips to the beginning of the next line of the file.</td>
</tr>
</tbody>
</table>

- `Readln(F)` with no parameters causes the current file position to advance to the beginning of the next line if there is one; otherwise, it goes to the end of the file. `Readln` with no parameter list corresponds to `Readln(Input)`.  
- With `$I-$`, `IOResult` returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

| Restrictions | Works only on text files. File must be open for input. |
| See also | `Read` |

Rectangle procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Draws a rectangle using the current line style and color.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>procedure Rectangle(X1, Y1, X2, Y2: Integer);</code></td>
</tr>
</tbody>
</table>
| Remarks | `(X1, Y1)` define the upper left corner of the rectangle, and `(X2, Y2)` define the lower right corner (0 <= X1 < X2 <= GetMaxX, and 0 <= Y1 < Y2 <= GetMaxY).  

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 XORed to the screen. |
Restrictions
Must be in graphics mode.

See also
Bar, Bar3D, GetViewSettings, InitGraph, SetColor, SetLineStyle, SetViewPort, SetWriteMode

Example
uses Crt, Graph;
var
  GraphDriver, GraphMode: Integer;
  X1, Y1, X2, Y2: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, ' ');
  if GraphResult<> grOk then
    Halt(1);
  Randomize;
  repeat
    X1 := Random(GetMaxX);
    Y1 := Random(GetMaxY);
    X2 := Random(GetMaxX - X1) + X1;
    Y2 := Random(GetMaxY - Y1) + Y1;
    Rectangle(X1, Y1, X2, Y2);
  until KeyPressed;
  CloseGraph;
end.

RegisterBGIdriver function

Registers a user-loaded or linked-in BGI driver with the graphics system.

Declaration
function RegisterBGIdriver(Driver: Pointer): Integer;

Remarks
If an error occurs, the return value is less than 0; otherwise, the internal driver number is returned.

This routine enables a user to load a driver file and "register" the driver by passing its memory location to RegisterBGIdriver. When that driver is used by InitGraph, the registered driver will be used (instead of being loaded from disk by the Graph unit). A user-registered driver can be loaded from disk onto the heap, or converted to an .OBJ file (using BINOBJ.EXE) and linked into the .EXE.

Returns grInvalidDriver if the driver header is not recognized.

The following program loads the CGA driver onto the heap, registers it with the graphics system, and calls InitGraph:
program LoadDriv;
uses Graph;
var
  Driver, Mode: Integer;
  DriverF: file;
  DriverP: Pointer;
begin
  { Open driver file, read into memory, register it }
  Assign(DriverF, 'CGA.BGI');
  Reset(DriverF, 1);
  GetMem(DriverP, FileSize(DriverF));
  BlockRead(DriverF, DriverP^, FileSize(DriverF));
  if RegisterBGIdriver(DriverP) < 0 then
    begin
      Writeln('Error registering driver: ',
        GraphErrorMsg(GraphResult));
      Halt(1);
    end;
  { Init graphics }
  Driver := CGA;
  Mode := CGAHi;
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  OutText('Driver loaded by user program');
  Readln;
  CloseGraph;
end.

The program begins by loading the CGA driver file from disk and registering it with the Graph unit. Then a call is made to InitGraph to initialize the graphics system. You might wish to incorporate one or more driver files directly into your .EXE file. In this way, the graphics drivers that your program needs will be built-in and only the .EXE will be needed in order to run. The process for incorporating a driver file into your .EXE is straightforward:

1. Run BINOBJ on the driver file(s).
2. Link the resulting .OBJ file(s) into your program.
3. Register the linked-in driver file(s) before calling InitGraph.

For a detailed explanation and example of the preceding, see the comments at the top of the BGILINK.PAS example program on the distribution disks. For information on the BINOBJ utility, see the file UTILS.DOC (in ONLINE.ZIP) on your distribution disks.
It is also possible to register font files; see the description of `RegisterBGIfont`.

**Restrictions**
Note that the driver must be registered before the call to `InitGraph`. If a call is made to `RegisterBGIdriver` once graphics have been activated, a value of `grError` will be returned. If you want to register a user-provided driver, you must first call `InstallUserDriver`, then proceed as described in the previous example.

**See also** `InitGraph`, `InstallUserDriver`, `RegisterBGIfont`

---

**RegisterBGIfont**

**Purpose**
Registers a user-loaded or linked-in BGI font with the graphics system.

**Declaration**
```pascal
function RegisterBGIfont(Font: Pointer): Integer;
```

**Remarks**
The return value is less than 0 if an error occurs. Possible error codes are `grError`, `grInvalidFont`, and `grInvalidFontNum`. If no error occurs, the internal font number is returned. This routine enables a user to load a font file and "register" the font by passing its memory location to `RegisterBGIfont`. When that font is selected with a call to `SetTextStyle`, the registered font will be used (instead of being loaded from disk by the `Graph` unit). A user-registered font can be loaded from disk onto the heap, or converted to an `.OBJ` file (using BINOBJ.EXE) and linked into the `.EXE`.

There are several reasons to load and register font files. First, `Graph` only keeps one stroked font in memory at a time. If you have a program that needs to quickly alternate between stroked fonts, you might want to load and register the fonts yourself at the beginning of your program. Then `Graph` will not load and unload the fonts each time a call to `SetTextStyle` is made.

Second, you might wish to incorporate the font files directly into your `.EXE` file. This way, the font files that your program needs will be built-in, and only the `.EXE` and driver files will be needed in order to run. The process for incorporating a font file into your `.EXE` is straightforward:

1. Run BINOBJ on the font file(s).
2. Link the resulting `.OBJ` file(s) into your program.
3. Register the linked-in font file(s) before calling `InitGraph`.

For a detailed explanation and example of the preceding, see the comments at the top of the BGILINK.PAS example program on the...
distribution disks. Documentation on the BINOBJ utility is available in the file UTILS.DOC (in ONLINE.ZIP) on your distribution disks.

Note that the default (8x8 bit-mapped) font is built into GRAPH.TPU, and thus is always in memory. Once a stroked font has been loaded, your program can alternate between the default font and the stroked font without having to reload either one of them.

It is also possible to register driver files; see the description of RegisterBGldriver.

The following program loads the triplex font onto the heap, registers it with the graphics system, and then alternates between using triplex and another stroked font that Graph loads from disk (SansSerifFont):

```pascal
program LoadFont;
uses Graph;
var
  Driver, Mode: Integer;
  FontF: file;
  FontP: Pointer;
begin
  { Open font file, read into memory, register it }
  Assign(FontF, 'TRIP.CHR');
  Reset(FontF, 1);
  GetMem(FontP, FileSize(FontF));
  BlockRead(FontF, FontP^, FileSize(FontF));
  if RegisterBGIfont(FontP) < 0 then
    begin
      Writeln('Error registering font: ', GraphErrorMsg(GraphResult));
      Halt(1);
    end;
  { Init graphics }
  Driver := Detect;
  InitGraph(Driver, Mode, '..\');
  if GraphResult < 0 then
    Halt(1);
  Readln;
  { Select registered font }
  SetTextStyle(TriplexFont, HorizDir, 4);
  OutText('Triplex loaded by user program');
  MoveTo(0, TextHeight('a'));
  Readln;
  { Select font that must be loaded from disk }
  SetTextStyle(SansSerifFont, HorizDir, 4);
  OutText('Your disk should be spinning...');
  MoveTo(0, GetY + TextHeight('a'));
  Readln;
```

Programmer's Reference
The program begins by loading the triplex font file from disk and registering it with the `Graph` unit. Then a call to `InitGraph` is made to initialize the graphics system. Watch the disk drive indicator and press `Enter`. Because the triplex font is already loaded into memory and registered, `Graph` does not have to load it from disk (and therefore your disk drive should not spin). Next, the program will activate the sans serif font by loading it from disk (it is unregistered). Press `Enter` again and watch the drive spin. Finally, the triplex font is selected again. Since it is in memory and already registered, the drive will not spin when you press `Enter`.

See also `InitGraph`, `InstallUserDriver`, `InstallUserFont`, `RegisterBGIFont`, `SetTextStyle`

### Registers type

#### Purpose

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

#### Declaration

```pascal
type
    Registers = record
        case Integer of
            0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
            1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
        end;
end;
```

Notice the use of a variant record to map the 8-bit registers on top of their 16-bit equivalents.

See also `Intr`, `MsDos`

### RemoveDir procedure

#### Purpose

Removes an empty subdirectory.

#### Declaration

```pascal
procedure RemoveDir(Dir: PChar);
```

#### Remarks

The subdirectory with the path specified by `Dir` is removed. Errors, such as a non-existing or non-empty subdirectory, are reported in the `DosError` variable.
RemoveDir procedure

See also GetCurDir, CreateDir, SetCurDir. RmDir removes an empty subdirectory also, but it takes a Pascal-style string as the argument rather than a null-terminated string.

Rename procedure

Purpose Renames an external file.

Declaration

procedure Rename(var F: file; Newname);

Remarks

F is a variable of any file type. Newname is a string-type expression or an expression of type PChar if the extended syntax is enabled. The external file associated with F is renamed to Newname. Further operations on F operate on the external file with the new name.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Never use Rename on an open file.

See also Erase

Reset procedure

Purpose Opens an existing file.

Declaration

procedure Reset(var F: file; RecSize: Word);

Remarks

F is a variable of any file type associated with an external file using Assign. RecSize is an optional expression of type Word, which can be specified only if F is an untyped file. If F is an untyped file, RecSize specifies the record size to be used in data transfers. If RecSize is omitted, a default record size of 128 bytes is assumed.

Reset opens the existing external file with the name assigned to F. An error results if no existing external file of the given name exists. If F is already open, it is first closed and then reopened. The current file position is set to the beginning of the file.

If F is assigned an empty name, such as Assign(F, ''), then after the call to Reset, F refers to the standard input file (standard handle number 0).

If F is a text file, F becomes read-only. After a call to Reset, Eof(F) is True if the file is empty; otherwise, Eof(F) is False.
Reset procedure

With {$I-$}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
Append, Assign, Close, Rewrite, Truncate

Example

function FileExists(FileName: String): Boolean;
{ Boolean function that returns True if the file exists; otherwise, it returns False. Closes the file if it exists. }
var F: file;
begin
{$I-$}
Assign(F, FileName);
FileMode := 0;  { Set file access to read only. }
Reset(F);
Close(F);
{$I+}
FileExists := (IOResult = 0) and (FileName <> '');
end;  { FileExists }

begin
if FileExists(ParamStr(1)) then  { Get file name from command line }
  Writeln('File exists')
else
  Writeln('File not found');
end.

RestoreCrtMode procedure

Graph

Purpose
Restores the screen mode to its original state before graphics mode was initialized.

Declaration
procedure RestoreCrtMode;

Remarks
Restores the original video mode detected by $InitGraph$. Can be used in conjunction with $SetGraphMode$ to switch back and forth between text and graphics modes.

Restrictions
Must be in graphics mode.

See also
CloseGraph, DetectGraph, $GetGraphMode$, $InitGraph$, $SetGraphMode$

Example
uses Graph;
var
  Gd, Gm: Integer;
  Mode: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
RestoreCrtMode procedure

if GraphResult <> grOk then
  Halt(1);
  OutText(' <ENTER> to leave graphics: ');
  Readln;
  RestoreCrtMode;
  Writeln('Now in text mode');
  Write(' <ENTER> to enter graphics mode: ');
  Readln;
  SetGraphMode(GetGraphMode);
  OutTextXY(O, 0, 'Back in graphics mode');
  OutTextXY(O, TextHeight('H'), ' <ENTER> to quit: ');
  Readln;
  CloseGraph;
end.

Rewrite procedure

Purpose
Creates and opens a new file.

Declaration
procedure Rewrite(var F : file; RecSize: Word );

Remarks
F is a variable of any file type associated with an external file using Assign.
RecSize is an optional expression of type Word, which can only be specified
if F is an untyped file. If F is an untyped file, RecSize specifies the record
size to be used in data transfers. If RecSize is omitted, a default record size
of 128 bytes is assumed.

Rewrite creates a new external file with the name assigned to F. If an
external file with the same name already exists, it is deleted and a new
empty file is created in its place. If F is already open, it is first closed and
then re-created. The current file position is set to the beginning of the
empty file.

If F was assigned an empty name, such as Assign(F, ''), then after the call
to Rewrite, F refers to the standard output file (standard handle number 1).

If F is a text file, F becomes write-only. After a call to Rewrite, Eof(F) is
always True.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it
returns a nonzero error code.

See also Append, Assign, FileMode, Lst, Reset, Truncate
Example

```pascal
var F: Text;
begin
  Assign(F, 'NEWFILE.$$');
  Rewrite(F);
  WriteLn(F, 'Just created file with this text in it...');
  Close(F);
end.
```

RmDir procedure

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Removes an empty subdirectory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>procedure RmDir(S: String);</td>
</tr>
<tr>
<td>Remarks</td>
<td>Removes the subdirectory with the path specified by S. If the path does not exist, is non-empty, or is the currently logged directory, an I/O error occurs.</td>
</tr>
<tr>
<td>Remarks</td>
<td>With ${-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.</td>
</tr>
<tr>
<td>See also</td>
<td>MkDir, ChDir, GetDir. RemoveDir performs the same function as RmDir, but it takes a null-terminated string as an argument rather than a Pascal-style string.</td>
</tr>
</tbody>
</table>
| Example          | begin
  {$I-}
  RmDir(ParamStr(1));    { Get directory name from command line }
  if IOResult <> 0 then
    WriteLn('Cannot remove directory')
  else
    WriteLn('Directory removed');
end. |

Round function

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Rounds a real-type value to an integer-type value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>function Round(X: Real): Longint;</td>
</tr>
<tr>
<td>Remarks</td>
<td>X is a real-type expression. Round returns a Longint value that is the value of X rounded to the nearest whole number. If X is exactly halfway between two whole numbers, the result is the number with the</td>
</tr>
</tbody>
</table>
Round function

The round function determines the closest integer to its argument, with ties being rounded towards positive infinity. A run-time error occurs if the rounded value of X is not within the Longint range.

See also: Int, Trunc

RunError procedure

**Purpose**: Stops program execution and generates a run-time error.

**Declaration**:
```pascal
procedure RunError ( ErrorCode: Byte );
```

**Remarks**: The RunError procedure corresponds to the Halt procedure, except in addition to stopping the program, it generates a run-time error at the current statement. ErrorCode is the run-time error number (0 if omitted). If the current module is compiled with debug information on, and you're running the program from the IDE, Turbo Pascal automatically takes you to the RunError call, just as if an ordinary run-time error occurred.

See also: Exit, Halt

**Example**:
```pascal
{$IFDEF Debug}
if P = nil then
  RunError(204);
{$ENDIF}
```

SaveIntXX variables

**Purpose**: Stores interrupt vectors.

**Declaration**: The System unit declares the following SaveIntXX variables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SaveInt00</td>
<td>Pointer</td>
<td>{ Saved interrupt $00 }</td>
</tr>
<tr>
<td>SaveInt02</td>
<td>Pointer</td>
<td>{ Saved interrupt $02 }</td>
</tr>
<tr>
<td>SaveInt1B</td>
<td>Pointer</td>
<td>{ Saved interrupt $1B }</td>
</tr>
<tr>
<td>SaveInt21</td>
<td>Pointer</td>
<td>{ Saved interrupt $21 }</td>
</tr>
<tr>
<td>SaveInt23</td>
<td>Pointer</td>
<td>{ Saved interrupt $23 }</td>
</tr>
<tr>
<td>SaveInt24</td>
<td>Pointer</td>
<td>{ Saved interrupt $24 }</td>
</tr>
<tr>
<td>SaveInt34</td>
<td>Pointer</td>
<td>{ Saved interrupt $34 }</td>
</tr>
<tr>
<td>SaveInt35</td>
<td>Pointer</td>
<td>{ Saved interrupt $35 }</td>
</tr>
<tr>
<td>SaveInt36</td>
<td>Pointer</td>
<td>{ Saved interrupt $36 }</td>
</tr>
<tr>
<td>SaveInt37</td>
<td>Pointer</td>
<td>{ Saved interrupt $37 }</td>
</tr>
<tr>
<td>SaveInt38</td>
<td>Pointer</td>
<td>{ Saved interrupt $38 }</td>
</tr>
<tr>
<td>SaveInt39</td>
<td>Pointer</td>
<td>{ Saved interrupt $39 }</td>
</tr>
<tr>
<td>SaveInt3A</td>
<td>Pointer</td>
<td>{ Saved interrupt $3A }</td>
</tr>
</tbody>
</table>

Programmer's Reference
SaveIntXX variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3B</td>
<td>Pointer</td>
<td>Saved interrupt $3B</td>
</tr>
<tr>
<td>$3C</td>
<td>Pointer</td>
<td>Saved interrupt $3C</td>
</tr>
<tr>
<td>$3D</td>
<td>Pointer</td>
<td>Saved interrupt $3D</td>
</tr>
<tr>
<td>$3E</td>
<td>Pointer</td>
<td>Saved interrupt $3E</td>
</tr>
<tr>
<td>$3F</td>
<td>Pointer</td>
<td>Saved interrupt $3F</td>
</tr>
<tr>
<td>$75</td>
<td>Pointer</td>
<td>Saved interrupt $75</td>
</tr>
</tbody>
</table>

Remarks

The System unit and a number of other run-time library units take over several interrupt vectors. The run-time library initialization code in the System unit stores the old vectors in the SaveIntXX variables before installing any interrupt handling routines. Likewise, the run-time library termination code restores the interrupt vectors using the SaveIntXX variables before returning to the operating system.

If an application needs to access the “original” interrupt vector (the one that was in place before the run-time library installed a new interrupt handler), it can access the corresponding SaveIntXX variable. If there is no SaveIntXX variable for a particular interrupt vector, it is because the run-time library doesn’t modify that vector.

See also Exec, SwapVectors

SearchRec type

Purpose

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

Declaration

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

The information for each file found by one of these procedures is reported back in a SearchRec. The Attr field contains the file’s attributes (constructed from file attribute constants), Time contains its packed date and time (use UnpackTime to unpack), Size contains its size in bytes, and Name contains its name. The Fill field is reserved by DOS and should never be modified.
Sector procedure

Purpose
Draws and fills an elliptical sector.

Declaration
procedure Sector(X, Y: Integer; StAngle, EndAngle, XRadius, YRadius: Word);

Remarks
Using (X, Y) as the center point, XRadius and YRadius specify the horizontal and vertical radii, respectively; Sector draws from StAngle to EndAngle, outlined in the current color and filled with the pattern and color defined by SetFillStyle or SetFillPattern.

A start angle of 0 and an end angle of 360 will draw and fill a complete ellipse. The angles for Arc, Ellipse, FillEllipse, PieSlice, and Sector are counterclockwise with 0 degrees at 3 o’clock, 90 degrees at 12 o’clock, and so on.

If an error occurs while filling the sector, GraphResult returns a value of grNoScanMem.

Restrictions
Must be in graphics mode.

See also
Arc, Circle, Ellipse, FillEllipse, GetArcCoords, GetAspectRatio, PieSlice, SetFillStyle, SetFillPattern, SetGraphBufSize

Example
uses Graph;
const R = 50;
var
    Driver, Mode: Integer;
    Xasp, Yasp: Word;
begin
    Driver := Detect;
    InitGraph(Driver, Mode, '');
    if GraphResult < 0 then
        Halt(1);
    Sector(GetMaxX div 2, GetMaxY div 2, 0, 45, R, R);
    GetAspectRatio(Xasp, Yasp);
    Sector(GetMaxX div 2, GetMaxY div 2, 180, 135,
        R, R * Longint(Xasp) div Yasp);
    Readln;
    CloseGraph;
end.
### Seek procedure

**Purpose**
Moves the current position of a file to a specified component.

**Declaration**

```plaintext
procedure Seek(var F; N: Longint);
```

**Remarks**

- F is any file variable type except text, and N is an expression of type `Longint`. The current file position of F is moved to component number N. The number of the first component of a file is 0. To expand a file, you can seek one component beyond the last component; that is, the statement `Seek(F, FileSize(F))` moves the current file position to the end of the file.

- With `{SI-}`, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

**Restrictions**

- Cannot be used on text files. File must be open.

**See also**

- `FilePos`

---

### SeekEof function

**Purpose**
Returns the end-of-file status of a file.

**Declaration**

```plaintext
function SeekEof [ (var F: Text) ]: Boolean;
```

**Remarks**

- `SeekEof` corresponds to `Eof` except that it skips all blanks, tabs, and end-of-line markers before returning the end-of-file status. This is useful when reading numeric values from a text file.

- With `{SI-}`, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

**Restrictions**

- Can be used only on text files. File must be open.

**See also**

- `Eof`, `SeekEoln`

---

### SeekEoln function

**Purpose**
Returns the end-of-line status of a file.

**Declaration**

```plaintext
function SeekEoln [ (var F: Text) ];
```

**Remarks**

- `SeekEoln` corresponds to `Eoln` except that it skips all blanks and tabs before returning the end-of-line status. This is useful when reading numeric values from a text file.

---

*Chapter 1, Library reference* 149
SeekEoln function

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Can be used only on text files. File must be open.

See also Eoln, SeekEof

Seg function

Purpose Returns the segment of a specified object.

Declaration function Seg(X): Word;

Remarks X is any variable, or a procedure or function identifier. The result, of type Word, is the segment part of the address of X.

See also Addr, Ofs

Seg0040 variable

Purpose Selector for segment $0040.

Declaration var Seg0040: Word;

Remarks Seg0040 contains a selector that can be used to access the ROM BIOS workspace at segment address $0040. This variable is included for compatibility between DOS real and protected mode. In real mode Seg0040 always contains the value $0040, but in protected mode the actual value can vary.

See also SegA000, SegB000, SegB800

SegA000 variable

Purpose Selector for segment $A000.

Declaration var SegA000: Word;

Remarks SegA000 contains a selector that can be used to access the EGA and VGA graphics memory pages at segment address $A000. This variable is included for compatibility between DOS real and protected mode. In
real mode SegA000 always contains the value $A000, but in protected mode the actual value can vary.

See also Seg0040, SegB000, SegB800

SegB000 variable

| Purpose | Selector for segment $B000. |
| Declaration | var SegB000: Word; |
| Remarks | SegB000 contains a selector that can be used to access the Monochrome Adapter video memory at segment address $B000. This variable is included for purposes of compatibility between DOS real and protected mode. In real mode SegB000 always contains the value $B000, but in protected mode the actual value might vary. |
| See also | Seg0040, SegA000, SegB800 |

SegB800 variable

| Purpose | Selector for segment $B800. |
| Declaration | var SegB800: Word; |
| Remarks | SegB800 contains a selector that can be used to access the Color Graphics Adapter video memory at segment address $B800. This variable is included for purposes of compatibility between DOS real and protected mode. In real mode SegB800 always contains the value $B800, but in protected mode the actual value can vary. |
| See also | Seg0040, SegA000, SegB800 |

SelectorInc variable

| Purpose | Selector increment value. |
| Declaration | var SelectorInc: Word; |
| Remarks | SelectorInc contains the value that must be added to or subtracted from the selector part of a pointer to increment or decrement the pointer by 64K bytes. In real mode, SelectorInc always contains $1000, but in protected mode the actual value can vary. |
**SetActivePage procedure**

**Purpose**
Set the active page for graphics output.

**Declaration**

```pascal
procedure SetActivePage(Page: Word);
```

**Remarks**
Makes `Page` the active graphics page, directing all subsequent graphics output to `Page`.

Multiple pages are supported only by the EGA (256K), VGA, and Hercules graphics cards. With multiple graphics pages, a program can direct graphics output to an off-screen page, then quickly display the off-screen image by changing the visual page with the `SetVisualPage` procedure. This technique is especially useful for animation.

**Restrictions**
Must be in graphics mode.

**See also**
`SetVisualPage`

**Example**
```pascal
uses Graph;
var Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ');
  if GraphResult <> grOk then
    Halt(1);
  if (Gd = HercMono) or (Gd = EGA) or (Gd = EGA64) or (Gd = VGA) then
  begin
    SetVisualPage(0);
    SetActivePage(1);
    Rectangle(10, 20, 30, 40);
    SetVisualPage(1);
  end
  else
    OutText('No paging supported.');
  Readln;
  CloseGraph;
end.
```

---

**SetAllPalette procedure**

**Purpose**
Changes all palette colors as specified.

**Declaration**

```pascal
procedure SetAllPalette(var Palette);
```
Remarks  *Palette* is an untyped parameter. The first byte is the length of the palette. The next \( n \) bytes will replace the current palette colors. Each color might range from -1 to 15. A value of -1 will not change the previous entry’s value.

Note that valid colors depend on the current graphics driver and current graphics mode.

If invalid input is passed to *SetAllPalette*, *GraphResult* returns a value of -11 (*grError*), and no changes to the palette settings will occur.

Changes made to the palette are seen immediately onscreen. In the example listed here, several lines are drawn onscreen, then the palette is changed. Each time a palette color is changed, all onscreen occurrences of that color will be changed to the new color value.

See *Color constants for SetRGBPalette* for a definition of color constants and to *PaletteType* for a definition of *PaletteType* record.

Restrictions  Must be in graphics mode, and can be used only with EGA, EGA 64, or VGA (not the IBM 8514 or the VGA in 256-color mode).

See also  *GetBkColor, GetColor, GetPalette, GraphResult, SetBkColor, SetColor, GetPalette, SetRGBPalette*

Example  uses Graph;
            var  
                Gd, Gm: Integer;
                Palette: PaletteType;
            begin
                Gd := Detect;
                InitGraph(Gd, Gm, ' ');
                if GraphResult <> grOk then  
                    Halt(1);
                Line(0, 0, GetMaxX, GetMaxY);
                with Palette do  
                    begin
                        Size := 4;
                        Colors[0] := 5;
                        Colors[1] := 3;
                        Colors[2] := 1;
                        Colors[3] := 2;
                        SetAllPalette(Palette);
                    end;
                Readln;
                CloseGraph;
            end.
SetAspectRatio procedure

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Changes the default aspect-ratio correction factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>procedure SetAspectRatio(Xasp, Yasp: Word): Word;</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><code>SetAspectRatio</code> 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 aligned properly. This can be corrected in the hardware by realigning the monitor, or can be corrected in the software by changing the aspect ratio using <code>SetAspectRatio</code>. To read the current aspect ratio from the system, use <code>GetAspectRatio</code>.</td>
</tr>
<tr>
<td><strong>Restrictions</strong></td>
<td>Must be in graphics mode.</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td><code>GetAspectRatio</code></td>
</tr>
</tbody>
</table>
| **Example** | ```
uses Crt, Graph;
const R = 50;
var
  Driver, Mode: Integer;
  Xasp, Yasp: Word;
begin
  DirectVideo := False;
  Driver := Detect;
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  GetAspectRatio(Xasp, Yasp);
  if Xasp = Yasp then
    { Adjust for VGA and 8514. They have 1:1 aspect }
    Yasp := 5 * Xasp;
  while (Xasp < Yasp) and not KeyPressed do
    { Keep modifying aspect ratio until 1:1 or key is pressed }
    begin
      SetAspectRatio(Xasp, Yasp);
      Circle(GetMaxX div 2, GetMaxY div 2, R);
      Inc(Xasp, 20);
    end;
  SetTextJustify(CenterText, CenterText);
  OutTextXY(GetMaxX div 2, GetMaxY div 2, 'Done!');
  Readln;
  CloseGraph;
end.`
``` |
SetBkColor procedure

**Purpose**
Sets the current background color using the palette.

**Declaration**
```plaintext
procedure SetBkColor(ColorNum: Word);
```

**Remarks**
Background colors range from 0 to 15, depending on the current graphics driver and current graphics mode. On a CGA, `SetBkColor` sets the flood overscan color.

`SetBkColor(N)` makes the Nth color in the palette the new background color. The only exception is `SetBkColor(0)`, which always sets the background color to black.

**Restrictions**
Must be in graphics mode.

**See also**
`GetBkColor, GetColor, GetPalette, SetAllPalette, SetColor, SetPalette, SetRGBPalette`

**Example**
```plaintext
uses Crt, Graph;
var
    GraphDriver, GraphMode: Integer;
    Palette: PaletteType;
begin
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, '');
    Randomize;
    if GraphResult <> grOk then
        Halt(1);
    GetPalette(Palette);
    repeat
        if Palette.Size <> 1 then
            SetBkColor(Random(Palette.Size));
        LineTo(Random(GetMaxX), Random(GetMaxY));
        until KeyPressed;
    CloseGraph;
end.
```

SetCBreak procedure

**Purpose**
Sets the state of Ctrl+Break checking in DOS.

**Declaration**
```plaintext
procedure SetCBreak(Break: Boolean);
```
SetCBreak procedure

Remarks SetCBreak sets the state of Ctrl+Break checking in DOS. When off (False), DOS only checks for Ctrl+Break during I/O to console, printer, or communication devices. When on (True), checks are made at every system call.

See also GetCBreak

SetColor procedure

Purpose Sets the current drawing color using the palette.

Declaration procedure SetColor(Color: Word);

Remarks SetColor(S) makes the fifth color in the palette the current drawing color. Drawing colors might range from 0 to 15, depending on the current graphics driver and current graphics mode.

GetMaxColor returns the highest valid color for the current driver and mode.

Restrictions Must be in graphics mode.

See also DrawPoly, GetBkColor, GetColor, GetMaxColor, GetPalette, GraphResult, SetAllPalette, SetBkColor, SetPalette, SetRGBPalette

Example uses Crt, Graph;
var
  GraphDriver, GraphMode: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, ");
  if GraphResult <> grOk then
    Halt(1);
  Randomize;
  repeat
    SetColor(Random(GetMaxColor) + 1);
    LineTo(Random(GetMaxX), Random(GetMaxY));
    until KeyPressed;
end.

SetCurDir procedure

Purpose Changes the current directory to the path specified by Dir.

Declaration procedure SetCurDir(Dir: PChar);
SetCurDir procedure

Remarks
If Dir specifies a drive letter, the current drive is also changed. Errors are reported in DosError.

See also
GetCurDir, CreateDir, RemoveDir. ChDir performs the same function as SetCurDir, but it takes a Pascal-style string as the argument rather than a null-terminated string.

SetDate procedure

Purpose
Sets the current date in the operating system.

Declaration
procedure SetDate(Year, Month, Day: Word);

Remarks
Valid parameter ranges are Year 1980..2099, Month 1..12, and Day 1..31. If the date is invalid, the request is ignored.

See also
GetDate, GetTime, SetTime

SetFAttr procedure

Purpose
Sets the attributes of a file.

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

Remarks
F must be a file variable (typed, untyped, or text file) that has been assigned but not opened. The attribute value is formed by adding the appropriate file attribute masks defined as constants in the Dos and WinDos units. See page 43 for a list of file attribute constants.

Errors are reported in DosError; possible error codes are 3 (Invalid path) and 5 (File access denied).

Restrictions
F cannot be open.

See also
File attribute, GetFAttr, GetFTime, SetFTime

Example
uses Dos;
var F: file;
begin
  Assign(F, 'C:\AUTOEXEC.BAT');
  SetFAttr(F, Hidden);
  Readln;
  SetFAttr(F, Archive);
end.
SetFillPattern procedure

SetFillPattern procedure

Purpose
Selects a user-defined fill pattern.

Declaration
`procedure SetFillPattern(Pattern: FillPatternType; Color: Word);`

Remarks
Sets the pattern and color for all filling done by `FillPoly`, `FloodFill`, `Bar`, `Bar3D`, and `PieSlice` to the bit pattern specified in `Pattern` and the color specified by `Color`. If invalid input is passed to `SetFillPattern`, `GraphResult` returns a value of `grError`, and the current fill settings will be unchanged. The fill pattern is based on the underlying Byte values contained in the `Pattern` array. The pattern array is 8 bytes long with each byte corresponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is valued at 1, a pixel will be plotted. For example, the following pattern represents a checkerboard (50% gray scale):

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>(byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101010</td>
<td>$AA</td>
<td>(1st)</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
<td>(2nd)</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
<td>(3rd)</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
<td>(4th)</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
<td>(5th)</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
<td>(6th)</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
<td>(7th)</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
<td>(8th)</td>
</tr>
</tbody>
</table>

User-defined fill patterns enable you to create patterns different from the predefined fill patterns that can be selected with the `SetFillStyle` procedure. Whenever you select a new fill pattern with `SetFillPattern` or `SetFillStyle`, all fill operations will use that fill pattern. Calling `SetFillStyle` (`UserField, SomeColor`) will always select the user-defined pattern. This lets you define and use a new pattern using `SetFillPattern`, then switch between your pattern and the built-ins by making calls to `SetTextStyle`.

Restrictions
Must be in graphics mode.

See also
`Bar`, `Bar3D`, `FillPoly`, `GetFillPattern`, `GetFillSettings`, `GraphResult`, `grXXXX constants`, `PieSlice`

Example
`uses Graph;
const
  Gray50: FillPatternType = ($AA, $55, $AA, $55, $AA, $55, $AA, $55);
var Gd, Gm: Integer;`
begin
    Gd := Detect;
    InitGraph(Gd, Gm, '');
    if GraphResult <> grOk then
        Halt(1);
    SetFillPattern(Gray50, White);
    Bar(0, 0, 100, 100);  { Draw a bar in a 50% gray scale }
    Readln;
    CloseGraph;
end.

SetFillStyle procedure

Purpose
Sets the fill pattern and color.

Declaration
procedure SetFillStyle(Pattern: Word; Color: Word);

Remarks
Sets the pattern and color for all filling done by FillPoly, Bar, Bar3D, and PieSlice. A variety of fill patterns are available. The default pattern is solid, and the default color is the maximum color in the palette. 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 page 48 for the declaration of Fill pattern constants.

Restrictions
Must be in graphics mode.

See also
Bar, Bar3D, FillPattern, FillPoly, GetFillSettings, PieSlice, GetMaxColor, GraphResult

Example
uses Graph;
var Gm, Gd: Integer;
begin
    Gd := Detect;
    InitGraph(Gd, Gm, ' ');  
    SetFillStyle(SolidFill, 0);
    Bar(0, 0, 50, 50);
    SetFillStyle(XHatchFill, 1);
    Bar(60, 0, 110, 50);
    Readln;
    CloseGraph;
end.
### SetFTime procedure

**Purpose**
Sets the date and time a file was last written.

**Declaration**
```pascal
procedure SetFTime(var F; Time: Longint);
```

**Remarks**
- `F` must be a file variable (typed, untyped, or text file) that has been assigned and opened.
- The `Time` parameter can be created by calling `PackTime`. Errors are reported in `DosError`; the only possible error code is 6 (Invalid file handle).

**Restrictions**
- `F` must be open.

**See also**
- `DosError`, `GetFTime`, `PackTime`, `SetFAttr`, `UnpackTime`

### SetGraphBufSize procedure

**Purpose**
Lets you change the size of the buffer used for scan and flood fills.

**Declaration**
```pascal
procedure SetGraphBufSize(BufSize: Word);
```

**Remarks**
- Sets the internal buffer size to `BufSize`, and allocates a buffer on the heap when a call is made to `InitGraph`.

The default buffer size is 4K, which is large enough to fill a polygon with about 650 vertices. Under rare circumstances, you might need to enlarge the buffer in order to avoid a buffer overflow.

**Restrictions**
- Note that after `InitGraph` is called, calls to `SetGraphBufSize` are ignored.

**See also**
- `FloodFill`, `FillPoly`, `InitGraph`

### SetGraphMode procedure

**Purpose**
Sets the system to graphics mode and clears the screen.

**Declaration**
```pascal
procedure SetGraphMode(Mode: Integer);
```

**Remarks**
- `Mode` must be a valid mode for the current device driver.
- `SetGraphMode` is used to select a graphics mode different than the default one set by `InitGraph`.

`SetGraphMode` can also be used in conjunction with `RestoreCrtMode` to switch back and forth between text and graphics modes.

`SetGraphMode` resets all graphics settings to their defaults (current pointer, palette, color, viewport, and so forth).
GetModeRange returns the lowest and highest valid modes for the current
driver.

If an attempt is made to select an invalid mode for the current device
driver, GraphResult returns a value of grInvalidMode.

See page 33, Drive and Mode constants, for a list of graphics drivers and
modes.

Restrictions
A successful call to InitGraph must have been made before calling this
routine.

See also ClearDevice, CloseGraph, DetectGraph, Driver and Mode, GetGraphMode,
GetModeRange, GraphResult, InitGraph, RestoreCrtMode

Example
uses Graph;
var
  GraphDriver: Integer;
  GraphMode: Integer;
  LowMode: Integer;
  HighMode: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, ' ');
  if GraphResult <> grOk then
    Halt(1);
  GetModeRange(GraphDriver, LowMode, HighMode);
  SetGraphMode(LowMode); { Select low-resolution mode }
  Line(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.

SetIntVec procedure
Dos, WinDos

Purpose
Sets a specified interrupt vector to a specified address.

Declaration
procedure SetIntVec(IntNo: Byte; Vector: Pointer);

Remarks
IntNo specifies the interrupt vector number (0..255), and Vector specifies
the address. Vector is often constructed with the @ operator to produce the
address of an interrupt procedure. Assuming Int1BSave is a variable of
type Pointer, and Int1BHandler is an interrupt procedure identifier, the
following statement sequence installs a new interrupt $1B handler and
later restores the original handler:
SetIntVec procedure

```pascal
GetIntVec($1B, Int1BSave);
GetIntVec($1B, @Int1BHandler);
SetIntVec($1B, Int1BSave);
```

See also  `GetIntVec`

SetLineStyle procedure

### Purpose
Sets the current line width and style.

### Declaration
```pascal
procedure SetLineStyle (LineStyle: Word; Pattern: Word; Thickness: Word);
```

### Remarks
Affects all lines drawn by `Line`, `LineTo`, `Rectangle`, `DrawPoly`, `Arc`, 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 `Line style constants` 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
```

### Restrictions
Must be in graphics mode.

### See also
`DrawPoly`, `GetLineSettings`, `GraphResult`, `Line`, `LineRel`, `LineTo`, `Line style`, `SetWriteMode`

### Example
```pascal
uses Graph;
var
  Gd, Gm: Integer;
  X1, Y1, X2, Y2: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  X1 := 10;
  Y1 := 10;
```
SetLineStyle procedure

X2 := 200;
Y2 := 150;
SetLineStyle(DottedLn, 0, NormWidth);
Rectangle(X1, Y1, X2, Y2);
SetLineStyle(UserBitLn, $C3, ThickWidth);
Rectangle(Pred(X1), Pred(Y1), Succ(X2), Succ(Y2));
Readln;
CloseGraph;
end.

SetPalette procedure

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

Declaration
procedure SetPalette(ColorNum: Word; Color: Shortint);

Remarks
Changes the ColorNum entry in the palette to Color. SetPalette(0, LightCyan) makes the first color in the palette light cyan. ColorNum might range from 0 to 15, depending on the current graphics driver and current graphics mode. If invalid input is passed to SetPalette, GraphResult returns a value of grError, and the palette remains unchanged.

Changes made to the palette are seen immediately onscreen. In the example here, several lines are drawn onscreen, then the palette is changed randomly. Each time a palette color is changed, all occurrences of that color onscreen will be changed to the new color value. See Color constants for a list of defined color constants.

Restrictions
Must be in graphics mode, and can be used only with EGA, EGA 64, or VGA (not the IBM 8514).

See also
GetBkColor, GetColor, GetPalette, GraphResult, SetAllPalette, SetBkColor, SetColor, SetRGBPalette

Example
uses Crt, Graph;
var
  GraphDriver, GraphMode: Integer;
  Color: Word;
  Palette: PaletteType;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, ");
  if GraphResult <> grOk then
    Halt(1);
  GetPalette(Palette);
  if Palette.Size <> 1 then
SetPalette procedure

```
begin
  for Color := 0 to Pred(Palette.Size) do
  begin
    SetColor(Color);
    Line(0, Color * 5, 100, Color * 5);
  end;
  Randomize;
  repeat
    SetPalette(Random(Palette.Size), Random(Palette.Size));
  until KeyPressed;
end
else
  Line(0, 0, 100, 0);
  Readln;
  CloseGraph;
end.
```

SetRGBPalette procedure

Purpose

Modifies palette entries for the IBM 8514 and VGA drivers.

Declaration

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

Remarks

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

For the IBM 8514 display, ColorNum is in the range 0..255. For the VGA in 256K color mode, ColorNum is the range 0..15. Only the lower byte of RedValue, GreenValue or BlueValue is used, and out of this byte, only the 6 most-significant bits are loaded in the palette.

For compatibility with other IBM graphics adapters, the BGI driver defines the first 16 palette entries of the IBM 8514 to the default colors of the EGA/VGA. These values can be used as is, or they can be changed by using SetRGBPalette.

Restrictions

SetRGBPalette can be used only with the IBM 8514 driver and the VGA.

See also

GetBkColor, GetColor, GetPalette, GraphResult, SetAllPalette, SetBkColor, SetColor, SetPalette

Example

The first example illustrates how to use SetRGBPalette on a system using an EGA graphics driver; the second example shows how to use SetRGBPalette on a system using a VGA graphics driver.
Example 1:

```pascal
uses Graph;

type
  RGBRec = record
    RedVal, GreenVal, BlueVal: Integer;
  end;

const
  EGAColors: array[0..MaxColors] of RGBRec =
    (RedVal:$00;GreenVal:$00;BlueVal:$00), {Black EGA 0}
    (RedVal:$00;GreenVal:$00;BlueVal:$FF), {Blue EGA 1}
    (RedVal:$24;GreenVal:$FF;BlueVal:$24), {Green EGA 2}
    (RedVal:$00;GreenVal:$FF;BlueVal:$24), {Cyan EGA 3}
    (RedVal:$FF;GreenVal:$14;BlueVal:$14), {Cyan EGA 4}
    (RedVal:$00;GreenVal:$00;BlueVal:$FF), {Magenta EGA 5}
    (RedVal:$70;GreenVal:$40;BlueVal:$00), {Brown EGA 20}
    (RedVal:$C4;GreenVal:$C4;BlueVal:$C4), {White EGA 7}
    (RedVal:$34;GreenVal:$34;BlueVal:$34), {Gray EGA 62}
    (RedVal:$00;GreenVal:$00;BlueVal:$70), {Lt Blue EGA 57}
    (RedVal:$00;GreenVal:$70;BlueVal:$00), {Lt Green EGA 58}
    (RedVal:$00;GreenVal:$70;BlueVal:$70), {Lt Cyan EGA 59}
    (RedVal:$70;GreenVal:$00;BlueVal:$00), {Lt Red EGA 60}
    (RedVal:$70;GreenVal:$00;BlueVal:$FF), {Lt Magenta EGA 61}
    (RedVal:$FF;GreenVal:$FF;BlueVal:$24), {Yellow EGA 60}
    (RedVal:$FF;GreenVal:$FF;BlueVal:$FF), {Br. White EGA 63}
  );

var
  Driver, Mode, I: Integer;

begin
  Driver := IBM8514; { Override detection }
  Mode := IBM8514Hi;
  InitGraph(Driver, Mode, ''); { Put in graphics mode }
  if GraphResult < 0 then
    Halt(1); { Zero palette, make all graphics output invisible }
  for I := 0 to MaxColors do
    with EGAColors[I] do
      SetRGBPalette(I, 0, 0, 0); { Display something }
  for I := 0 to MaxColors do
    begin
      SetColor(I);
      OutTextXY(10, I * 10, '..Press any key..');
    end;
```

Chapter 7, Library reference
SetRGBPalette procedure

{ Restore default EGA colors to 8514 palette }
for I := 0 to MaxColors do
  with EGAColors[I] do
    SetRGBPalette(I, RedVal, GreenVal, BlueVal);
Readln;
CloseGraph;
e.

Example 2:

{ Example for SetRGBPalette with VGA 16 color modes }
uses Graph, CRT;
type
  RGBRec = record
    RedVal, GreenVal, BlueVal : Integer;
    { Intensity values (values from 0 to 63) }
    Name: String;
    ColorNum: Integer;
    { The VGA color palette number as mapped into 16 color palette }
  end;
const
  { Table of suggested colors for VGA 16 color modes }
Colors : array[0..MaxColors] of RGBRec = (
  ( RedVal:0;GreenVal:0;BlueVal:0;Name:'Black';ColorNum: 0),
  ( RedVal:0;GreenVal:0;BlueVal:40;Name:'Blue';ColorNum: 1),
  ( RedVal:0;GreenVal:40;BlueVal:0;Name:'Green';ColorNum: 2),
  ( RedVal:0;GreenVal:40;BlueVal:40;Name:'Cyan';ColorNum: 3),
  ( RedVal:40;GreenVal:7;BlueVal:7;Name:'Red';ColorNum: 4),
  ( RedVal:40;GreenVal:0;BlueVal:40;Name:'Magenta';ColorNum: 5),
  ( RedVal:40;GreenVal:30; BlueVal:0;Name:'Brown';ColorNum: 20),
  ( RedVal:49;GreenVal:49;BlueVal:49;Name:'Light Gray';ColorNum: 7),
  ( RedVal:26;GreenVal:26;BlueVal:26;Name:'Dark Gray';ColorNum: 56),
  ( RedVal:0;GreenVal:0;BlueVal:63;Name:'Light Blue';ColorNum: 57),
  ( RedVal:9;GreenVal:63;BlueVal:9;Name:'Light Green';ColorNum: 58),
  ( RedVal:0;GreenVal:63;BlueVal:63;Name:'Light Cyan';ColorNum: 59),
  ( RedVal:63;GreenVal:10;BlueVal:10;Name:'Light Red';ColorNum: 60),
  ( RedVal:44;GreenVal:0;BlueVal:63;Name:'Light Magenta',
    ColorNum: 61),
  ( RedVal:63;GreenVal:63;BlueVal:18;Name:'Yellow';ColorNum: 62),
  ( RedVal:63; GreenVal:63; BlueVal:63; Name: 'White'; ColorNum: 63)
);
var
  Driver, Mode, I, Error: Integer;
begin
  { Initialize Graphics Mode }
  Driver := VGA;
  Mode := VGAHI;
SetRGBPalette procedure

InitGraph(Driver, Mode, 'C:\TP\BGI');
Error := GraphResult;
if Error <> GrOk then
  begin
    writeln(GraphErrorMsg(Error));
    halt(1);
  end;
SetFillStyle(SolidFill, Green);  ( Clear )
Bar(0, 0, GetMaxX, GetMaxY);
if GraphResult < 0 then
  Hint(1);  ( Zero palette, make graphics invisible )
SetRGBPalette(Colors[0].ColorNum, 63, 63, 63);
for i := 1 to 15 do
  with Colors[i] do
    SetRGBPalette(ColorNum, 0, 0, 0);

{ Display the color name using its color with an appropriate background }
( Notice how with the current palette settings, only the text for "Press any key..., "Black", "Light Gray", and "White" are visible. This occurs because the palette entry for color 0 (Black) has been set to display as white. For the text "Light Gray" and "White," color 0 (Black) is used at the background.)
SetColor(0);
OutTextXY(0, 10, 'Press Any Key...');
for I := 0 to 15 do
  begin
    with Colors[I] do
      begin
        SetColor(I);
        SetFillStyle(SolidFill, (I xor 15) and 7);
        ( *(I xor 15)* gives an appropriate background )
        ( " and 7" reduces the intensity of the background )
        Bar(10, (I + 2) * 10 - 1, 10 + TextWidth(Name),
          (I + 2) * 10 + TextHeight(Name) - 1);
        OutTextXY(10, (I + 2) * 10, Name);
      end;
  end;
ReadKey;
( Restore original colors to the palette. The default colors might vary depending upon the initial values used by your video system.)
for i := 0 to 15 do
  with Colors[i] do
    SetRGBPalette(ColorNum, RedVal, GreenVal, BlueVal);
{ Wait for a keypress and then quit graphics and end. }
ReadKey;
Closegraph;
end.
**SetTextBuf procedure**

**Purpose**
Assigns an I/O buffer to a text file.

**Declaration**
```
procedure SetTextBuf(var F: Text; var Buf [, Size: Word ]);  
```

**Remarks**
- *F* is a text file variable, *Buf* is any variable, and *Size* is an optional expression of type *Word*.

Each text file variable has an internal 128-byte buffer that, by default, is used to buffer *Read* and *Write* operations. This buffer is adequate for most applications. However, heavily I/O-bound programs, such as applications that copy or convert text files, benefit from a larger buffer because it reduces disk head movement and file system overhead.

`SetTextBuf` changes the text file *F* to use the buffer specified by *Buf* instead of *F*’s internal buffer. *Size* specifies the size of the buffer in bytes. If *Size* is omitted, `SizeOf(Buf)` is assumed; that is, by default, the entire memory region occupied by *Buf* is used as a buffer. The new buffer remains in effect until *F* is next passed to `Assign`.

**Restrictions**
`SetTextBuf` should never be applied to an open file, although it can be called immediately after *Reset*, *Rewrite*, and *Append*. Calling `SetTextBuf` on an open file once I/O operations has taken place can cause loss of data because of the change of buffer.

Turbo Pascal doesn’t ensure that the buffer exists for the entire duration of I/O operations on the file. In particular, a common error is to install a local variable as a buffer, then use the file outside the procedure that declared the buffer.

**Example**
```
var
  F: Text;
  Ch: Char;
  Buf: array[0..4095] of Char;  
  { 4K buffer }
begin
  { Get file to read from command line }
  Assign(F, ParamStr(1));
  { Bigger buffer for faster reads }
  SetTextBuf(F, Buf);
  Reset(F);
  { Dump text file onto screen }
  while not Eof(f) do
    begin
      Read(F, Ch);
      Write(Ch);
    end;
  end.
```

SetTextJustify procedure

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

**Declaration**

```plaintext
procedure SetTextJustify(Horiz, Vert: Word);
```

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

```plaintext
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 invalid input is passed to `SetTextJustify`, `GraphResult` returns a value of `grError`, and the current text justification settings will be unchanged. See page 99 for a list of `Justification` constants.

**Restrictions**
Must be in graphics mode.

**See also**
`GetTextSettings`, `GraphResult`, `Justification`, `OutText`, `OutTextXY`, `SetLineStyle`, `SetUserCharSize`, `TextHeight`, `TextWidth`

**Example**

```plaintext
uses Graph;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, ");
if GraphResult <> grOk then
  Halt(1);
{ Center text onscreen }
SetTextJustify(CenterText, CenterText);
OutTextXY(Succ(GetMaxX) div 2, Succ(GetMaxY) div 2, 'Easily Centered');
Readln;
CloseGraph;
end.
```

SetTextStyle procedure

**Purpose**
Sets the current text font, style, and character magnification factor.

**Declaration**

```plaintext
procedure SetTextStyle(Font: Word; Direction: Word; CharSize: Word);
```

**Remarks**
Affects all text output by `OutText` and `OutTextXY`. One 8×8 bit-mapped font and several stroked fonts are available. Font directions supported are normal (left to right) and vertical (90 degrees to normal text, starts at the bottom and goes up). The size of each character can be magnified using the `CharSize` factor. A `CharSize` value of one will display the 8×8 bit-
SetTextStyle procedure

mapped font in an 8x8 pixel rectangle onscreen, a CharSize value equal to 2 will display the 8x8 bit-mapped font in a 16x16 pixel rectangle and so on (up to a limit of 10 times the normal size). Always use TextHeight and TextWidth to determine the actual dimensions of the text.

The normal size values for text are 1 for the default font and 4 for a stroked font. These are the values that should be passed as the CharSize parameter to SetTextStyle. SetUserCharSize can be used to customize the dimensions of stroked font text.

Normally, stroked fonts are loaded from disk onto the heap when a call is made to SetTextStyle. However, you can load the fonts yourself or link them directly to your .EXE file. In either case, use RegisterBGIfont to register the font with the Graph unit.

When stroked fonts are loaded from disk, errors can occur when trying to load them. If an error occurs, GraphResult returns one of the following values: grFontNotFound, grNoFontMem, grError, grIOError, grInvalidFont, or grInvalidFontNum.

Restrictions Must be in graphics mode.

See also Font control, GetTextStyle, GraphResult, OutText, OutTextXY, RegisterBGIfont, SetTextStyle, SetUserCharSize, TextHeight, TextWidth

Example uses Graph;
var
  Gd, Gm: Integer;
  Y, Size: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  Y := 0;
  for Size := 1 to 4 do
    begin
      SetTextStyle(DefaultFont, HorizDir, Size);
      OutTextXY(0, Y, 'Size = ' + Chr(Size + 48));
      Inc(Y, TextHeight('H') + 1);
    end;
  Readln;
  CloseGraph;
end.
SetTime procedure

Sets the current time in the operating system.

Declaration

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

Remarks

Valid ranges are Hour 0..23, Minute 0..59, Second 0..59, and Sec100 (hundredths of seconds) 0..99. If the time isn’t valid, the request is ignored.

See also

GetDate, GetTime, PackTime, SetDate, UnpackTime

SetUserCharSize procedure

Allows the user to vary the character width and height for stroked fonts.

Declaration

```
procedure SetUserCharSize(MultX, DivX, MultY, DivY: Word);
```

Remarks

`MultX:DivX` is the ratio multiplied by the normal width for the active font; `MultY:DivY` is the ratio multiplied by the normal height for the active font. In order to make text twice as wide, for example, use a `MultX` value of 2, and set `DivX` equal to 1 (2 div 1 = 2). Calling `SetUserCharSize` sets the current character size to the specified values.

Restrictions

Must be in graphics mode.

See also

SetTextStyle, OutText, OutTextXY, TextHeight, TextWidth

Example

The following program shows how to change the height and width of text:

```
uses Graph;
var Driver, Mode: Integer;
begins
    Driver := Detect;
    InitGraph(Driver, Mode, ");
    if GraphResult <> grOk then
        Halt(1);
    { Showoff }
    SetTextStyle(TriplexFont, HorizDir, 4);
    OutText('Norm');
    SetUserCharSize(1, 3, 1, 1);
    OutText('Short ');
    SetUserCharSize(3, 1, 1, 1);
    OutText('Wide');
    Readln;
    CloseGraph;
end.
```
SetVerify procedure

Sets the state of the verify flag in DOS.

Declaration

procedure SetVerify(Verify: Boolean);

Remarks

SetVerify sets the state of the verify flag in DOS. When off (False), disk writes are not verified. When on (True), DOS verifies all disk writes to ensure proper writing.

See also

GetVerify

SetViewPort procedure

Sets the current output viewport or window for graphics output.

Declaration

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

Remarks

(X1, Y1) define the upper left corner of the viewport, and (X2, Y2) define the lower right corner (0 <= X1 < X2 and 0 <= Y1 < Y2). 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 invalid input is passed to SetViewPort, GraphResult returns grError, and the current view settings will be unchanged.

All graphics commands (for example, GetX, OutText, Rectangle, MoveTo, and so on) are viewport-relative. In the following example, the coordinates of the dot in the middle are relative to the boundaries of the viewport.

\[ (0,0) \quad \text{to} \quad (GetMaxX,0) \]

\[ (X1,Y1) \quad \text{to} \quad (X2,Y1) \]

\[ (X1,Y2) \quad \text{to} \quad (X2,Y2) \]

\[ (0,GetMaxY) \quad \text{to} \quad (GetMaxX,GetMaxY) \]

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 complete line requested because the line will be clipped to the current viewport:

```pascal
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 was performed. But since clipping was performed, the actual line that would be drawn would start at absolute coordinates (10, 15) and terminate at coordinates (20, 15).

`InitGraph`, `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).

**Restrictions**
Must be in graphics mode.

**See also**
`ClearColorPort`, `GetViewSettings`, `GraphResult`  

**Example**
```pascal
uses Graph;
var Gd, Gm: Integer;
begin
Gd := Detect;
InitGraph(Gd, Gm, '');
if GraphResult <> grOk then
  Halt(1);
if (Gd = HercMono) or (Gd = EGA) or (Gd = EGA64) or (Gd = VGA) then
  begin
    SetVisualPage(0);
    SetActivePage(1);
    Rectangle(10, 20, 30, 40);
    SetVisualPage(1);
  end
else
  begin
    OutText('No paging supported.');
    Readln;
    CloseGraph;
  end.
```

---

**SetVisualPage procedure**

**Purpose**
Sets the visual graphics page number.

**Declaration**
```pascal
procedure SetVisualPage(Page: Word);
```

---

`Chapter 1, Library reference`
SetVisualPage procedure

Remarks  Makes Page the visual graphics page.

Multiple pages are only supported by the EGA (256K), VGA, and Hercules graphics cards. With multiple graphics pages, a program can direct graphics output to an off-screen page, then quickly display the off-screen image by changing the visual page with the SetVisualPage procedure. This technique is especially useful for animation.

Restrictions  Must be in graphics mode.

See also  SetActivePage

Example  uses Graph;
var Gd, Gm: Integer;
begint
Gd := Detect;
InitGraph(Gd, Gm, '');
if GraphResult <> grOk then
  Halt(1);
if (Gd = HercMono) or (Gd = EGA) or (Gd = EGA64) or (Gd = VGA) then
  begin
    SetVisualPage(0);
    SetActivePage(1);
    Rectangle(10, 20, 30, 40);
    SetVisualPage(1);
  end
else
  OutText('No paging supported. ');
Readln;
CloseGraph;
end.

SetWriteMode procedure

Purpose  Sets the writing mode for line drawing.

Declaration  procedure SetWriteMode(WriteMode: Integer);

Remarks  See page 12 for a list of BitBlt operators used by SetWriteMode. Each 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 with the line 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 procedure

SetWriteMode affects calls only to the following routines: DrawPoly, Line, LineRel, LineTo, and Rectangle.

See also BitBlt operators, Line, LineTo, PutImage, SetLineStyle

Example

uses Crt, Graph;
var
  Driver, Mode, I: Integer;
  X1, Y1, Dx, Dy: Integer;
  FillInfo: FillSettingsType;
begin
  DirectVideo := False;
  Randomize;
  Driver := Detect;
  InitGraph(Driver, Mode, ");
  if GraphResult < 0 then
    Halt(1);
  GetFillSettings(FillInfo);
  SetFillStyle(WideDotFill, FillInfo.Color);
  Bar(O, 0, GetMaxX, GetMaxY);
  Dx := GetMaxX div 4;
  Dy := GetMaxY div 4;
  SetLineStyle(SolidLn, 0, ThickWidth);
  SetWriteMode(XORPut);
  repeat
    X1 := Random(GetMaxX - Dx);
    Y1 := Random(GetMaxY - Dy);
    Rectangle(X1, Y1, X1 + Dx, Y1 + Dy);
    Delay(10);
    Rectangle(X1, Y1, X1 + Dx, Y1 + Dy);
  until KeyPressed;
  Readln;
  CloseGraph;
end.

Sin function

Purpose Returns the sine of the argument.

Declaration function Sin(X: Real): Real;

Remarks X is a real-type expression. Returns the sine of the angle X in radians.

See also ArcTan, Cos
Example

```pascal
var R: Real;
begin
  R := Sin(Pi);
end.
```

---

**SizeOf function**

**Purpose**: Returns the number of bytes occupied by the argument.

**Declaration**: function SizeOf(X): Word;

**Remarks**: X is either a variable reference or a type identifier. SizeOf returns the number of bytes of memory occupied by X.

SizeOf should always be used when passing values to FillChar, Move, GetMem, and so on:

```pascal
FillChar(S, SizeOf(S), 0);
GetMem(P, SizeOf(RecordType));
```

**Example**

```pascal
type
  CustRec = record
    Name: string[30];
    Phone: string[14];
  end;
var P: ^CustRec;
begin
  GetMem(P, SizeOf(CustRec));
end.
```

---

**Sound procedure**

**Purpose**: Starts the internal speaker.

**Declaration**: procedure Sound(Hz: Word);

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

**See also**: NoSound

**Example**

```pascal
uses Crt;
begin
  Sound(220);
  Delay(200);
  NoSound;
end.
```
**SPtr function**

**Purpose**
Returns the current value of the SP register.

**Declaration**
`function SPtr: Word;`

**Remarks**
Returns the offset of the stack pointer within the stack segment.

**See also**
SSeg

---

**Sqr function**

**Purpose**
Returns the square of the argument.

**Declaration**
`function Sqr(X);`

**Result type**
Same type as parameter.

**Remarks**
X is an integer-type or real-type expression. The result, of the same type as X, is the square of X, or X*X.

---

**Sqrt function**

**Purpose**
Returns the square root of the argument.

**Declaration**
`function Sqrt(X: Real): Real;`

**Remarks**
X is a real-type expression. The result is the square root of X.

---

**SSeg function**

**Purpose**
Returns the current value of the SS register.

**Declaration**
`function SSeg: Word;`

**Remarks**
The result, of type `Word`, is the segment address of the stack segment.

**See also**
`SPtr, CSeg, DSeg`

---

**StackLimit variable**

**Purpose**
Contains the offset of the bottom of the stack in the stack segment.

**Declaration**
`var StackLimit: Word;`
### StackLimit variable

**Remarks**  
*StackLimit* returns the lowest value the SP register can contain before it is considered a stack overflow.

**See also**  
*SPtr*

### Str procedure

**Purpose**  
Converts a numeric value to its string representation.

**Declaration**  
`procedure Str(X [: Width [: Decimals ] ]; var S);`

**Remarks**  
X is an integer-type or real-type expression. *Width* and *Decimals* are integer-type expressions. *S* is a string-type variable or a zero-based character array variable if extended syntax is enabled. *Str* converts X to its string representation, according to the *Width* and *Decimals* formatting parameters. The effect is exactly the same as a call to the *Write* standard procedure with the same parameters, except that the resulting string is stored in *S* instead of being written to a text file.

**See also**  
*Val, Write*

**Example**  
```delphi
function IntToStr(I: Longint): String;
{ Convert any integer type to a string }
var S: string[11];
beg
  Str(I, S);
  IntToStr := S;
end;
beg
  Writeln(IntToStr(-5322));
end.
```

### StrCat function

**Purpose**  
Appends a copy of one string to the end of another and returns the concatenated string.

**Declaration**  
`function StrCat(Dest, Source: PChar): PChar;`

**Remarks**  
*StrCat* appends a copy of *Source* to *Dest* and returns *Dest*. *StrCat* does not perform any length checking. You must ensure that the buffer given by *Dest* has room for at least *StrLen(Dest) + StrLen(Source) + 1* characters. If you want length checking, use the *StrLCat* function.

**See also**  
*StrLCat*
**Example**

```pascal
uses Strings;

const
    Turbo: PChar = 'Turbo';
    Pascal: PChar = 'Pascal';

var
    S: array[0..15] of Char;

begin
    StrCopy(S, Turbo);
    StrCat(S, ' ');
    StrCat(S, Pascal);
    Writeln(S);
end.
```

---

**StrComp function**

**Purpose**
Compares two strings.

**Declaration**
```pascal
function StrComp(Str1, Str2: PChar): Integer;
```

**Remarks**
`StrComp` compares `Str1` to `Str2`. The return value is less than 0 if `Str1 < Str2`, 0 if `Str1 = Str2`, or greater than 0 if `Str1 > Str2`.

**See also**
`StrIComp`, `StrLComp`, `StrILIComp`

**Example**

```pascal
uses Strings;

var
    C: Integer;
    Result: PChar;
    S1, S2: array[0..79] of Char;

begin
    Readln(S1);
    Readln(S2);
    C := StrComp(S1, S2);
    if C < 0 then Result := ' is less than ' else
        if C > 0 then Result := ' is greater than ' else
            Result := ' is equal to ';
    Writeln(S1, Result, S2);
end.
```

---

**StrCopy function**

**Purpose**
Copies one string to another.

**Declaration**
```pascal
function StrCopy(Dest, Source: PChar): PChar;
```
**StrCopy function**

Remarks  StrCopy copies Source to Dest and returns Dest. StrCopy does not perform any length checking. You must ensure that the buffer given by Dest has room for at least StrLen(Source) + 1 characters. If you want length checking, use the StrLCopy function.

See also  StrECopy, StrLCopy

Example  uses Strings;
          var
          S: array[0..15] of Char;
          begin
            StrCopy(S, 'Turbo Pascal');
            Writeln(S);
          end.

**StrDispose function**

Purpose  Disposes of a string on the heap.

Declaration  function StrDispose(Str: PChar);

Remarks  StrDispose disposes of a string that was previously allocated with StrNew. If Str is nil, StrDispose does nothing.

See also  StrNew

**StrECopy function**

Purpose  Copies one string to another, returning a pointer to the end of the resulting string.

Declaration  function StrECopy(Dest, Source: PChar): PChar;

Remarks  StrECopy copies Source to Dest and returns StrEnd(Dest). You must ensure that the buffer given by Dest has room for at least StrLen(Source) + 1 characters. Nested calls to StrECopy can be used to concatenate a sequence of strings—this is illustrated by the example that follows.

See also  StrCopy, StrEnd

Example  uses Strings;
          const
          Turbo: PChar = 'Turbo';
          Pascal: PChar = 'Pascal';
          var
          S: array[0..15] of Char;
begin
    StrECopy(StrECopy(StrECopy(S, Turbo), ' '), Pascal);
    Writeln(S);
end.

StrEnd function

Purpose
Returns a pointer to the end of a string.

Declaration
function StrEnd (Str: PChar): PChar;

Remarks
StrEnd returns a pointer to the null character that terminates Str.

See also
StrLen

Example
uses Strings;
var
    S: array[0..79] of Char;
begin
    Readln(S);
    Writeln('String length is ', StrEnd(S) - S);
end.

StrIComp function

Purpose
Compares two strings without case sensitivity.

Declaration
function StrIComp(Str1, Str2: PChar): Integer;

Remarks
StrIComp compares Str1 to Str2 without case sensitivity. The return value is the same as StrComp.

See also
StrComp, StrLCat, StrLIComp

StrLCat function

Purpose
Appends characters from a string to the end of another, and returns the concatenated string.

Declaration
function StrLCat(Dest, Source: PChar; MaxLen: Word): PChar;
StrLCat function

Remarks

*StrLCat* appends at most $\text{MaxLen} - \text{StrLen(Dest)}$ characters from *Source* to the end of *Dest*, and returns *Dest*. The *SizeOf* standard function can be used to determine the *MaxLen* parameter.

See also

*StrCat*

Example

```pascal
uses Strings;
var
  S: array[0..9] of Char;
begin
  StrLCopy(S, 'Turbo', SizeOf(S) - 1)
  StrLCat(S, ' ', SizeOf(S) - 1);
  StrLCat(S, 'Pascal', SizeOf(S) - 1);
  Writeln(S);
end.
```

StrLComp function

Purpose

Compares two strings, up to a maximum length.

Declaration

```pascal
function StrLComp(Str1, Str2: PChar; MaxLen: Word): Integer;
```

Remarks

*StrLComp* compares *Str1* to *Str2*, up to a maximum length of *MaxLen* characters. The return value is the same as *StrComp*.

See also

*StrComp, StrLIComp, StrIComp*

Example

```pascal
uses Strings;
var
  Result: PChar;
  S1, S2: array[0..79] of Char;
begin
  Readln(S1);
  Readln(S2);
  if StrLComp(S1, S2, 5) = 0 then
    Result := 'equal'
  else
    Result := 'different';
  Writeln('The first five characters are ', Result);
end.
```

StrLCopy function

Purpose

Copies characters from one string to another.

Declaration

```pascal
function StrLCopy(Dest, Source: PChar; MaxLen: Word): PChar;
```
StrLCopy function

Remarks

StrLCopy copies at most MaxLen characters from Source to Dest and returns Dest. The SizeOf standard function can be used to determine the MaxLen parameter—this is demonstrated by the example that follows.

See also

StrCopy

Example

uses Strings;
var
  S: array[0..9] of Char;
begin
  StrLCopy(S, 'Turbo Pascal', SizeOf(S) - 1);
  Writeln(S);
end.

StrLen function

Purpose

Returns the number of characters in Str.

Declaration

function StrLen(Str: PChar): Word;

Remarks

StrLen returns the number of characters in Str, not counting the null terminator.

See also

StrEnd

Example

uses Strings;
var
  S: array[0..79] of Char;
begin
  Readln(S);
  Writeln('String length is ', StrLen(S));
end.

StrLIComp function

Purpose

Compares two strings, up to a maximum length, without case sensitivity.

Declaration

function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;

Remarks

StrLIComp compares Str1 to Str2, up to a maximum length of MaxLen characters, without case sensitivity. The return value is the same as StrComp.

See also

StrComp, StrIComp, StrLComp
StrLower function

**Purpose**
Converts a string to lowercase.

**Declaration**
function StrLower(Str: PChar): PChar;

**Remarks**
StrLower converts Str to lowercase and returns Str.

**See also**
StrUpper

**Example**
uses Strings;
var
  S: array[0..79] of Char;
begin
  Readln(S);
  Writeln(StrLower(S));
  Writeln(StrUpper(S));
end.

StrMove function

**Purpose**
Copies characters from one string to another.

**Declaration**
function StrMove(Dest, Source: PChar; Count: Word): PChar;

**Remarks**
StrMove copies exactly Count characters from Source to Dest and returns Dest. Source and Dest can overlap.

**Example**
function StrNew(S: PChar): PChar; { Allocate string on heap }
var,
  L: Word;
  P: PChar;
begin
  if (S = nil) or (S^ = #0) then StrNew := nil else
    begin
      L := StrLen(S) + 1;
      GetMem(P, L);
      StrNew := StrMove(P, S, L);
    end;
  end;
procedure StrDispose(S: PChar); { Dispose of string on heap }
begin
  if S <> nil then FreeMem(S, StrLen(S) + 1);
end;
StrNew function

**Purpose**
Allocates a string on the heap.

**Declaration**
function StrNew(Str: PChar): PChar;

**Remarks**
StrNew allocates a copy of Str on the heap. If Str is nil or points to an empty string, StrNew returns nil and doesn’t allocate any heap space. Otherwise, StrNew makes a duplicate of Str, obtaining space with a call to the GetMem standard procedure, and returns a pointer to the duplicated string. The allocated space is StrLen(Str) + 1 bytes long.

**See also**
StrDispose

**Example**
uses Strings;
var
  P: PChar;
  S: array[0..79] of Char;
begin
  Readln(S);
  P := StrNew(S);
  Writeln(P);
  StrDispose(P);
end.

StrPas function

**Purpose**
Converts a null-terminated string to a Pascal-style string.

**Declaration**
function StrPas(Str: PChar): String;

**Remarks**
StrPas converts Str to a Pascal-style string.

**See also**
StrPCopy

**Example**
uses Strings;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  Readln(A);
  S := StrPas(A);
  Writeln(S);
end.
**StrPCopy function**

**Purpose**
Copies a Pascal-style string into a null-terminated string.

**Declaration**
function StrPCopy(Dest: PChari Source: String): PChar;

**Remarks**
`StrPCopy` copies the Pascal-style string `Source` into `Dest` and returns `Dest`. You must ensure that the buffer given by `Dest` has room for at least `Length(Source) + 1` characters.

**See also**
`StrCopy`

**Example**
uses Strings;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  Readln(S);
  StrPCopy(A, S);
  Writeln(A);
end.

**StrPos function**

**Purpose**
Returns a pointer to the first occurrence of a string in another string.

**Declaration**
function StrPos(St1, St2: PChar): PChar;

**Remarks**
`StrPos` returns a pointer to the first occurrence of `St2` in `St1`. If `St2` does not occur in `St1`, `StrPos` returns `nil`.

**Example**
uses Strings;
var
  P: PChar;
  S, SubStr: array[0..79] of Char;
begin
  Readln(S);
  Readln(SubStr);
  P := StrPos(S, SubStr);
  if P = nil then
    Writeln('Substring not found');
  else
    Writeln('Substring found at index ', P - S);
end.
### StrRScan function

**Purpose**
Returns a pointer to the last occurrence of a character in a string.

**Declaration**
```pascal
function StrRScan(Str: PChar; Chr: Char): PChar;
```

**Remarks**
`StrRScan` returns a pointer to the last occurrence of `Chr` in `Str`. If `Chr` does not occur in `Str`, `StrRScan` returns `nil`. The null terminator is considered to be part of the string.

**See also**
`StrScan`

**Example**
```pascal
{ Return pointer to name part of a full path name }
function NamePart(FileName: PChar): PChar;
var
  P: PChar;
begin
  P := StrRScan(FileName, '\');
  if P = nil then
    begin
      P := StrRScan(FileName, ':');
      if P = nil then P := FileName;
    end;
  NamePart := P;
end;
```

---

### StrScan function

**Purpose**
Returns a pointer to the first occurrence of a character in a string.

**Declaration**
```pascal
function StrScan(Str: PChar; Chr: Char): PChar;
```

**Remarks**
`StrScan` returns a pointer to the first occurrence of `Chr` in `Str`. If `Chr` does not occur in `Str`, `StrScan` returns `nil`. The null terminator is considered to be part of the string.

**See also**
`StrRScan`

**Example**
```pascal
{ Return True if file name has wildcards in it }
function HasWildcards(FileName: PChar): Boolean;
begin
  HasWildcards := (StrScan(FileName, '*') <> nil) or
                  (StrScan(FileName, '?') <> nil);
end;
```
StrUpper function

**Purpose**
Converts a string to uppercase.

**Declaration**
```pascal
function StrUpper(Str: PChar): PChar;
```

**Remarks**
`StrUpper` converts `Str` to uppercase and returns `Str`.

**See also**
`StrLower`.

**Example**
```pascal
uses Strings;
var
  S: array[0..79] of Char;
begin
  Readln(S);
  Writeln(StrUpper(S));
  Writeln(StrLower(S));
end.
```

Succ function

**Purpose**
Returns the successor of the argument.

**Declaration**
```pascal
function Succ(X): Result type
```

**Remarks**
`X` is an ordinal-type expression. The result, of the same type as `X`, is the successor of `X`.

**See also**
`Dec, Inc, Pred`

Swap function

**Purpose**
Swaps the high- and low-order bytes of the argument.

**Declaration**
```pascal
function Swap(X);
```

**Remarks**
`X` is an expression of type `Integer` or `Word`.

**See also**
`Hi, Lo`
SwapVectors procedure

**Purpose**
Swaps interrupt vectors.

**Declaration**
```
procedure SwapVectors;
```

**Remarks**
Swaps the contents of the SaveIntXX pointers in the System unit with the current contents of the interrupt vectors. SwapVectors is typically called just before and just after a call to Exec. This ensures that the Execed process does not use any interrupt handlers installed by the current process and vice versa.

**See also**
Exec, SaveIntXX

**Example**
```
{$M 8192,0,0}
uses Dos;
var Command: string[79];
begin
Write('Enter DOS command: '); Readln(Command);
if Command <> '' then
    Command:= '/C ' + Command;
SwapVectors;
Exec(GetEnv('COMSPEC'), Command);
SwapVectors;
if DosError <> 0 then
    Writeln('Could not execute COMMAND.COM');
end.
```

TDateTime type

**Purpose**
Variables of type TDateTime are used in connection with UnpackTime and PackTime procedures to examine and construct 4-byte, packed date-and-time values for the GetFTime, SetFTime, FindFirst, and FinNext procedures.

**Declaration**
```
type
    TDateTime = record
        Year, Month, Day, Hour, Min, Sec: Word;
    end;
```
TDate&Time type

Remarks
Valid ranges are Year 1980..2099, Month 1..12, Day 1..31, Hour 0..23, Min 0..59, and Sec 0..59.

See also PackTime

Test8086 variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Identifies the type of 80x86 processor the system contains.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var Test8086: Byte;</td>
</tr>
<tr>
<td>Remarks</td>
<td>The run-time library's start-up code contains detection logic that automatically determines what kind of 80x86 processor the system contains. The result of the CPU detection is stored in Test8086 as one of the following values:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Processor is an 8086</td>
</tr>
<tr>
<td>1</td>
<td>Processor is an 80286</td>
</tr>
<tr>
<td>2</td>
<td>Processor is an 80386 or later</td>
</tr>
</tbody>
</table>

When the run-time library detects that the processor is an 80386 or later CPU, it uses 80386 instructions to speed up certain operations. In particular, Longint multiplication, division, and shifts are performed using 32-bit instructions when an 80386 is detected.

See also Test8087

Test8087 variable

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stores the results of the 80x87 autodetection logic and coprocessor classification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>var Test8087: Byte;</td>
</tr>
<tr>
<td>Remarks</td>
<td>The Test8087 variable indicates whether floating-point instructions are being emulated or actually executed. The following values stored in Test8087 are defined.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No coprocessor detected</td>
</tr>
<tr>
<td>1</td>
<td>8087 detected</td>
</tr>
<tr>
<td>2</td>
<td>80287 detected</td>
</tr>
<tr>
<td>3</td>
<td>80387 or later detected</td>
</tr>
</tbody>
</table>
If an application contains no 80x87 instructions, the 80x87 detection logic is not linked into the executable, and Test8087 will therefore always contain zero.

For additional information on writing programs using the 80x87, see Chapter 14, “Using the 80x87,” in the Language Guide.

Example

The following program tests for the existence of a coprocessor.

```pascal
program Test87;
{$N+}  { Enable 80x87 instructions }
{$E+}  { Include 80x87 emulator library }
var
  X: Single;
begin
  X := 0;  { Force generation of 80x87 instructions }
  case Test8087 of
    0: Writeln ('No numeric coprocessor detected.');
    1: Writeln ('8087 detected.');
    2: Writeln ('80287 detected.');
    3: Writeln ('80387 or later detected.');
  end;
end.
```

**TextAttr variable**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stores the currently selected text attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>var TextAttr: Byte;</code></td>
</tr>
<tr>
<td>Remarks</td>
<td>Although text attributes are normally set through calls to <code>TextColor</code> and <code>TextBackground</code>, you can also set them by directly storing a value in <code>TextAttr</code>. The color information is encoded in <code>TextAttr</code> as follows:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bit</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

where `ffff` is the 4-bit foreground color, `bbbb` is the 3-bit background color, and `B` is the blink-enable bit. If you use the color constants for creating `TextAttr` values, the background color can only be selected from the first 8 colors, and it must be multiplied by 16 to get it into the correct bit positions. For example, the following assignment selects blinking yellow characters on a blue background:

```
TextAttr := Yellow + Blue * 16 + Blink;
```

See also `LowVideo, NormVideo, TextBackground, TextColor`
Text color constants

Purpose
Represents the text colors.

Remarks
The following constants are used in connection with the TextColor and TextBackground procedures.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>2</td>
</tr>
<tr>
<td>Cyan</td>
<td>3</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Magenta</td>
<td>5</td>
</tr>
<tr>
<td>Brown</td>
<td>6</td>
</tr>
<tr>
<td>LightGray</td>
<td>7</td>
</tr>
<tr>
<td>DarkGray</td>
<td>8</td>
</tr>
<tr>
<td>LightBlue</td>
<td>9</td>
</tr>
<tr>
<td>LightGreen</td>
<td>10</td>
</tr>
<tr>
<td>LightCyan</td>
<td>11</td>
</tr>
<tr>
<td>LightRed</td>
<td>12</td>
</tr>
<tr>
<td>LightMagenta</td>
<td>13</td>
</tr>
<tr>
<td>Yellow</td>
<td>14</td>
</tr>
<tr>
<td>White</td>
<td>15</td>
</tr>
<tr>
<td>Blink</td>
<td>128</td>
</tr>
</tbody>
</table>

Text colors are represented by the numbers between 0 and 15; to easily identify each color, you can use these constants instead of numbers. In the color text modes, the foreground of each character is selectable from 16 colors, and the background from 8 colors. The foreground of each character can also be made to blink.

See also TextAttr, TextBackground, TextColor

TextBackground procedure

Purpose
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. There is a byte variable in Crt—TextAttr—that is used to hold the current video attribute. TextBackground sets bits 4–6 of TextAttr to Color.
TextBackground procedure

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

See also  HighVideo, LowVideo, NormVideo, TextColor, Text color

TextColor procedure

Purpose  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.

There is a byte-type variable in Crt—TextAttr—that is used to hold the current video attribute. 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.

You can make characters blink by adding 128 to the color value. The Blink constant is defined for that purpose; in fact, for compatibility with Turbo Pascal 3.0, any Color value above 15 causes the characters to blink. The foreground of all characters subsequently written will be in the specified color.

See also  HighVideo, LowVideo, NormVideo, TextBackground, Text color

Example

TextColor(Green);  { Green characters }
TextColor(LightRed + Blink);  { Blinking light-red characters }
TextColor(14);  { Yellow characters }

TextHeight function

Purpose  Returns the height of a string in pixels.

Declaration  function TextHeight(TextString: String): Word;

Remarks  Takes the current font size and multiplication factor, and determines the height of 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, and more.

For example, with the 8×8 bit-mapped font and a multiplication factor of 1 (set by SetTextStyle), the string Turbo 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.
TextHeight function

**Restrictions**  Must be in graphics mode.

**See also**  *OutText, OutTextXY, SetTextStyle, SetUserCharSize, TextWidth*

**Example**
```
uses Graph;
var
    Gd, Gm: Integer;
    Y, Size: Integer;
begin
    Gd := Detect;
    InitGraph(Gd, Gm, '');
    if GraphResult <> grOk then
        Halt(1);
    Y := 0;
    for Size := 1 to 5 do
    begin
        SetTextStyle(DefaultFont, HorizDir, Size);
        OutTextXY(0, Y, 'Turbo Graphics');
        Inc(Y, TextHeight('Turbo Graphics'));
    end;
    Readln;
    CloseGraph;
end.
```

TextMode procedure

**Purpose**  Selects a specific text mode.

**Declaration**  `procedure TextMode (Mode: Word);`

**Remarks**
See page 25 for a list of defined *Crt mode constants*. 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 reselected. This is useful when you want to return to text mode after using a graphics package, such as `Graph` or `Graph3`.

The following call to `TextMode`:
```
    TextMode(C80 + Font8x8)
```

will reset the display into 43 lines and 80 columns on an EGA, or 50 lines and 80 columns on a VGA with a color monitor. `TextMode(Lo(LastMode))`
always turns off 43- or 50-line mode and resets the display (although it leaves the video mode unchanged); while

\[
\text{TextMode}(\text{Lo(LastMode) + Font8x8})
\]

will keep the video mode the same, but reset the display into 43 or 50 lines.

If your system is in 43- or 50-line mode when you load a Turbo Pascal program, the mode will be preserved by the \textit{Crt} startup code, and the window variable that keeps track of the maximum number of lines onscreen (\textit{WindMax}) will be initialized correctly.

Here's how to write a "well-behaved" program that will restore the video mode to its original state:

```pascal
program Video;
uses Crt;
var OrigMode: Integer;
begin
  OrigMode := LastMode;  \{ Remember original mode \}
  TextMode(OrigMode);
end.
```

Note that \textit{TextMode} does not support graphics modes, and therefore \textit{TextMode(OrigMode)} will only restore those modes supported by \textit{TextMode}.

\textbf{See also} \textit{Crt mode constants, RestoreCrtMode}

\textbf{TextRec type} \hspace{1cm} \textbf{Dos}

\textbf{Purpose} Record definition used internally by Turbo Pascal and also declared in the \textit{Dos} unit.

\textbf{Declaration} \texttt{type}

\begin{verbatim}
TextBuf = array[0..127] of Char;
TextRec = record
  Handle: Word;
  Mode: Word;
  BufSize: Word;
  Private: Word;
  BufPos: Word;
  BufEnd: Word;
  BufPtr: ^TextBuf;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
end;
\end{verbatim}
TextRec type

FlushPunc: Pointer;
ClosePunc: Pointer;
UserData: array[1..16] of Byte;
Name: array[0..79] of Char;
Buffer: TextBuf;
end;

Remarks  TextRec is the internal format of a variable of type Text. See Chapter 18, “Using overlays,” in the Language Guide for additional information.

See also  FileRec

TextSettingsType type

Purpose  The record that defines the text attributes used by GetTextSettings.

Declaration  type
TextSettingsType = record
  Font: Word;
  Direction: Word;
  CharSize: Word;
  Horiz: Word;
  Vert: Word;
end;

Remarks  See page 55 for a list of the Font control control constants used to identify font attributes.

TextWidth function

Purpose  Returns the width of a string in pixels.

Declaration  function TextWidth(TextString: String): Word;

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

For example, with the 8×8 bit-mapped font and a multiplication factor of 1 (set by SetTextStyle), the string Turbo is 40 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.

Restrictions  Must be in graphics mode.
TextWidth function

See also  *OutText, OutTextXY, SetTextStyle, SetUserCharSize, TextHeight*

Example

```pascal
uses Graph;
var
  Gd, Gm: Integer;
  Row: Integer;
  Title: String;
  Size: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ');
  if GraphResult <> grOk then
    Halt(1);
  Row := 0;
  Title := 'Turbo Graphics';
  Size := 1;
  while TextWidth(Title) < GetMaxX do
    begin
      OutTextXY(0, Row, Title);
      Inc(Row, TextHeight('M'));
      Inc(Size);
      SetTextStyle(DefaultFont, HorizDir, Size);
    end;
  Readln;
  CloseGraph;
end.
```

TFileRec type

**WinDos**

**Purpose**

A record definition used for both typed and untyped files.

**Declaration**

```pascal
type TFileRec = record
  Handle: Word;
  Mode: Word;
  RecSize: Word;
  Private: array[1..26] of Byte;
  UserData: array[1..16] of Byte;
  Name: array[0..79] of Char;
end;
```

**Remarks**

*TFileRec* is a record definition used internally by Turbo Pascal as well as being declared in the *WinDos* unit. See “Internal data formats” in Chapter 19 in the *Language Guide*.
**TRegisters type**

**Purpose**
Variables of type `TRegisters` are used by Intr and MsDos procedures to specify input register contents and examine output register contents of a software interrupt.

**Declaration**

```plaintext
type TRegisters = record
  case Integer of
    0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
    1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
  end;
```

Notice the use of a variant record to map the 8-bit registers on top of their 16-bit equivalents.

**Trunc function**

**Purpose**
Truncates a real-type value to an integer-type value.

**Declaration**

```plaintext
function Trunc(X: Real): Longint;
```

**Remarks**

X is a real-type expression. `Trunc` returns a `Longint` value that is the value of X rounded toward zero.

**Restrictions**
A run-time error occurs if the truncated value of X is not within the `Longint` range.

**See also**
Round, Int

**Truncate procedure**

**Purpose**
Truncates the file size at the current file position.

**Declaration**

```plaintext
procedure Truncate(var F);
```

**Remarks**

F is a file variable of any type. All records past F are deleted, and the current file position also becomes end-of-file (`Eof(F)` is `True`).

If I/O-checking is off, the IOResult function returns a nonzero value if an error occurs.

**Restrictions**

F must be open. `Truncate` does not work on text files.

**See also**
Reset, Rewrite, Seek
TSearchRec type

Purpose
Variables of type TSearchRec are used by the FindFirst and FindNext procedures to scan directories.

Declaration
type TSearchRec = record
  Fill: array[1..21] of Byte;
  Attr: Byte;
  Time: Longint;
  Size: Longint;
  Name: array[0..12] of Char;
end;

Remarks
The information for each file found by one of these procedures is reported back in a TSearchRec. The Attr field contains the file's attributes (constructed from file attribute constants), Time contains its packed date and time (use UnpackTime to unpack), Size contains its size in bytes, and Name contains its name. The Fill field is reserved by DOS; don’t modify it.

See also
FindFirst, FindNext

TTextRec type

Purpose
A record definition that is the internal format of a variable of type Text.

Declaration
type PTextBuf = 'TTextBuf;
TTextBuf = array[0..127] of Char;
TTextRec = record
  Handle: Word;
  Mode: Word;
  BufSize: Word;
  Private: Word;
  BufPos: Word;
  BufEnd: Word;
  BufPtr: PTextBuf;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  UserData: array[1..16] of Byte;
  Name: array[0..79] of Char;
  Buffer: TTextBuf;
end;
**TTextRec type**

**Remarks**  
*TTextRec* is a record definition used internally by Turbo Pascal as well as being declared in the *WinDos* unit. See "Internal data formats" in Chapter 19 in the *Language Guide*.

---

**TypeOf function**  
*System*

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Returns a pointer to an object type's virtual method table (VMT).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function TypeOf(X): Pointer;</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td>X is either an object type identifier or an instance of an object type. In either case, <em>TypeOf</em> returns the address of the object type's virtual method table. <em>TypeOf</em> can be applied only to object types that have a VMT; all other types result in an error. See Chapter 19, “Memory issues,” in the <em>Language Guide</em>.</td>
</tr>
</tbody>
</table>

---

**UnpackTime procedure**  
*Dos, WinDos*

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Converts a 4-byte, packed date-and-time <em>Longint</em> returned by <em>GetFTime, FindFirst, or FindNext</em> into an unpacked <em>DateTime</em> record.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>procedure UnpackTime(Time: Longint; var DT: DateTime);</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><em>DateTime</em> is a record declared in the <em>Dos</em> unit. If you are writing Windows programs, use <em>TDateTime</em>. The fields of the <em>Time</em> record are not range-checked. See page 26 for the <em>DateTime</em> record declaration and page 189 for the <em>TDateTime</em> record declaration.</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td><em>DateTime, GetFTime, GetTime, PackTime, SetFTime, SetTime, TDateTime</em></td>
</tr>
</tbody>
</table>

---

**UpCase function**  
*System*

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Converts a character to uppercase.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration</strong></td>
<td><code>function UpCase(Ch: Char): Char;</code></td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><em>Ch</em> is an expression of type <em>Char</em>. The result of type <em>Char</em> is <em>Ch</em> converted to uppercase. Character values not in the range <em>a..z</em> are unaffected.</td>
</tr>
</tbody>
</table>
Val procedure

Purpose
Converts the string value to its numeric representation.

Declaration
procedure Val(S; var V; var Code: Integer);

Remarks
S is a string-type expression or an expression of type PChar if the extended syntax is enabled. V is an integer-type or real-type variable. Code is a variable of type Integer. S must be a sequence of characters that form a signed whole number according to the syntax shown in the section “Numbers” in Chapter 2 in the Language Guide. Val converts S to its numeric representation and stores the result in V. If the string is somehow invalid, the index of the offending character is stored in Code; otherwise, Code is set to zero. For a null-terminated string, the error position returned in Code is one larger than the actual zero-based index of the character in error.

Val performs range checking differently depending on the state of {$R} and the type of the parameter V.

With range checking on, {$R+}, an out-of-range value always generates a run-time error. With range checking off, {$R-}, the values for an out-of-range value vary depending upon the data type of V. If V is a Real or LongInt type, the value of V is undefined and Code returns a nonzero value. For any other numeric type, Code returns a value of zero, and V will contain the results of an overflow calculation (assuming the string value is within the long integer range).

Therefore, you should pass Val a LongInt variable and perform range checking before making an assignment of the returned value:

{SR-}
Val('65536', LongIntVar, Code)
if (Code <> 0) or (LongIntVar < 0) or (LongIntVar > 65535) then { Error }
else
WordVar := LongIntVar;

In this example, LongIntVar would be set to 65,536, and Code would equal 0. Because 65,536 is out of range for a Word variable, an error would be reported.

Restrictions
Trailing spaces must be deleted.

See also
Str
Val procedure

Example

```pascal
var I, Code: Integer;
begin
  Val(ParamStr(l), I, Code);  { Get text from command line }
  if code <> 0 then
    Writeln('Error at position: ', Code)
  else
    Writeln('Value = ', I);
end.
```

ViewPortType type

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

**Declaration**

type
ViewPortType = record
  X1, Y1, X2, Y2: Integer;
  Clip: Boolean;
end;

**Remarks**
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.

**See also**
`GetViewSettings`

WhereX function

**Purpose**
Returns the X-coordinate of the current cursor position, relative to the current window.

**Declaration**

```pascal
function WhereX: Byte;
```

**See also**
`GotoXY, WhereY, Window`

WhereY function

**Purpose**
Returns the Y-coordinate of the current cursor position, relative to the current window.

**Declaration**

```pascal
function WhereY: Byte;
```

**See also**
`GotoXY, WhereX, Window`
WindMax and WindMin variables

**Purpose**  
Store the screen coordinates of the current window.

**Declaration**  
var WindMax, WindMin: Word;

**Remarks**  
These variables are set by calls to the Window procedure. WindMin defines the upper left corner, and 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 upper left corner of the screen corresponds to \((x,y) = (0,0)\). However, for coordinates passed to Window and GotoXY, the upper left corner is at \((1,1)\).

**See also**  
GotoXY, High, Lo, LoWindow

Window procedure

**Purpose**  
Defines a text window onscreen.

**Declaration**  
procedure Window(Xl, 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 in any way invalid, 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 corner of the current window.

Many Crt procedures and functions are window-relative, including ClrEol, ClrScr, DelLine, GotoXY, InsLine, WhereX, WhereY, Read, Readln, Write, Writeln.

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

**See also**  
ClrEol, ClrScr, DelLine, GotoXY, WhereX, WhereY

Chapter 1, Library reference 203
Window procedure

**Example**

```pascal
uses Crt;
var
  X, Y: Byte;
begin
  TextBackground(Black); { Clear screen }
  ClrScr;
  repeat
    X := Succ(Random(80)); { Draw random windows }
    Y := Succ(Random(25));
    Window(X, Y, X + Random(10), Y + Random(8));
    TextBackground(Random(16)); { In random colors }
    ClrScr;
  until KeYPRESSED;
end.
```

Write procedure (text files)

**Purpose**

Writes one or more values to a text file.

**Declaration**

```pascal
procedure Write( [ var F: Text; ] P_1 [, P_2, ..., P_n ] );
```

**Remarks**

*F*, if specified, is a text file variable. If *F* is omitted, the standard file variable *Output* is assumed. Each *P* is a write parameter. Each write parameter includes an output expression whose value is to be written to the file. A write parameter can also contain the specifications of a field width and a number of decimal places. Each output expression must be of a type *Char*, *Integer*, *Real*, string, packed string, or *Boolean*.

A write parameter has the form

```pascal
OutExpr [: MinWidth [: DecPlaces ] ]
```

where *OutExpr* is an output expression. *MinWidth* and *DecPlaces* are type integer expressions.

*MinWidth* specifies the minimum field width, which must be greater than 0. Exactly *MinWidth* characters are written (using leading blanks if necessary) except when *OutExpr* has a value that must be represented in more than *MinWidth* characters. In that case, enough characters are written to represent the value of *OutExpr*. Likewise, if *MinWidth* is omitted, then the necessary number of characters are written to represent the value of *OutExpr*.

*DecPlaces* specifies the number of decimal places in a fixed-point representation of a type *Real* value. It can be specified only if *OutExpr* is of type
Real, and if MinWidth is also specified. When MinWidth is specified, it must be greater than or equal to 0.

**Write with a character-type value:** If MinWidth is omitted, the character value of OutExpr is written to the file. Otherwise, MinWidth – 1 blanks followed by the character value of OutExpr is written.

**Write with a type integer value:** If MinWidth is omitted, the decimal representation of OutExpr is written to the file with no preceding blanks. If MinWidth is specified and its value is larger than the length of the decimal string, enough blanks are written before the decimal string to make the field width MinWidth.

**Write with a type real value:** If OutExpr has a type real value, its decimal representation is written to the file. The format of the representation depends on the presence or absence of DecPlaces.

If DecPlaces is omitted (or if it is present but has a negative value), a floating-point decimal string is written. If MinWidth is also omitted, a default MinWidth of 17 is assumed; otherwise, if MinWidth is less than 8, it is assumed to be 8. The format of the floating-point string is

```
[ | ] <digit> . <decimals> E [ + | - ] <exponent>
```

The components of the output string are shown in Table 1.3:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>]</td>
</tr>
<tr>
<td>&lt;digit&gt;</td>
<td>Single digit, &quot;0&quot; only if OutExpr is 0</td>
</tr>
<tr>
<td>&lt;decimals&gt;</td>
<td>Digit string of MinWidth-7 (but at most 10) digits</td>
</tr>
<tr>
<td>E</td>
<td>Uppercase [E] character</td>
</tr>
<tr>
<td>[ +</td>
<td>- ]</td>
</tr>
<tr>
<td>&lt;exponent&gt;</td>
<td>Two-digit decimal exponent</td>
</tr>
</tbody>
</table>

If DecPlaces is present, a fixed-point decimal string is written. If DecPlaces is larger than 11, it is assumed to be 11. The format of the fixed-point string follows:

```
[ <blanks> ] [- ] <digits> [. <decimals> ]
```

The components of the fixed-point string are shown in Table 1.4:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ &lt;blanks&gt; ]</td>
<td>Blanks to satisfy MinWidth</td>
</tr>
<tr>
<td>[- ]</td>
<td>If OutExpr is negative</td>
</tr>
<tr>
<td>&lt;digits&gt;</td>
<td>At least one digit, but no leading zeros</td>
</tr>
<tr>
<td>[. &lt;decimals&gt; ]</td>
<td>Decimals if DecPlaces &gt; 0</td>
</tr>
</tbody>
</table>
Write with a string-type value: If MinWidth is omitted, the string value of OutExpr is written to the file with no leading blanks. If MinWidth is specified, and its value is larger than the length of OutExpr, enough blanks are written before the decimal string to make the field width MinWidth.

Write with a packed string-type value: If OutExpr is of packed string type, the effect is the same as writing a string whose length is the number of elements in the packed string type.

Write with a Boolean value: If OutExpr is of type Boolean, the effect is the same as writing the strings True or False, depending on the value of OutExpr.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open for output.

See also Read, Readln, Writeln

Write procedure (typed files)

Purpose Writes a variable into a file component.

Declaration procedure Write(F, V1 [, V2, ..., VN]);

Remarks F is a file variable, and each V is a variable of the same type as the component type of F. For each variable written, the current file position is advanced to the next component. If the current file position is at the end of the file (that is, if $EOF(F)$ is True) the file is expanded.

With {$I-}, IOResult returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also Writeln

Writeln procedure

Purpose Executes the Write procedure, then outputs an end-of-line marker to the file.

Declaration procedure Write(F, [ var F: Text; ] P1 [, P2, ..., PN]);

Remarks Writeln procedure is an extension to the Write procedure, as it is defined for text files. After executing Write, Writeln writes an end-of-line marker (carriage-return/linefeed) to the file. Writeln(F) with no parameters writes
Writeln procedure

an end-of-line marker to the file. (*Writeln* with no parameter list altogether corresponds to *Writeln(Output)*.)

**Restrictions**  File must be open for output.

**See also**  *Write*
This chapter describes the compiler directives you can use to control the features of the Turbo Pascal compiler. Listed alphabetically, each compiler directive is classified as either a switch, parameter, or conditional compilation directive. Following the list of compiler directives is a brief discussion of how to use the conditional compilation directives. This section describes how to use conditional constructs and symbols to produce different code from the same source text.

A compiler directive is a comment with a special syntax. Turbo Pascal allows compiler directives wherever comments are allowed. A compiler directive starts with a $ as the first character after the opening comment delimiter, immediately followed by a name (one or more letters) that designates the particular directive. You can include comments after the directive and any necessary parameters.

There are three types of directives described in this chapter:

- **Switch directives** turn particular compiler features on or off by specifying + or − immediately after the directive name. Switch directives are either global or local.
  
  • *Global directives* affect the entire compilation and must appear before the declaration part of the program or the unit being compiled.
  
  • *Local directives* affect only the part of the compilation that extends from the directive until the next occurrence of the same directive. They can appear anywhere.
You can group switch directives in a single compiler directive comment by separating them with commas with no intervening spaces. For example,

\{B+, R-, S-\}

- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.

- **Conditional directives.** These directives control conditional compilation of parts of the source text, based on user-definable conditional symbols.

All directives, except switch directives, must have at least one blank between the directive name and the parameters. Here are some examples of compiler directives:

\{B+\}
\{R- Turn off range checking\}
\{I TYPES.INC\}
\{S0 EdFormat\}
\{SM 65520, 8192, 655360\}
\{DEFINE Debug\}
\{IFDEF Debug\}
\{ENDIF\}

You can put compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (TPC.EXE) and the IDE (TURBO.EXE or TPX.EXE). The Options | Compiler menu contains many of the compiler directives; any changes you make to the settings there will affect all subsequent compilations.

When using the command-line compiler, you can specify compiler directives on the command line (for example, TPC /S+ MYPROG), or you can place directives in a configuration file (see Chapter 3). Compiler directives in the source code always override the default values in both the command-line compiler and the IDE.

If you are working in the IDE, using the editor's Alternate command set, and want a quick way to see what compiler directives are in effect, press Ctrl+O. Turbo Pascal will insert the current settings at the top of your edit window.
### Align data

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$A+} or {$A-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$A+}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Remarks</td>
<td>The {$A} directive switches between byte and word alignment of variables and typed constants. Word alignment has no effect on the 8088 CPU. However, on all 80x86 CPUs, word alignment means faster execution because word-sized items on even addresses are accessed in one memory cycle rather than two memory cycles for words on odd addresses. In the {$A+} state, all variables and typed constants larger than one byte are aligned on a machine-word boundary (an even-numbered address). If required, unused bytes are inserted between variables to achieve word alignment. The {$A+} directive does not affect byte-sized variables, nor does it affect fields of record structures and elements of arrays. A field in a record will align on a word boundary only if the total size of all fields before it is even. For every element of an array to align on a word boundary, the size of the elements must be even. In the {$A-} state, no alignment measures are taken. Variables and typed constants are simply placed at the next available address, regardless of their size. Regardless of the state of the {$A} directive, each global var and const declaration section always starts at a word boundary. Likewise, the compiler always keeps the stack pointer (SP) word aligned by allocating an extra unused byte in a procedure’s stack frame if required.</td>
</tr>
</tbody>
</table>

### Boolean evaluation

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$B+} or {$B-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$B-}</td>
</tr>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The {$B} directive switches between the two different models of code generation for the and and or Boolean operators. In the {$B+} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the and and or operators is guaranteed to be evaluated, even when the result of the entire expression is already known.</td>
</tr>
</tbody>
</table>

---

Chapter 2, Compiler directives  211
Boolean evaluation

In the {$B-} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident.

For further details, see the section “Boolean operators” in Chapter 6, “Expressions,” in the Language Guide.

### Debug information

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>{$D+} or {$D-}</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>{$D+}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Remarks</td>
<td>The $\text{D}$ directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object code addresses into source text line numbers. For units, the debug information is recorded in the .TPU file along with the unit’s object code. Debug information increases the size of .TPU files and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled in the {$O+} state, Turbo Pascal’s integrated debugger lets you single-step and set breakpoints in that module. The Standalone debugging (Options</td>
</tr>
</tbody>
</table>

If you want to use Turbo Debugger to debug your program, set Compile | Destination to Disk, choose Options | Debugger, and select the Standalone option.

### DEFINE directive

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Conditional compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{$DEFINE name}</td>
<td>Defines a conditional symbol with the given name. The symbol is recognized for the remainder of the compilation of the current module in</td>
</tr>
</tbody>
</table>
which the symbol is declared, or until it appears in an \texttt{\$UNDEF name} directive. The \texttt{\$DEFINE name} directive has no effect if \textit{name} is already defined.

\begin{center}

\textbf{ELSE directive} \\
Conditional compilation
\end{center}

\begin{tabular}{ll}
\textbf{Syntax} & \texttt{\$ELSE} \\
\textbf{Remarks} & Switches between compiling and ignoring the source text delimited by the last \texttt{\$IFxxx} and the next \texttt{\$ENDIF}.
\end{tabular}

\begin{center}

\textbf{Emulation} \\
Switch
\end{center}

\begin{tabular}{ll}
\textbf{Syntax} & \texttt{\$E+} or \texttt{\$E-} \\
\textbf{Default} & \texttt{\$E+} \\
\textbf{Type} & Global \\
\textbf{Remarks} & The \texttt{\$E} directive enables or disables linking with a run-time library that will emulate the 80x87 numeric coprocessor if one is not present. When you compile a program in the \texttt{\$N+,E+} state, Turbo Pascal links with the full 80x87 emulator. The resulting .EXE file can be used on any machine, regardless of whether an 80x87 is present. If one is found, Turbo Pascal will use it; otherwise, the run-time library emulates it. In the \texttt{\$N+,E-} state, Turbo Pascal produces a program which can only be used if an 80x87 is present. The 80x87 emulation switch has no effect if used in a unit; it applies only to the compilation of a program. Furthermore, if the program is compiled in the \texttt{\$N-} state, and if all the units used by the program were compiled with \texttt{\$N-}, then an 80x87 run-time library is not required, and the 80x87 emulation switch is ignored.
\end{tabular}

\begin{center}

\textbf{ENDIF directive} \\
Conditional compilation
\end{center}

\begin{tabular}{ll}
\textbf{Syntax} & \texttt{\$ENDIF} \\
\textbf{Remarks} & Ends the conditional compilation initiated by the last \texttt{\$IFxxx} directive.
\end{tabular}

\textit{Chapter 2, Compiler directives} 213
The {$X+} directive does not apply to built-in functions (those defined in the System unit).

---

### Force far calls

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$F+} or {$F-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$F-}</td>
</tr>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
</tbody>
</table>
| Remarks    | The {$F} directive determines which call model to use for subsequently compiled procedures and functions. Procedures and functions compiled in the {$F+} state always use the far call model. In the {$F-} state, Turbo Pascal automatically selects the appropriate model: far if the procedure or function is declared in the interface section of a unit; otherwise it selects near.

The near and far call models are described in full in Chapter 20, “Control issues,” in the Language Guide.
Generate 80286 Code

Syntax  \{G+\} or \{G-\}
Default \{G-\}
Type Global
Remarks The $G$ directive enables or disables 80286 code generation. In the \{G-\} state, only generic 8086 instructions are generated, and programs compiled in this state can run on any 80x86 family processor. You can specify \{G-\} any place within your code.
In the \{G+\} state, the compiler uses the additional instructions of the 80286 to improve code generation, but programs compiled in this state cannot run on 8088 and 8086 processors. Additional instructions used in the \{G+\} state include ENTER, LEAVE, PUSH immediate, extended IMUL, and extended SHL and SHR.

IFDEF directive

Syntax  \{IFDEF name\}
Remarks Compiles the source text that follows it if name is defined.

IFNDEF directive

Syntax  \{IFNDEF name\}
Remarks Compiles the source text that follows it if name is not defined.

IFOPT directive

Syntax  \{IFOPT switch\}
Remarks Compiles the source text that follows it if switch is currently in the specified state. switch consists of the name of a switch option, followed by a + or a - symbol. For example, the construct

\begin{verbatim}
\{IFOPT N+\}
  type Real = Extended;
\{ENDIF\}
\end{verbatim}

will compile the type declaration if the $N$ option is currently active.
Include file

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$I filename}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
</tbody>
</table>
| Remarks     | The $I$ parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {$I filename} directive. The default extension for filename is .PAS. If filename does not specify a directory path, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the Options | Directories | Include Directories input box (or in the directories specified in the /I option on the TPC command line).

There is one restriction to the use of include files: An include file can’t be specified in the middle of a statement part. In fact, all statements between the begin and end of a statement part must exist in the same source file.

Input/output checking

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$I+} or {$I-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$I+}</td>
</tr>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The $I$ switch directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in Chapter 13, “Input and output,” in the Language Guide. If an I/O procedure returns a nonzero I/O result when this switch is on, the program terminates and displays a run-time error message. When this switch is off, you must check for I/O errors by calling IOResult.</td>
</tr>
</tbody>
</table>

Link object file

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$L filename}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The $L$ parameter directive instructs the compiler to link the named file with the program or unit being compiled. The $L$ directive is used to link with code written in assembly language for subprograms declared to be external. The named file must be an Intel relocatable object file (.OBJ file).</td>
</tr>
</tbody>
</table>
The default extension for `filename` is `.OBJ`. If `filename` does not specify a directory path, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the Options | Directories | Object Directories input box (or in the directories specified in the `/O` option on the TPC command line). For further details about linking with assembly language, see Chapter 23, “Linking assembler code,” in the Language Guide.

### Local symbol information

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{$L+}</code> or <code>{$L-}</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default</th>
<th>Global</th>
</tr>
</thead>
</table>

| Remarks | The `$L` switch directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module’s implementation part, and the symbols within the module’s procedures and functions. For units, the local symbol information is recorded in the `.TPU` file along with the unit’s object code. Local symbol information increases the size of `.TPU` files, and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled in the `{$L+}` state, Turbo Pascal’s integrated debugger lets you examine and modify the module’s local variables. Furthermore, calls to the module’s procedures and functions can be examined via View | Call Stack. The Standalone debugging (Options | Debugger) and Map file (Options | Linker) options produce local symbol information for a given module only if that module was compiled in the `{$L+}` state. The `$L` switch is usually used in conjunction with the `$D` switch, which enables and disables the generation of line-number tables for debugging. The `$L` directive is ignored if the compiler is in the `{$D-}` state. |

---

Chapter 2, Compiler directives 217
### Memory allocation sizes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Syntax</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory allocation sizes</td>
<td>{M stacksize, heapmin, heapmax}</td>
<td>{M 16384, 0, 655360}</td>
</tr>
</tbody>
</table>

#### Remarks
The $M$ directive specifies an application's memory allocation parameters. *stacksize* must be an integer number in the range 1,024 to 65,520 which specifies the size of the stack segment. *heapmin* and *heapmax* specify the minimum and maximum sizes of the heap, respectively. *heapmin* must be in the range 0 to 655360, and *heapmax* must be in the range *heapmin* to 655360.

The $M$ directive has no effect when used in a unit.

### Numeric coprocessor

<table>
<thead>
<tr>
<th>Switch</th>
<th>Syntax</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric coprocessor</td>
<td>{$N+$} or {$N-$}</td>
<td>{$N-$}</td>
</tr>
</tbody>
</table>

#### Remarks
The $N$ directive switches between the two different models of floating-point code generation supported by Turbo Pascal. In the \{$N-$\} state, code is generated to perform all real-type calculations in software by calling run-time library routines. In the \{$N+$\} state, code is generated to perform all real-type calculations using the 80x87 numeric coprocessor.

### Open string parameters

<table>
<thead>
<tr>
<th>Switch</th>
<th>Syntax</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open string parameters</td>
<td>{$P+$} or {$P-$}</td>
<td>{$P-$}</td>
</tr>
</tbody>
</table>

#### Remarks
The $P$ directive controls the meaning of variable parameters declared using the *string* keyword. In the \{$P-$\} state, variable parameters declared using the *string* keyword are normal variable parameters, but in the \{$P+$\} state, they are open string parameters. Regardless of the setting of the $P$ directive, the *OpenString* identifier can always be used to declare open...
string parameters. For more information about open parameters, see Chapter 9, “Procedures and functions,” in the Language Guide.

### Overflow checking

<table>
<thead>
<tr>
<th>Switch</th>
<th>Syntax</th>
<th>Default</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{$Q+} or {$Q-}</td>
<td>{$Q-}</td>
<td>Local</td>
<td>The {$Q} directive controls the generation of overflow checking code. In the {$Q+} state, certain integer arithmetic operations (+, −, *, Abs, Sqr, Succ, and Pred) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, the program terminates and displays a run-time error message. The {$Q+} does not affect the Inc and Dec standard procedures. These procedures are never checked for overflow. The {$Q} switch is usually used in conjunction with the {$R} switch, which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger, so use {$Q+} only for debugging.</td>
</tr>
</tbody>
</table>

### Overlay code generation

<table>
<thead>
<tr>
<th>Switch</th>
<th>Syntax</th>
<th>Default</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{$O+} or {$O-}</td>
<td>{$O-}</td>
<td>Global</td>
<td>The {$O} switch directive enables or disables overlay code generation. Turbo Pascal allows a unit to be overlaid only if it was compiled with {$O+}. In this state, the code generator takes special precautions when passing string and set constant parameters from one overlaid procedure or function to another. The use of {$O+} in a unit does not force you to overlay that unit. It just instructs Turbo Pascal to ensure that the unit can be overlaid, if so desired. If you develop units that you plan to use in overlaid as well as non-overlaid applications, then compiling them with {$O+} ensures that you can indeed do both with just one version of the unit.</td>
</tr>
</tbody>
</table>

Chapter 2, Compiler directives 219
Overlay code generation

A \{$O+\} compiler directive is almost always used in conjunction with a \{$F+\} directive to satisfy the overlay manager's far call requirement.

For further details on overlay code generation, see Chapter 18, "Using overlays," in the Language Guide.

### Overlay unit name

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$O \text{unitname}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The Overlay unit name directive turns a unit into an overlay. The {$O \text{unitname}} directive has no effect if used in a unit; when compiling a program, it specifies which of the units used by the program should be placed in an .OVR file instead of in the .EXE file. {$O \text{unitname}} directives must be placed after the program's uses clause. Turbo Pascal reports an error if you attempt to overlay a unit that wasn't compiled in the {$O+} state.</td>
</tr>
</tbody>
</table>

### Range checking

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$R+} or {$R-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$R-}</td>
</tr>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The $R directive enables or disables the generation of range-checking code. In the {$R+} state, all array and string-indexing expressions are verified as being within the defined bounds and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, the program terminates and displays a run-time error message. If $R is switched on, all calls to virtual methods are checked for the initialization status of the object instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs. Enabling range checking and virtual method call checking slows down your program and makes it somewhat larger, so use the {$R+} only for debugging.</td>
</tr>
</tbody>
</table>

220  
Programmer's Reference
### Stack-overflow checking

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$S+$} or {$S-$}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$S+$}</td>
</tr>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Remarks</td>
<td>The {$S$} directive enables or disables the generation of stack-overflow checking code. In the {$S+$} state, the compiler generates code at the beginning of each procedure or function that checks whether there is sufficient stack space for the local variables and other temporary storage. When there is not enough stack space, a call to a procedure or function compiled with {$SS+$} causes the program to terminate and display a run-time error message. In the {$SS-$} state, such a call is likely to cause a system crash.</td>
</tr>
</tbody>
</table>

### Symbol reference information

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$Y+$} or {$Y-$}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$Y+$}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Remarks</td>
<td>The {$Y$} directive enables or disables generation of symbol reference information. This information consists of tables that provide the line numbers of all declarations of and references to symbols in a module. For units, the symbol reference information is recorded in the .TPU file along with the unit’s object code. Symbol reference information increases the size of .TPU files, but it does not affect the size or speed of the executable program. When a program or unit is compiled in the {$Y+$} state, Turbo Pascal’s integrated browser can display symbol definition and reference information for that module. The {$Y$} switch is usually used in conjunction with the {$D$} and {$L$} switches, which control generation of debug information and local symbol information. The {$Y$} directive has no effect unless both {$D$} and {$L$} are enabled.</td>
</tr>
</tbody>
</table>
Type-checked pointers

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>{$T+} or {$T-}</td>
<td></td>
</tr>
</tbody>
</table>

Default: {$T-}

Type: Global

Remarks: The $T directive controls the types of pointer values generated by the @ operator. In the {$T-} state, the result type of the @ operator is always Pointer. In other words, the result is an untyped pointer that is compatible with all other pointer types. When @ is applied to a variable reference in the {$T+} state, the type of the result is $^T$, where $T$ is the type of the variable reference. In other words, the result is of a type that is compatible only with other pointers to the type of the variable.

UNDEF directive

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Conditional compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{$UNDEF name}</td>
<td></td>
</tr>
</tbody>
</table>

Remarks: Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation or until it reappears in a {$DEFINE name} directive. The {$UNDEF name} directive has no effect if name is already undefined.

Var-string checking

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>{$V+} or {$V-}</td>
<td></td>
</tr>
</tbody>
</table>

Default: {$V+}

Type: Local

Remarks: The $V directive controls type checking on strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types. In the {$V-} (relaxed) state, any string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

The {$V-} state essentially provides an “unsafe” version of open string parameters. Although {$V-} is still supported, you should use open string parameters. For additional information, see “Open string parameters” in Chapter 9 in the Language Guide.
Using conditional compilation directives

Two basic conditional compilation constructs closely resemble Pascal's `if` statement. The first construct,

```
{$IFDEF xxx}
  :  
{$ENDIF}
```

causes the source text between `{IFDEF xxx}` and `{ENDIF}` to be compiled only if the condition specified in `{IFDEF xxx}` is `True`. If the condition is `False`, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFDEF xxx}
  :  
{$ELSE}
  :  
{$ENDIF}
```

causes either the source text between `{IFDEF xxx}` and `{ELSE}` or the source text between `{ELSE}` and `{ENDIF}` to be compiled, depending on the condition specified by the `{IFDEF xxx}`.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
   Real = Double;
{$ELSE}
  {$N-}
  type
   Single = Real;
   Double = Real;
   Extended = Real;
   Comp = Real;
{$ENDIF}
```

You can nest conditional compilation constructs up to 16 levels deep. For every `{IFDEF xxx}`, the corresponding `{ENDIF}` must be found within the same source file—which means there must be an equal number of `{IFDEF xxx}`'s and `{ENDIF}`'s in every source file.
Conditional symbols

Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined using the directives

\[
\begin{align*}
&\{\text{$\text{DEFINE name}$}\} \\
&\{\text{$\text{UNDEF name}$}\}
\end{align*}
\]

You can also use the /D switch in the command-line compiler (or place it in the Conditional Defines input box from within Options | Compiler of the IDE).

Conditional symbols are best compared to Boolean variables: They are either True (defined) or False (undefined). The \{\text{$\text{DEFINE}$}\} directive sets a given symbol to True, and the \{\text{$\text{UNDEF}$}\} directive sets it to False.

Conditional symbols follow the same rules as Pascal identifiers: They must start with a letter, followed by any combination of letters, digits, and underscores. They can be of any length, but only the first 63 characters are significant.

Conditional symbols and Pascal identifiers have no correlation whatsoever. Conditional symbols cannot be referenced in the actual program and the program's identifiers cannot be referenced in conditional directives. For example, the construct

\[
\text{const} \\
\quad \text{Debug} \ = \ True; \\
\text{begin} \\
\quad \{\text{$\text{IFDEF Debug}$}\} \\
\quad \text{Writeln('Debug is on');} \\
\quad \{\text{$\text{ENDIF}$}\} \\
\text{end;}
\]

will not compile the Writeln statement. Likewise, the construct

\[
\{\text{$\text{DEFINE Debug}$}\} \\
\text{begin} \\
\quad \text{if } \text{Debug} \ \text{then} \\
\quad \text{Writeln('Debug is on');} \\
\text{end;}
\]

will result in an unknown identifier error in the if statement.

Turbo Pascal defines the following standard conditional symbols:
VER70 Always defined, indicating that this is version 7.0 of Turbo Pascal. Each version has corresponding predefined symbols; for example, version 8.0 would have VER80 defined, version 8.5 would have VER85 defined, and so on.

MSDOS Always defined, indicating that the operating system is MS-DOS or PC-DOS.

CPU86 Always defined, indicating that the CPU belongs to the 80x86 family of processors. Versions of Turbo Pascal for other CPUs will instead define a symbolic name for that particular CPU.

CPU87 Defined if an 80x87 numeric coprocessor is present at compile time. If the construct

\[
\text{($IFDEF} \ CPU87 \text{($N+) ($ELSE) ($N-) ($ENDIF)}
\]

appears at the beginning of a compilation, Turbo Pascal automatically selects the appropriate model of floating-point code generation for that particular computer.

Other conditional symbols can be defined before a compilation by using the Conditional Defines input box (Options | Compiler), or the /D command-line option if you are using the command-line compiler.
Turbo Pascal command-line compiler (TPC.EXE) lets you invoke all the functions of the IDE compilers (TURBO.EXE and TPX.EXE) from the DOS command line.

You run TPC.EXE from the DOS prompt using a command with the following syntax:

```
TPC [options] filename [options]
```

*options* are zero or more optional parameters that provide additional information to the compiler. *filename* is the name of the source file to compile. If you type TPC alone, it displays a help screen of command-line options and syntax.

If *filename* does not have an extension, TPC assumes .PAS. If you don’t want the file you’re compiling to have an extension, you must append a period (.) to the end of *filename*. If the source text contained in *filename* is a program, TPC creates an executable file named FILENAME.EXE. If *filename* contains a unit, TPC creates a Turbo Pascal unit file named FILENAME.TPU.

You can specify a number of options for TPC. An option consists of a slash (/) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such as a number, a symbol, or a directory name. Options can be given in any order and can come before and/or after the file name.
Command-line compiler options

The IDE lets you set various options through the menus; TPC gives you access to these options using the slash (/) delimiter. You can also precede options with a hyphen (-) instead of a slash (/), but those options that start with a hyphen must be separated by blanks. For example, the following two command lines are equivalent and legal:

```
TPC -IC:\TP -DDEBUG SORTNAME -$S- -$F+
TPC /IC:\TP/DDEBUG SORTNAME /$S-/$F+
```

The first command line uses hyphens with at least one blank separating the options; the second uses slashes, and no separation is needed.

The following table lists the command-line options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/A+</td>
<td>Align data on word boundaries</td>
</tr>
<tr>
<td>/A-</td>
<td>Align data on byte boundaries</td>
</tr>
<tr>
<td>/B+</td>
<td>Complete Boolean evaluation</td>
</tr>
<tr>
<td>/B-</td>
<td>Short circuit Boolean evaluation</td>
</tr>
<tr>
<td>/D+</td>
<td>Debugging information on</td>
</tr>
<tr>
<td>/D-</td>
<td>Debugging information off</td>
</tr>
<tr>
<td>/E+</td>
<td>Emulation on</td>
</tr>
<tr>
<td>/E-</td>
<td>Emulation off</td>
</tr>
<tr>
<td>/F+</td>
<td>Force far calls on</td>
</tr>
<tr>
<td>/F-</td>
<td>Force far calls off</td>
</tr>
<tr>
<td>/G+</td>
<td>286 code generation on</td>
</tr>
<tr>
<td>/G-</td>
<td>286 code generation off</td>
</tr>
<tr>
<td>/I+</td>
<td>I/O checking on</td>
</tr>
<tr>
<td>/I-</td>
<td>I/O checking off</td>
</tr>
<tr>
<td>/L+</td>
<td>Local symbols on</td>
</tr>
<tr>
<td>/L-</td>
<td>Local symbols off</td>
</tr>
<tr>
<td>/Mstack,min,max</td>
<td>Memory sizes</td>
</tr>
<tr>
<td>/N+</td>
<td>Numeric coprocessor on</td>
</tr>
<tr>
<td>/N-</td>
<td>Numeric coprocessor off</td>
</tr>
<tr>
<td>/O+</td>
<td>Overlay code generation on</td>
</tr>
<tr>
<td>/O-</td>
<td>Overlay code generation off</td>
</tr>
<tr>
<td>/P+</td>
<td>Open parameters on</td>
</tr>
<tr>
<td>/P-</td>
<td>Open parameters off</td>
</tr>
<tr>
<td>/S+</td>
<td>Overflow checking on</td>
</tr>
<tr>
<td>/S-</td>
<td>Overflow checking off</td>
</tr>
<tr>
<td>/S+</td>
<td>Range checking on</td>
</tr>
<tr>
<td>/S-</td>
<td>Range checking off</td>
</tr>
<tr>
<td>/S+</td>
<td>Stack checking on</td>
</tr>
<tr>
<td>/S-</td>
<td>Stack checking off</td>
</tr>
<tr>
<td>/T+</td>
<td>Type-checked pointers on</td>
</tr>
<tr>
<td>/T-</td>
<td>Type-checked pointers off</td>
</tr>
<tr>
<td>/V+</td>
<td>Strict var-string checking</td>
</tr>
<tr>
<td>/V-</td>
<td>Relaxed var-string checking</td>
</tr>
<tr>
<td>/X+</td>
<td>Extended syntax support on</td>
</tr>
</tbody>
</table>

If you type TPC alone at the command line, a list of command-line compiler options appears on your screen.
Turbo Pascal supports several compiler directives, all of which are described in Chapter 2, "Compiler directives."

The /$ and /D command-line options let you change the default states of most compiler directives. Using /$ and /D on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The switch directive option

The /$ option lets you change the default state of all the switch directives. The syntax of a switch directive option is /$ followed by the directive letter, followed by a plus (+) or a minus (-). For example,

```
TPC MYSTUFF /$R-
```

compiles MYSTUFF.PAS with range checking turned off, while

```
TPC MYSTUFF /$R+
```

compiles it with range checking turned on. Note that if a {SR+} or {SR-} compiler directive appears in the source text, it overrides the /$R command-line option.

You can repeat the /$ option in order to specify multiple compiler directives:

```
TPC MYSTUFF /$R-/$I-/$V-/$F+
```

Alternately, TPC lets you write a list of directives (except for $M), separated by commas:
Note that only one dollar sign ($$) is needed.

In addition to changing switch directives, \$/S also lets you specify a program's memory allocation parameters, using the following format:

\$/M\text{stacksize,heapmin,heapmax}

where \text{stacksize} is the stack size, \text{heapmin} is the minimum heap size, and \text{heapmax} is the maximum heap size. The values are in bytes, and each is a decimal number unless it is preceded by a dollar sign ($$), in which case it is assumed to be hexadecimal. So, for example, the following command lines are equivalent:

\begin{verbatim}
TPC MYSTUFF /$M16384,256,4096
TPC MYSTUFF /$M$4000,$100,$1000
\end{verbatim}

Note that, because of its format, you cannot use the \$/M option in a list of directives separated by commas.

The conditional defines option

The \$/D option lets you define conditional symbols, corresponding to the \{\$DEFINE symbol\} compiler directive. The \$/D option must be followed by one or more conditional symbols, separated by semicolons (;). For example, the following command line

\begin{verbatim}
TPC MYSTUFF /DIOCHECK;DEBUG;LIST
\end{verbatim}

defines three conditional symbols, \text{iocheck}, \text{debug}, and \text{list}, for the compilation of \text{MYSTUFF.PAS}. This is equivalent to inserting

\begin{verbatim}
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
\end{verbatim}

at the beginning of \text{MYSTUFF.PAS}. If you specify multiple \$/D directives, you can concatenate the symbol lists. Therefore,

\begin{verbatim}
TPC MYSTUFF /DIOCHECK/DEBUG/DLIST
\end{verbatim}

is equivalent to the first example.

Compiler mode options

A few options affect how the compiler itself functions. These are \$/M (Make), \$/B (Build), \$/F (Find Error), \$/L (Link Buffer), and \$/Q (Quiet). As with the other options, you can use the hyphen format (remember to separate the options with at least one blank).
The make (/M) option | TPC has a built-in MAKE utility to aid in project maintenance. The /M option instructs TPC to check all units upon which the file being compiled depends.

A unit will be recompiled if

- The source file for that unit has been modified since the .TPU file was created.
- Any file included with the $I$ directive, or any .OBJ file linked in by the $L$ directive, is newer than the unit's .TPU file.
- The interface section of a unit referenced in a uses statement has changed.

Units in TURBO.TPL are excluded from this process.

If you were applying this option to the previous example, the command would be

```
TPC MYSTUFF /M
```

The build all (/B) option | Instead of relying on the /M option to determine what needs to be updated, you can tell TPC to update all units upon which your program depends using the /B option.

If you were using this option in the previous example, the command would be

```
TPC MYSTUFF /B
```

The find error (/F) option | When a program terminates due to a run-time error, it displays an error code and the address (segment:offset) at which the error occurred. By specifying that address in a /Fsegment:offset option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the $D$ compiler directive).

Suppose you have a file called TEST.PAS that contains the following program:

```pascal
program Test;
var
  x : Real;
begin
  x := 0;
  x := x / x;  { Force a divide by zero error }
end.
```

Chapter 3, Command-line compiler
First, compile this program using the command-line compiler:

```
TPC TEST
```

If you do a DIR TEST.*, DOS lists two files: TEST.PAS, your source code, and TEST.EXE, the executable file.

Now, type TEST to run. You’ll get a run-time error: “Run-time error 200 at 0000:003D.” Notice that you’re given an error code (200) and the address (0000:003D in hex) of the instruction pointer (CS:IP) where the error occurred. To figure out which line in your source caused the error, simply invoke the compiler, use /F and specify the segment and offset as reported in the error message:

```
C:\>TPC TEST /F0:3D
Turbo Pascal 7.0 Copyright (c) 1983,92 Borland International
10/02/92  14:09:53
TEST.PAS(7)
TEST.PAS(6) : Target address found.
   x := x / x;
```

In order for TPC to find the run-time error with /F, you must compile the program with all the same command-line parameters you used the first time you compiled it.

The compiler now gives you the file name and line number, and points to the offending line number and text in your source code.

As mentioned previously, you must compile your program and units with debug information enabled for TPC to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a {$O-} compiler directive or a /D- option, TPC will not be able to locate run-time errors.

The link buffer (/L) option

The /L option disables buffering in memory when .TPU files are linked to create an .EXE file. Turbo Pascal’s built-in linker makes two passes. In the first pass through the .TPU files, the linker marks every procedure that gets called by other procedures. In the second pass, it generates an .EXE file by extracting the marked procedures from the .TPU files.

By default, the .TPU files are kept in memory between the two passes; however, if the /L option is specified, they are read again from disk during the second pass. The default method is faster but requires more memory; for very large programs, you may have to specify /L to link successfully.
The quiet (/Q) option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When TPC is invoked with the quiet mode option

```
TPC MYSTUFF /Q
```

its output is limited to the sign-on message and the usual statistics at the end of compilation. If an error occurs, it will be reported.

Directory options

TPC supports several options that let you specify the five directory lists used by TPC: TPL & CFG, EXE & TPU, Include, Unit, and Object.

Excluding the EXE and TPU directory option, you may specify one or multiple directories for each command-line directory option. If you specify multiple directories, separate them with semicolons (;). For example, this command line tells TPC to search for Include files in C:\TP\INCLUDE and D:\INC after searching the current directory:

```
TPC MYSTUFF /IC:C:\TP\INCLUDE;D:\INC
```

If you specify multiple directives, the directory lists are concatenated. Therefore,

```
TPC MYSTUFF /IC:C:\TP\INCLUDE /ID:D:\INC
```

is equivalent to the first example.

The TPL & CFG directory (/T) option

TPC looks for two files when it is executed: TPC.CFG, the configuration file, and TURBO.TPL, the resident library file. TPC automatically searches the current directory and the directory containing TPC.EXE. The /T option lets you specify other directories in which to search. For example, you could say

```
TPC /TC:C:\TP\BIN MYSTUFF
```

If you want the /T option to affect the search for TPC.CFG, it must be the very first command-line argument, as in the previous example.
The EXE & TPU directory (/E) option

This option lets you tell TPC where to put the .EXE and .TPU files it creates. It takes a directory path as its argument:

```
TPC MYSTUFF /EC:TP\BIN
```

You can specify only one EXE and TPU directory

If no such option is given, TPC creates the .EXE and .TPU files in the same directories as their corresponding source files.

The include directories (/E) option

Turbo Pascal supports include files through the \{Sl filename\} compiler directive. The /I option lets you specify a list of directories in which to search for Include files.

The unit directories (/U) option

When you compile a program that uses units, TPC first attempts to find the units in TURBO.TPL (which is loaded along with TPC.EXE). If they cannot be found there, TPC searches for unitname.TPU in the current directory. The /U option lets you specify additional directories in which to search for units.

The object files directories (/O) option

Using \{SL filename\} compiler directives, Turbo Pascal lets you link in .OBJ files containing external assembly language routines, as explained in Chapter 23, “Linking assembler code,” in the Language Guide. The /O option lets you specify a list of directories in which to search for such .OBJ files.

Debug options

TPC has two command-line options that enable you to generate debugging information: the map file option and the debugging option.

The map file (/G) option

Unlike the binary format of .EXE and .TPU files, a .MAP file is a legible text file that can be output on a printer or loaded into the editor.

The /G option instructs TPC to generate a .MAP file that shows the layout of the .EXE file. The /G option must be followed by the letter S, P, or D to indicate the desired level of information in the .MAP file. A .MAP file is divided into three sections:

- Segment
- Publics
- Line Numbers

The /GS option outputs only the Segment section, /GP outputs the Segment and Publics section, and /GD outputs all three sections.
For modules (program and units) compiled in the \{SD+,L+\} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the \{SD+,L-\} state, only symbols defined in a unit's interface part are listed in the Publics section. For modules compiled in the \{SD-\} state, there are no entries in the Line Numbers section.

When you specify the `N` option on the command line, TPC appends Turbo Debugger-compatible debug information at the end of the .EXE file. Turbo Debugger includes both source- and machine-level debugging and powerful breakpoints including breakpoints with conditionals or expressions attached to them.

Even though the debug information generated by `N` makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and if it is executed, the .EXE file does not require additional memory.

The extent of debug information appended to the .EXE file depends on the setting of the `SD` and `SL` compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the \{SD+,L+\} state, which is the default, all constant, variable, type, procedure, and function symbols become known to the debugger. In the \{SD+,L-\} state, only symbols defined in a unit's interface section become known to the debugger. In the \{SD-\} state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

### The TPC.CFG file

You can set up a list of options in a configuration file called TPC.CFG, which will then be used in addition to the options entered on the command line. Each line in TPC.CFG corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a TPC.CFG file, you can change the default setting of any command-line option.

TPC lets you enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a TPC.CFG file, you can still override them on the command line.
When TPC starts, it looks for TPC.CFG in the current directory. If the file isn’t found there, TPC looks in the directory where TPC.EXE resides. To force TPC to look in a specific list of directories (in addition to the current directory), specify a /T command-line option as the first option on the command line.

If TPC.CFG contains a line that does not start with a slash (/) or a hyphen (-), that line defines a default file name to compile. In that case, starting TPC with an empty command line (or with a command line consisting of command-line options only and no file name) will cause it to compile the default file name, instead of displaying a syntax summary.

Here’s an example TPC.CFG file, defining some default directories for include, object, and unit files, and changing the default states of the $F and $S compiler directives:

```
/IC:\TP\INC;C:\TP\SRC
/OC:\TP\ASM
/UC:\TP\UNIT
/$F+
/$S-
```

Now, if you type

```
TPC MYSTUFF
```

at the system prompt, TPC acts as if you had typed in the following:

```
TPC /IC:\TP\INC;C:\TP\SRC /OC:\TP\ASM /UC:\TP\UNIT /$F+ /$S- MYSTUFF
```
This chapter describes the possible error messages you can get from Turbo Pascal during program development. The error messages are grouped according to the categories listed in Table 4.1. Run-time errors are subdivided into DOS, I/O, critical, and fatal errors. Within each of the groups, the errors are listed in numerical order.

<table>
<thead>
<tr>
<th>Type of message</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>See page 238.</td>
</tr>
<tr>
<td>DOS</td>
<td>See page 258.</td>
</tr>
<tr>
<td>I/O</td>
<td>See page 260.</td>
</tr>
<tr>
<td>Critical</td>
<td>See page 261.</td>
</tr>
<tr>
<td>Fatal</td>
<td>See page 261.</td>
</tr>
</tbody>
</table>

**Compiler error messages**

Whenever possible, the compiler will display additional diagnostic information in the form of an identifier or a file name. For example,

Error 15: File not found (GRAPH.TPU).

When an error is detected, Turbo Pascal (in the IDE) automatically loads the source file and places the cursor at the error. The command-line compiler displays the error message and number...
and the source line, and uses a caret (^) to indicate where the error occurred. Note, however, that some errors are not detected until a little later in the source text. For example, a type mismatch in an assignment statement cannot be detected until the entire expression after the := has been evaluated. In such cases, look for the error to the left of or above the cursor.

1 **Out of memory.**

This error occurs when the compiler runs out of memory. Try these possible solutions:

- If Compile | Destination is set to Memory, set it to Disk in the integrated environment.
- If Options | Linker | Link Buffer is set to Memory, toggle it to Disk. Use a /L option to place the link buffer on disk when using the command-line compiler.

If these suggestions don’t help, your program or unit might simply be too large to compile in the amount of memory available, and you might have to break it into two or more smaller units.

2 **Identifier expected.**

An identifier was expected at this point. You might be trying to redeclare a reserved word.

3 **Unknown identifier.**

This identifier has not been declared, or might not be visible within the current scope.

4 **Duplicate identifier.**

The identifier already represents a program or unit’s name, a constant, a variable, a type, a procedure, or a function declared within the current block.

5 **Syntax error.**

An illegal character was found in the source text. You might have forgotten the quotes around a string constant.
6 Error in real constant.
The syntax of type constants is defined in Chapter 2, “Tokens,” in the Language Guide.

7 Error in integer constant.
The syntax of integer-type constants is defined in Chapter 2, “Tokens,” in the Language Guide. Note that whole real numbers outside the maximum integer range must be followed by a decimal point and a zero; for example, 12345678912.0.

8 String constant exceeds line.
You have most likely forgotten the ending quote in a string constant.

10 Unexpected end of file.
You might have gotten this error message for one of the following reasons:

- Your source file ends before the final end of the main statement part. Most likely, your begin and end statements do not match.
- An Include file ends in the middle of a statement part. Every statement part must be entirely contained in one file.
- You didn’t close a comment.

11 Line too long.
The maximum line length is 127 characters.

12 Type identifier expected.
The identifier does not denote a type as it should.

13 Too many open files.
If this error occurs, your CONFIG.SYS file does not include a FILES=xx entry or the entry specifies too few files. Increase the number to some suitable value, such as 20.
14 Invalid file name.
The file name is invalid or specifies a nonexistent path.

15 File not found.
The compiler could not find the file in the current directory or in any of the search directories that apply to this type of file.

16 Disk full.
Delete some files or use a different disk.

17 Invalid compiler directive.
The compiler directive letter is unknown, one of the compiler directive parameters is invalid, or you are using a global compiler directive when compilation of the body of the program has begun.

18 Too many files.
There are too many files involved in the compilation of the program or unit. Try to use fewer files. For example, you could merge include files. You could also shorten the file names or move all the files into one directory and make it the current directory at compile time.

19 Undefined type in pointer definition.
The type was referenced in a pointer-type declaration previously, but it was never declared.

20 Variable identifier expected.
The identifier does not denote a variable as it should.

21 Error in type.
This symbol cannot start a type definition.

22 Structure too large.
The maximum allowable size of a structured type is 65,535 bytes.
23 Set base type out of range.
The base type of a set must be a subrange with bounds in the range 0..255 or an enumerated type with no more than 256 possible values.

24 File components may not be files or objects.
file of file and file of object constructs are not allowed; nor is a file of any structured type that includes an object type or file type.

25 Invalid string length.
The declared maximum length of a string must be in the range 1..255.

26 Type mismatch.
This is due to one of the following:

- Incompatible types of the variable and the expression in an assignment statement
- Incompatible types of the actual and formal parameter in a call to a procedure or function
- An expression type that is incompatible with the index type in array indexing
- Incompatible types of operands in an expression

27 Invalid subrange base type.
All ordinal types are valid base types.

28 Lower bound greater than upper bound.
The declaration of a subrange type specifies a lower bound greater than the upper bound.

29 Ordinal type expected.
Real types, string types, structured types, and pointer types are not allowed here.

30 Integer constant expected.
31 Constant expected.

32 Integer or real constant expected.

33 Pointer type identifier expected.
The identifier does not denote a pointer type as it should.

34 Invalid function result type.
Valid function result types are all simple types, string types, and pointer types.

35 Label identifier expected.
The identifier does not denote a label as it should.

36 BEGIN expected.
A begin is expected here, or there is an error in the block structure of the unit or program.

37 END expected.
An end is expected here, or there is an error in the block structure of the unit or program.

38 Integer expression expected.
The preceding expression must be of an integer type.

39 Ordinal expression expected.
The preceding expression must be of an ordinal type.

40 Boolean expression expected.
The preceding expression must be of a boolean type.

41 Operand types do not match operator.
The operator cannot be applied to operands of this type, for example, ‘A’ div ‘2’.
42 Error in expression.
This symbol cannot participate in an expression in the way it does. You might have forgotten to write an operator between two operands.

43 Illegal assignment.
- Files and untyped variables cannot be assigned values.
- A function identifier can only be assigned values within the statement part of the function.

44 Field identifier expected.
The identifier does not denote a field in the corresponding record or object variable.

45 Object file too large.
Turbo Pascal cannot link in .OBJ files larger than 64K.

46 Undefined external.
The external procedure or function did not have a matching PUBLIC definition in an object file. Make sure you have specified all object files in {$L filename} directives, and check the spelling of the procedure or function identifier in the .ASM file.

47 Invalid object file record.
The .OBJ file contains an invalid object record; make sure the file is in fact an .OBJ file.

48 Code segment too large.
The maximum size of the code of a program or unit is 65,520 bytes. If you are compiling a program, move some procedures or functions into a unit. If you are compiling a unit, break it into two or more units.
49 Data segment too large.
The maximum size of a program's data segment is 65,520 bytes, including data declared by the used units. If you need more global data than this, declare the larger structures as pointers, and allocate them dynamically using the New procedure.

50 DO expected.
The reserved word do does not appear where it should.

51 Invalid PUBLIC definition.
- Two or more PUBLIC directives in assembly language define the same identifier.
- The .OBJ file defines PUBLIC symbols that do not reside in the CODE segment.

52 Invalid EXTRN definition.
- The identifier was referred to through an EXTRN directive in assembly language, but it is not declared in the Pascal program or unit, nor in the interface part of any of the used units.
- The identifier denotes an absolute variable.
- The identifier denotes an inline procedure or function.

53 Too many EXTRN definitions.
Turbo Pascal cannot handle .OBJ files with more than 256 EXTRN definitions.

54 OF expected.
The reserved word of does not appear where it should.

55 INTERFACE expected.
The reserved word interface does not appear where it should.

56 Invalid relocatable reference.
- The .OBJ file contains data and relocatable references in segments other than CODE. For example, you are attempting to declare initialized variables in the DATA segment.
• The .OBJ file contains byte-sized references to relocatable symbols. This error occurs if you use the \texttt{HIGH} and \texttt{LOW} operators with relocatable symbols or if you refer to relocatable symbols in \texttt{DB} directives.

• An operand refers to a relocatable symbol that was not defined in the \texttt{CODE} segment or in the \texttt{DATA} segment.

• An operand refers to an \texttt{EXTRN} procedure or function with an offset, for example, \texttt{CALL SortProc+8}.

57 \textbf{THEN expected.}

The reserved word \texttt{then} does not appear where it should.

58 \textbf{TO or DOWNTO expected.}

The reserved word \texttt{to} or \texttt{downto} does not appear where it should.

59 \textbf{Undefined forward.}

• The procedure or function was declared in the \texttt{interface} part of a unit, but its definition never occurred in the \texttt{implementation} part.

• The procedure or function was declared with \texttt{forward}, but its definition was never found.

61 \textbf{Invalid typecast.}

• The sizes of the variable reference and the destination type differ in a variable typecast.

• You are attempting to typecast an expression where only a variable reference is allowed.

62 \textbf{Division by zero.}

The preceding operand attempts to divide by zero.

63 \textbf{Invalid file type.}

The file type is not supported by the file-handling procedure; for example, \texttt{Readln} with a typed file or \texttt{Seek} with a text file.
64 Cannot Read or Write variables of this type.

- Read and Readln can input variables of character, integer, real, and string types.
- Write and Writeln can output variables of character, integer, real, string, and boolean types.

65 Pointer variable expected.
The preceding variable must be of a pointer type.

66 String variable expected.
The preceding variable must be of a string type.

67 String expression expected.
The preceding expression must be of a string type.

68 Circular unit reference.
Two units are not allowed to use each other in the interface part. It is legal for two units to use each other in the implementation part. Rearrange your uses clauses so that circular references occur only in the implementation parts. For more details, see “Circular unit references” in Chapter 10 in the Language Guide.

69 Unit name mismatch.
The name of the unit found in the .TPU file does not match the name specified in the uses clause.

70 Unit version mismatch.
One or more of the units used by this unit have been changed since the unit was compiled. Use Compile | Make or Compile | Build in the IDE and /M or /B options in the command-line compiler to automatically compile units that need recompilation.

71 Internal stack overflow.
The compiler’s internal stack is exhausted due to too many levels of nested statements. Rearrange your code so it is not nested so deeply. For example, move the inner levels of nested statements into a separate procedure.
72 Unit file format error.
The .TPU file is somehow invalid; make sure it is in fact a .TPU file. The .TPU file might have been created with an older version of Turbo Pascal. In this case, a new .TPU must be created by recompiling the source file.

73 IMPLEMENTATION expected.
The reserved word implementation does not appear where it should. You are probably including the implementation of a procedure, function, or method in the interface part of the unit.

74 Constant and case types do not match.
The type of the case constant is incompatible with the case statement's selector expression.

75 Record or object variable expected.
The preceding variable must be of a record or object type.

76 Constant out of range.
You are trying to
- Index an array with an out-of-range constant
- Assign an out-of-range constant to a variable
- Pass an out-of-range constant as a parameter to a procedure or function

77 File variable expected.
The preceding variable must be of a file type.

78 Pointer expression expected.
The preceding expression must be of a pointer type.

79 Integer or real expression expected.
The preceding expression must be of an integer or a real type.
80 Label not within current block.
A goto statement cannot reference a label outside the current block.

81 Label already defined.
The label already marks a statement.

82 Undefined label in preceding statement part.
The label was declared and referenced in the preceding statement part, but it was never defined.

83 Invalid @ argument.
Valid arguments are variable references and procedure or function identifiers.

84 UNIT expected.
The reserved word unit does not appear where it should.

85 ";" expected.
A semicolon does not appear where it should.

86 ":" expected.
A colon does not appear where it should.

87 "," expected.
A comma does not appear where it should.

88 "(" expected.
An opening parenthesis does not appear where it should.

89 ")" expected.
A closing parenthesis does not appear where it should.
90 "=" expected.
An equal sign does not appear where it should.

91 ":=" expected.
An assignment operator does not appear where it should.

92 "[" or "(." expected.
A left bracket does not appear where it should.

93 "]" or ".)" expected.
A right bracket does not appear where it should.

94 "." expected.
A period does not appear where it should. Check to make sure that a type is not being used as a variable or that the name of the program does not override an important identifier from another unit.

95 ".." expected.
A subrange does not appear where it should.

96 Too many variables.
- The total size of the global variables declared within a program or unit cannot exceed 64K.
- The total size of the local variables declared within a procedure or function cannot exceed 64K.

97 Invalid FOR control variable.
The for statement control variable must be a simple variable defined in the declaration part of the current subprogram.

98 Integer variable expected.
The preceding variable must be of an integer type.
99 File types are not allowed here.
A typed constant cannot be of a file type.

100 String length mismatch.
The length of the string constant does not match the number of components in the character array.

101 Invalid ordering of fields.
The fields of a record- or object-type constant must be written in the order of declaration.

102 String constant expected.
A string constant does not appear where it should.

103 Integer or real variable expected.
The preceding variable must be of an integer or real type.

104 Ordinal variable expected.
The preceding variable must be of an ordinal type.

105 INLINE error.
The < operator is not allowed in conjunction with relocatable references to variables—such references are always word-sized.

106 Character expression expected.
The preceding expression must be of a character type.

107 Too many relocation items.
The size of the relocation table part of the .EXE file exceeds 64K, which is Turbo Pascal’s upper limit. If you encounter this error, your program is simply too big for Turbo Pascal’s linker to handle. It is probably also too big for DOS to execute. You will have to split the program into a “main” part that executes two or more “subprogram” parts using the Exec procedure in the Dos unit.
108 Overflow in arithmetic operation.
The result of the preceding arithmetic operation is not in the Longint range (-2147483648..2147483647). Correct the operation or use real-type values instead of integer-type values.

109 No enclosing FOR, WHILE, or REPEAT statement.
The Break and Continue standard procedures cannot be used outside a for, while, or repeat statement.

112 CASE constant out of range.
For integer-type case statements, the constants must be within the range -32768..32767.

113 Error in statement.
This symbol cannot start a statement.

114 Cannot call an interrupt procedure.
You cannot directly call an interrupt procedure.

116 Must be in 8087 mode to compile this.
This construct can only be compiled in the {$N+} state. Operations on the 80x87 real types (Single, Double, Extended, and Comp) are not allowed in the {$N-} state.

117 Target address not found.
The Search | Find Error command in the IDE or the /F option in the command-line version could not locate a statement that corresponds to the specified address.

118 Include files are not allowed here.
Every statement part must be entirely contained in one file.

119 No inherited methods are accessible here.
You are using the inherited keyword outside a method or in a method of an object type that has no ancestor.
121 Invalid qualifier.
You are trying to do one of the following:
- Index a variable that is not an array.
- Specify fields in a variable that is not a record.
- Dereference a variable that is not a pointer.

122 Invalid variable reference.
The preceding construct follows the syntax of a variable reference, but it does not denote a memory location. Most likely, you are trying to modify a const parameter, or you are calling a pointer function but forgetting to dereference the result.

123 Too many symbols.
The program or unit declares more than 64K of symbols. If you are compiling with {$D+}, try turning it off—note, however, that this will prevent you from finding run-time errors in that module. Otherwise, you could try moving some declarations into a separate unit.

124 Statement part too large.
Turbo Pascal limits the size of a statement part to about 24K. If you encounter this error, move sections of the statement part into one or more procedures. In any case, with a statement part of that size, it's worth the effort to clarify the structure of your program.

126 Files must be var parameters.
You are attempting to declare a file-type value parameter. File-type parameters must be var parameters.

127 Too many conditional symbols.
There is not enough room to define further conditional symbols. Try to eliminate some symbols, or shorten some of the symbolic names.
128 Misplaced conditional directive.
The compiler encountered an {$ELSE} or {$ENDIF} directive without a matching {$IFDEF}, {$IFNDEF}, or {$IFOPT} directive.

129 ENDF directive missing.
The source file ended within a conditional compilation construct. There must be an equal number of {$IFDEF}s and {$ENDIF}s in a source file.

130 Error in initial conditional defines.
The initial conditional symbols specified in Options | Compiler | Conditional Defines (in the IDE) or in a /D directive (with the command-line compiler) are invalid. Turbo Pascal expects zero or more identifiers separated by blanks, commas, or semicolons.

131 Header does not match previous definition.
The procedure or function header specified in the interface part or forward declaration does not match this header.

133 Cannot evaluate this expression.
You are attempting to use a non-supported feature in a constant expression. For example, you’re attempting to use the Sin function in a const declaration. For a description of the allowed syntax of constant expressions, see Chapter 3, “Constants,” in the Language Guide.

134 Expression incorrectly terminated.
Turbo Pascal expects either an operator or the end of the expression at this point, but found neither.

135 Invalid format specifier.
You are using an invalid format specifier, or the numeric argument of a format specifier is out of range. For a list of valid format specifiers, see Chapter 5, “Debugging in the IDE,” in the User’s Guide.
136 Invalid indirect reference.
The statement attempts to make an invalid indirect reference. For example, you are using an absolute variable whose base variable is not known in the current module, or you are using an inline routine that references a variable not known in the current module.

137 Structured variables are not allowed here.
You are attempting to perform a non-supported operation on a structured variable. For example, you are trying to multiply two records.

138 Cannot evaluate without system unit.
Your TURBO.TPL library must contain the System unit for the debugger to be able to evaluate expressions.

139 Cannot access this symbol.
A program's entire set of symbols is available as soon as you have compiled the program. However, certain symbols, such as variables, cannot be accessed until you actually run the program.

140 Invalid floating-point operation.
An operation on two real type values produced an overflow or a division by zero.

141 Cannot compile overlays to memory.
A program that uses overlays must be compiled to disk.

142 Pointer or procedural variable expected.
The Assigned standard function requires the argument to be a variable of a pointer or procedural type.

143 Invalid procedure or function reference.
- You are attempting to call a procedure in an expression.
- If you are going to assign a procedure or function to a procedural variable, it must be compiled in the {$F+} state and cannot be declared with inline or interrupt.
144 Cannot overlay this unit.
You are attempting to overlay a unit that wasn’t compiled in the \{O+\} state.

146 File access denied.
The file could not be opened or created. Most likely, the compiler is trying to write to a read-only file.

147 Object type expected.
The identifier does not denote an object type.

148 Local object types are not allowed.
Object types can be defined only in the outermost scope of a program or unit. Object-type definitions within procedures and functions are not allowed.

149 VIRTUAL expected.
The reserved word virtual is missing.

150 Method identifier expected.
The identifier does not denote a method.

151 Virtual constructors are not allowed.
A constructor method must be static.

152 Constructor identifier expected.
The identifier does not denote a constructor.

153 Destructor identifier expected.
The identifier does not denote a destructor.

154 Fail only allowed within constructors.
The Fail standard procedure can be used only within constructors.
155 Invalid combination of opcode and operands.

The assembler opcode does not accept this combination of operands. Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, INC AX, BX or MOV AX.
- The number of operands is correct, but their types or order do not match the opcode; for example, DEC 1, MOV AX, CL or MOV 1, AX.

156 Memory reference expected.

The assembler operand is not a memory reference, which is required here. Most likely you have forgotten to put square brackets around an index register operand, for example, MOV AX, BX+SI instead of MOV AX, [BX+SI].

157 Cannot add or subtract relocatable symbols.

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that Var is a variable and Const is a constant, then the instructions MOV AX, Const+Const and MOV AX, Var+Const are valid, but MOV AX, Var+Var is not.

158 Invalid register combination.

Valid index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. Other index register combinations (such as [AX], [BP+BX], and [SI+DX]) are not allowed.

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed via the BP register. The assembler automatically adds [BP] in references to such variables, so even though a construct like Local[BX] (where Local is a local variable) appears valid, it is not since the final operand would become Local[BP+BX].
159 286/287 instructions are not enabled.
Use a \$G+ compiler directive to enable 286/287 opcodes, but be aware that the resulting code cannot be run on 8086- and 8088-based machines.

160 Invalid symbol reference.
This symbol cannot be accessed in an assembler operand. Possible causes follow:

- You are attempting to access a standard procedure, a standard function, or the Mem, MemW, MemL, Port, or PortW special arrays in an assembler operand.
- You are attempting to access a string, floating-point, or set constant in an assembler operand.
- You are attempting to access an inline procedure or function in an assembler operand.
- You are attempting to access the @Result special symbol outside a function.
- You are attempting to generate a short JMP instruction that jumps to something other than a label.

161 Code generation error.
The preceding statement part contains a LOOPNE, LOOPE, LOOP, or JCXZ instruction that cannot reach its target label.

162 ASM expected.
You are attempting to compile a built-in assembler function or procedure that contains a begin...end statement instead of asm...end.

Run-time errors

Certain errors at run time cause the program to display an error message and terminate:

Run-time error nnn at xxxx:yyyy

where nnn is the run-time error number, and xxxx:yyyy is the run-time error address (segment and offset).
The run-time errors are divided into four categories: DOS errors, 1 through 99; I/O errors, 100 through 149, critical errors, 150 through 199; and fatal errors, 200 through 255.

**DOS errors**

1. **Invalid function number.**
   You made a call to a nonexistent DOS function.

2. **File not found.**
   Reported by *Reset, Append, Rename, Rewrite* if the file name is invalid, or *Erase* if the name assigned to the file variable does not specify an existing file.

3. **Path not found.**
   - Reported by *Reset, Rewrite, Append, Rename, or Erase* if the name assigned to the file variable is invalid or specifies a nonexistent subdirectory.
   - Reported by *ChDir, MkDir, or RmDir* if the path is invalid or specifies a nonexistent subdirectory.

4. **Too many open files.**
   Reported by *Reset, Rewrite, or Append* if the program has too many open files. DOS never allows more than 15 open files per process. If you get this error with less than 15 open files, it might indicate that the CONFIG.SYS file does not include a FILES=xx entry or that the entry specifies too few files. Increase the number to some suitable value, such as 20.

5. **File access denied.**
   - Reported by *Reset* or *Append* if *FileMode* allows writing and the name assigned to the file variable specifies a directory or a read-only file.
   - Reported by *Rewrite* if the directory is full or if the name assigned to the file variable specifies a directory or an existing read-only file.
   - Reported by *Rename* if the name assigned to the file variable specifies a directory or if the new name specifies an existing file.
- Reported by *Erase* if the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *MkDir* if a file with the same name exists in the parent directory, if there is no room in the parent directory, or if the path specifies a device.
- Reported by *RmDir* if the directory isn’t empty, if the path doesn’t specify a directory, or if the path specifies the root directory.
- Reported by *Read* or *BlockRead* on a typed or untyped file if the file is not open for reading.
- Reported by *Write* or *BlockWrite* on a typed or untyped file if the file is not open for writing.

6 Invalid file handle.

This error is reported if an invalid file handle is passed to a DOS system call. It should never occur; if it does, it is an indication that the file variable is somehow trashed.

12 Invalid file access code.

Reported by *Reset* or *Append* on a typed or untyped file if the value of *FileMode* is invalid.

15 Invalid drive number.

Reported by *GetDir* or *ChDir* if the drive number is invalid.

16 Cannot remove current directory.

Reported by *RmDir* if the path specifies the current directory.

17 Cannot rename across drives.

Reported by *Rename* if both names are not on the same drive.

18 No more files.

Reported by the *DosError* variable in the *Dos* and *WinDos* units when a call to *FindFirst* or *FindNext* finds no files matching the specified file name and set of attributes.
I/O errors

These errors cause termination if the particular statement was compiled in the \( \{\text{I+}\} \) state. In the \( \{\text{I-}\} \) state, the program continues to execute, and the error is reported by the \textit{IOResult} function.

100 Disk read error.
Reported by \textit{Read} on a typed file if you attempt to read past the end of the file.

101 Disk write error.
Reported by \textit{Close}, \textit{Write}, \textit{Writeln}, or \textit{Flush} if the disk becomes full.

102 File not assigned.
Reported by \textit{Reset}, \textit{Rewrite}, \textit{Append}, \textit{Rename}, and \textit{Erase} if the file variable has not been assigned a name through a call to \textit{Assign}.

103 File not open.
Reported by \textit{Close}, \textit{Read}, \textit{Write}, \textit{Seek}, \textit{Eof}, \textit{FilePos}, \textit{FileSize}, \textit{Flush}, \textit{BlockRead}, or \textit{BlockWrite} if the file is not open.

104 File not open for input.
Reported by \textit{Read}, \textit{Readln}, \textit{Eof}, \textit{Eoln}, \textit{SeekEof}, or \textit{SeekEoln} on a text file if the file is not open for input.

105 File not open for output.
Reported by \textit{Write} and \textit{Writeln} on a text file if the file is not open for output.

106 Invalid numeric format.
Reported by \textit{Read} or \textit{Readln} if a numeric value read from a text file does not conform to the proper numeric format.
Critical Errors

For more information about these errors, see your DOS programmer's reference manual.

150  Disk is write protected.
151  Unknown unit.
152  Drive not ready.
153  Unknown command.
154  CRC error in data.
155  Bad drive request structure length.
156  Disk seek error.
157  Unknown media type.
158  Sector not found.
159  Printer out of paper.
160  Device write fault.
161  Device read fault.
162  Hardware failure.

Dos reports this error as a result of sharing violations and various network errors.

Fatal errors

These errors always immediately terminate the program.

200  Division by zero.

The program attempted to divide a number by zero during a /, mod, or div operation.

201  Range check error.

This error is reported by statements compiled in the [SR+] state when one of the following situations arises:

- The index expression of an array qualifier was out of range.
- You attempted to assign an out-of-range value to a variable.
You attempted to assign an out-of-range value as a parameter to a procedure or function.

202 Stack overflow error.
This error is reported on entry to a procedure or function compiled in the {SS+} state when there is not enough stack space to allocate the subprogram’s local variables. Increase the size of the stack by using the $M compiler directive.
This error might also be caused by infinite recursion, or by an assembly language procedure that does not maintain the stack properly.

203 Heap overflow error.
This error is reported by New or GetMem when there is not enough free space in the heap to allocate a block of the requested size.
For a complete discussion of the heap manager, see Chapter 19, “Memory issues,” in the Language Guide.

204 Invalid pointer operation.
This error is reported by Dispose or FreeMem if the pointer is nil or points to a location outside the heap.

205 Floating point overflow.
A floating-point operation produced a number too large for Turbo Pascal or the numeric coprocessor (if any) to handle.

206 Floating point underflow.
A floating-point operation produced an underflow. This error is only reported if you are using the 8087 numeric coprocessor with a control word that unmarks underflow exceptions. By default, an underflow causes a result of zero to be returned.

207 Invalid floating point operation.
The real value passed to Trunc or Round could not be converted to an integer within the Longint range (-2,147,483,648 to 2,147,483,647).
• The argument passed to the \textit{Sqrt} function was negative.
• The argument passed to the \textit{Ln} function was zero or negative.
• An 8087 stack overflow occurred. For further details on correctly programming the 8087, see Chapter 14, "Using the 80x87," in the Language Guide.

208 Overlay manager not installed.
Your program is calling an overlaid procedure or function, but the overlay manager is not installed. Most likely, you are not calling \textit{OvrInit}, or the call to \textit{OvrInit} failed. Note that, if you have initialization code in any of your overlaid units, you must create an additional non-overlaid unit which calls \textit{OvrInit}, and use that unit before any of the overlaid units.

209 Overlay file read error.
A read error occurred when the overlay manager tried to read an overlay from the overlay file.

210 Object not initialized.
With range-checking on, you made a call to an object’s virtual method, before the object had been initialized via a constructor call.

211 Call to abstract method.
This error is generated by the \textit{Abstract} procedure in the \textit{Objects} unit; it indicates that your program tried to execute an abstract virtual method. When an object type contains one or more abstract methods it is called an \textit{abstract object type}. It is an error to instantiate objects of an abstract type—abstract object types exist only so that you can inherit from them and override the abstract methods.

For example, the \textit{Compare} method of the \textit{TSortedCollection} type in the \textit{Objects} unit is abstract, indicating that to implement a sorted collection you must create an object type that inherits from \textit{TSortedCollection} and overrides the \textit{Compare} method.
212 **Stream registration error.**

This error is generated by the `RegisterType` procedure in the `Objects` unit indicating that one of the following errors has occurred:

- The stream registration record does not reside in the data segment.
- The `ObjType` field of the stream registration record is zero.
- The type has already been registered.
- Another type with the same `ObjType` value already exists.

213 **Collection index out of range.**

The index passed to a method of a `TCollection` is out of range.

214 **Collection overflow error.**

The error is reported by a `TCollection` if an attempt is made to add an element when the collection cannot be expanded.

215 **Arithmetic overflow error.**

This error is reported by statements compiled in the `$Q+$` state when an integer arithmetic operation caused an overflow, such as when the result of the operation was outside the supported range.
Editor reference

The tables in this appendix list all the available editing commands you can use in the Turbo Pascal IDE. If two sets of key combinations can be used for a single command, the second set is listed as an alternate key combination. Footnoted references in Table A.1 mark those commands that are described in depth in Tables A.2, A.3, and A.4.
Table A.1
Editing commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Alternate Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cursor movement commands</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character left</td>
<td>←</td>
<td>Ctrl+S</td>
</tr>
<tr>
<td>Character right</td>
<td>→</td>
<td>Ctrl+D</td>
</tr>
<tr>
<td>Word left</td>
<td>Ctrl+←</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Word right</td>
<td>Ctrl+→</td>
<td>Ctrl+F</td>
</tr>
<tr>
<td>Line up</td>
<td>↑</td>
<td>Ctrl+E</td>
</tr>
<tr>
<td>Line down</td>
<td>↓</td>
<td>Ctrl+X</td>
</tr>
<tr>
<td>Scroll up one line</td>
<td>Ctrl+W</td>
<td></td>
</tr>
<tr>
<td>Scroll down one line</td>
<td>Ctrl+Z</td>
<td></td>
</tr>
<tr>
<td>Page up</td>
<td>PgUp</td>
<td>Ctrl+R</td>
</tr>
<tr>
<td>Page down</td>
<td>PgDn</td>
<td>Ctrl+C</td>
</tr>
<tr>
<td>Beginning of line</td>
<td>Home</td>
<td>Ctrl+Q S</td>
</tr>
<tr>
<td>End of line</td>
<td>End</td>
<td>Ctrl+Q D</td>
</tr>
<tr>
<td>Top of window</td>
<td>Ctrl+Q E</td>
<td>Ctrl+Home</td>
</tr>
<tr>
<td>Bottom of window</td>
<td>Ctrl+Q X</td>
<td>Ctrl+End</td>
</tr>
<tr>
<td>Top of file</td>
<td>Ctrl+Q R</td>
<td>Ctrl+PgUp</td>
</tr>
<tr>
<td>Bottom of file</td>
<td>Ctrl+Q C</td>
<td>Ctrl+PgDn</td>
</tr>
<tr>
<td>Move to previous position</td>
<td>Ctrl+Q P</td>
<td></td>
</tr>
<tr>
<td><strong>Insert and delete commands</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete character</td>
<td>Del</td>
<td>Ctrl+G</td>
</tr>
<tr>
<td>Delete character to left</td>
<td>Backspace</td>
<td>Ctrl+H</td>
</tr>
<tr>
<td>Delete line</td>
<td>Ctrl+Y</td>
<td></td>
</tr>
<tr>
<td>Delete to end of line</td>
<td>Ctrl+Q Y</td>
<td></td>
</tr>
<tr>
<td>Delete to end of word</td>
<td>Ctrl+T</td>
<td></td>
</tr>
<tr>
<td>Insert newline</td>
<td>Ctrl+N</td>
<td></td>
</tr>
<tr>
<td>Insert mode on/off</td>
<td>Ins</td>
<td>Ctrl+V</td>
</tr>
<tr>
<td><strong>Block commands</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move to beginning of block</td>
<td>Ctrl+Q B</td>
<td></td>
</tr>
<tr>
<td>Move to end of block</td>
<td>Ctrl+Q K</td>
<td></td>
</tr>
<tr>
<td>Set beginning of block §</td>
<td>Ctrl+K B</td>
<td></td>
</tr>
<tr>
<td>Set end of block §</td>
<td>Ctrl+K K</td>
<td></td>
</tr>
<tr>
<td>Exit to menu bar</td>
<td>Ctrl+K D</td>
<td></td>
</tr>
<tr>
<td>Hide/Show block §</td>
<td>Ctrl+K H</td>
<td></td>
</tr>
<tr>
<td>Mark line</td>
<td>Ctrl+K L</td>
<td></td>
</tr>
<tr>
<td>Print selected block §</td>
<td>Ctrl+K P</td>
<td></td>
</tr>
<tr>
<td>Mark word</td>
<td>Ctrl+K T</td>
<td></td>
</tr>
<tr>
<td>Delete block</td>
<td>Ctrl+K Y</td>
<td></td>
</tr>
</tbody>
</table>

* \( n \) represents a number from 0 to 9.
† Enter control characters by first pressing \( Ctrl+P \), then pressing the desired control character.
‡ See Table A.2.
§ See Table A.3.
# See Table A.4.

A word is defined as a sequence of characters separated by one of the following: space \(<>\), \(,\), \(.\), \(),\), \(\)\), \(^\^\), \(\^\), \(*\), \(+\), \(-\), \(/\), \($\), \#), \(=\), \(\sim\), \(!\), \(?\), \(\&\), \(':\), \(\%\), \('\), \('\), \('@\), \(\,\), and all control and graphic characters.
Table A.1: Editing commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Alternate Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy block §</td>
<td>Ctrl+K C</td>
<td></td>
</tr>
<tr>
<td>Move block §</td>
<td>Ctrl+K V</td>
<td></td>
</tr>
<tr>
<td>Copy to Clipboard †</td>
<td>Ctrl+Ins</td>
<td></td>
</tr>
<tr>
<td>Cut to Clipboard ‡</td>
<td>Shift+Del</td>
<td></td>
</tr>
<tr>
<td>Delete block †</td>
<td>Ctrl+Del</td>
<td></td>
</tr>
<tr>
<td>Indent block</td>
<td>Ctrl+K I</td>
<td></td>
</tr>
<tr>
<td>Paste from Clipboard ‡</td>
<td>Shift+Ins</td>
<td></td>
</tr>
<tr>
<td>Read block from disk ‡</td>
<td>Ctrl+K R</td>
<td></td>
</tr>
<tr>
<td>Unindent block</td>
<td>Ctrl+K U</td>
<td></td>
</tr>
<tr>
<td>Write block to disk ‡</td>
<td>Ctrl+K W</td>
<td></td>
</tr>
</tbody>
</table>

**Extending selected blocks**

- Left one character: Shift+←
- Right one character: Shift+→
- End of line: Shift+End
- Beginning of line: Shift+Home
- Same column on next line: Shift+↓
- Same column on previous line: Shift+↑
- One page down: Shift+PgDn
- One page up: Shift+PgUp
- Left one word: Shift+Ctrl+←
- Right one word: Shift+Ctrl+→
- End of file: Shift+Ctrl+End
- Beginning of file: Shift+Ctrl+Home

**Other editing commands**

- Autoindent mode on/off #: Ctrl+O I
- Cursor through tabs on/off #: Ctrl+O R
- Exit the IDE: Alt+X
- Find place marker #: Ctrl+Q n *
- Help: F1
- Help index: Shift+F1
- Insert control character: Ctrl+P †
- Maximize window: F5
- Open file #: F3
- Optimal fill mode on/off #: Ctrl+O F
- Pair matching: Ctrl+Q [, ]
- Save file #: Ctrl+K S
- Search: Ctrl+Q F
- Search again: Ctrl+L
- Search and replace: Ctrl+Q A

* n represents a number from 0 to 9.
† Enter control characters by first pressing Ctrl+P, then pressing the desired control character.
‡ See Table A.2.
§ See Table A.3.
# See Table A.4.
Table A.1: Editing commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Alternate Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set marker #</td>
<td>$Ctrl+K n$</td>
<td></td>
</tr>
<tr>
<td>Tabs mode on/off #</td>
<td>$Ctrl+O T$</td>
<td></td>
</tr>
<tr>
<td>Topic search help</td>
<td>$Ctrl+F1$</td>
<td></td>
</tr>
<tr>
<td>Undo</td>
<td>$Alt+Backspace$</td>
<td></td>
</tr>
<tr>
<td>Unindent mode on/off #</td>
<td>$Ctrl+O U$</td>
<td></td>
</tr>
<tr>
<td>Display compiler directives</td>
<td>$Ctrl+O O$</td>
<td></td>
</tr>
</tbody>
</table>

* $n$ represents a number from 0 to 9.
† Enter control characters by first pressing $Ctrl+P$, then pressing the desired control character.
‡ See Table A.2.
§ See Table A.3.
# See Table A.4.

Table A.2: Block commands in depth

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy to Clipboard and Paste from Clipboard</td>
<td>$Ctrl+Ins$, $Shift+Ins$</td>
<td>Copies a previously selected block to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The original block is unchanged. If no block is selected, nothing happens.</td>
</tr>
<tr>
<td>Copy to Clipboard</td>
<td>$Ctrl+Ins$</td>
<td>Copies selected text to the Clipboard.</td>
</tr>
<tr>
<td>Cut to Clipboard</td>
<td>$Shift+Del$</td>
<td>Cuts selected text to the Clipboard.</td>
</tr>
<tr>
<td>Delete block</td>
<td>$Ctrl+Del$</td>
<td>Deletes a selected block. You can “undelete” a block with Undo.</td>
</tr>
<tr>
<td>Cut to Clipboard and Paste from Clipboard</td>
<td>$Shift+Del$, $Shift+Ins$</td>
<td>Moves a previously selected block from its original position to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The block disappears from its original position. If no block is selected, nothing happens.</td>
</tr>
<tr>
<td>Paste from Clipboard</td>
<td>$Shift+Ins$</td>
<td>Pastes the contents of the Clipboard.</td>
</tr>
<tr>
<td>Read block from disk</td>
<td>$Ctrl+K R$</td>
<td>Reads a disk file into the current text at the cursor position exactly as if it were a block. The text read is then selected as a block. When this command is issued, you are prompted for the name of the file to read. You can use wildcards to select a file to read; a directory is displayed. The file specified can be any legal file name.</td>
</tr>
<tr>
<td>Write block to disk</td>
<td>$Ctrl+K W$</td>
<td>Writes a selected block to a file. When you give this command, you are prompted for the name of the file to write to. The file can be given any legal name (the default extension is PAS). If you prefer to use a file name without an extension, append a period to the end of its name.</td>
</tr>
</tbody>
</table>
If you have used Borland editors in the past, you might prefer to use the block commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set beginning of block</td>
<td>Ctrl+KB</td>
<td>Begin selection of text.</td>
</tr>
<tr>
<td>Set end of block</td>
<td>Ctrl+KK</td>
<td>End selection of text.</td>
</tr>
<tr>
<td>Hide/show block</td>
<td>Ctrl+KH</td>
<td>Alternately displays and hides selected text.</td>
</tr>
<tr>
<td>Copy block</td>
<td>Ctrl+KC</td>
<td>Copies the selected text to the position of the cursor. Useful only with the Persistent Block option.</td>
</tr>
<tr>
<td>Move block</td>
<td>Ctrl+KV</td>
<td>Moves the selected text to the position of the cursor. Useful only with the Persistent Block option.</td>
</tr>
</tbody>
</table>

The next table describes certain editing commands in more detail. The table is arranged alphabetically by command name.

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoindent mode on/off</td>
<td>Ctrl+O</td>
<td>Toggles the automatic indenting of successive lines. You can also use Options</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Editor Autoindent in the IDE to turn automatic indenting on and off.</td>
</tr>
<tr>
<td>Cursor through tabs on/off</td>
<td>Ctrl+OR</td>
<td>The arrow keys will move the cursor to the middle of tabs when this option is on; otherwise the cursor jumps several columns when moving the cursor over multiple tabs. Ctrl+OR is a toggle.</td>
</tr>
<tr>
<td>Find place marker</td>
<td>Ctrl+Q</td>
<td>Finds up to ten place markers (n can be any number in the range 0 to 9) in text. Move the cursor to any previously set marker by pressing Ctrl+Q and the marker number.</td>
</tr>
<tr>
<td>Save file</td>
<td>F2</td>
<td>Saves the file and returns to the editor.</td>
</tr>
<tr>
<td>Open file</td>
<td>F3</td>
<td>Lets you load an existing file into an edit window.</td>
</tr>
<tr>
<td>Optimal fill mode on/off</td>
<td>Ctrl+OF</td>
<td>Toggles optimal fill. Optimal fill begins every line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters.</td>
</tr>
</tbody>
</table>

*n represents a number from 0 to 9.
Table A.4: Other editor commands in depth (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set marker</td>
<td>Ctrl+K n*</td>
<td>You can mark up to ten places in text. After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the Find Place Marker command (being sure to use the same marker number). You can have ten places marked in each window.</td>
</tr>
<tr>
<td>Tabs mode on/off</td>
<td>Ctrl+O T</td>
<td>Toggles Tab mode. You can specify the use of true tab characters in the IDE with the Options</td>
</tr>
<tr>
<td>Unindent mode on/off</td>
<td>Ctrl+O U</td>
<td>Toggles Unindent. You can turn Unindent on and off from the IDE with the Options</td>
</tr>
</tbody>
</table>

* n represents a number from 0 to 9.

Searching with regular expressions

You can choose to search for text using wildcards in the search string. This table lists the wildcards you can use:

Table A.5: Regular expression wildcards

<table>
<thead>
<tr>
<th>Expression</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>A circumflex at the start of the string matches the start of a line.</td>
</tr>
<tr>
<td>$</td>
<td>A dollar sign at the end of the expression matches the end of a line.</td>
</tr>
<tr>
<td>.</td>
<td>A period matches any character.</td>
</tr>
<tr>
<td>*</td>
<td>A character followed by an asterisk matches any number of occurrences (including zero) of that character. For example, bo* matches bot, b, boo, and also be.</td>
</tr>
<tr>
<td>+</td>
<td>A character followed by a plus sign matches any number of occurrences (but not zero) of that character. For example, bo+ matches bot and boo, but not be or b.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Characters in brackets match any one character that appears in the brackets but no others. For example [bot] matches b, o, or t.</td>
</tr>
<tr>
<td>[^]</td>
<td>A circumflex at the start of the string in brackets means not. Hence, [^bot] matches any character except b, o, or t.</td>
</tr>
<tr>
<td>[-]</td>
<td>A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.</td>
</tr>
<tr>
<td>\</td>
<td>A backslash before a wildcard character tells Turbo Pascal to treat that character literally, not as a wildcard. For example, ^ matches ^ and does not look for the start of a line.</td>
</tr>
</tbody>
</table>
Compiler directives quick reference

This appendix lists all of the Turbo Pascal compiler directives. It shows the syntax as you would enter it in your source code, displays the command-line equivalent, and briefly describes each directive.

Asterisks (*) indicate the default setting. For example, the default setting for debug information {$D+} is on.

Table B.1: Compiler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Source Code Syntax</th>
<th>Default</th>
<th>Command-line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align data word</td>
<td>{$A+}</td>
<td>*</td>
<td>/$A+</td>
<td>Aligns variables and typed constants on word boundaries.</td>
</tr>
<tr>
<td>Align data byte</td>
<td>{$A-}</td>
<td></td>
<td>/$A-</td>
<td>Aligns variables and typed constants on byte boundaries.</td>
</tr>
<tr>
<td>Boolean evaluation –</td>
<td>{$B+}</td>
<td></td>
<td>/$B+</td>
<td>Complete Boolean expression evaluation.</td>
</tr>
<tr>
<td>complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean evaluation –</td>
<td>{$B-}</td>
<td>*</td>
<td>/$B-</td>
<td>Short circuit Boolean expression evaluation.</td>
</tr>
<tr>
<td>short circuit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debug information</td>
<td>{$D+}</td>
<td>*</td>
<td>/$D+</td>
<td>Generates debug information.</td>
</tr>
<tr>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debug information</td>
<td>{$D-}</td>
<td></td>
<td>/$D-</td>
<td>Turns off debug information.</td>
</tr>
<tr>
<td>off</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Directive</td>
<td>Source Code Syntax</td>
<td>Default</td>
<td>Command-line</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------</td>
<td>---------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DEFINE</td>
<td>{DEFINE name}</td>
<td>/Dname</td>
<td></td>
<td>Defines a conditional symbol of name.</td>
</tr>
<tr>
<td>ELSE</td>
<td>{ELSE}</td>
<td></td>
<td></td>
<td>Switches between compiling and ignoring source delimited by $IFxxx and $ENDIF.</td>
</tr>
<tr>
<td>Emulation on</td>
<td>{$E+}</td>
<td>*</td>
<td>/$E+</td>
<td>Enables linking with a run-time library that emulates the 80x87 numeric coprocessor.</td>
</tr>
<tr>
<td>Emulation off</td>
<td>{$E-}</td>
<td></td>
<td>/$E-</td>
<td>Disables linking with a run-time library that emulates the 80x87 numeric coprocessor.</td>
</tr>
<tr>
<td>ENDF</td>
<td>{$ENDIF}</td>
<td></td>
<td></td>
<td>Ends conditional compilation started by last $IFxxx.</td>
</tr>
<tr>
<td>Extended syntax¹</td>
<td>Force far calls on</td>
<td>{$F+}</td>
<td>/$F+</td>
<td>Procedures and functions compiled always use far call model.</td>
</tr>
<tr>
<td></td>
<td>Force far calls off</td>
<td>{$F-}</td>
<td>/$F-</td>
<td>Compiler selects appropriate model: far or near.</td>
</tr>
<tr>
<td></td>
<td>80286 code generation on</td>
<td>{$G+}</td>
<td>/$G+</td>
<td>Generates 80286 instructions to improve code generation.</td>
</tr>
<tr>
<td></td>
<td>80286 code generation off</td>
<td>{$G-}</td>
<td>/$G-</td>
<td>Generates only generic 8086 instructions.</td>
</tr>
<tr>
<td></td>
<td>Input/output checking on</td>
<td>{$I+}</td>
<td>/$I+</td>
<td>Enables the automatic code generation that checks the result of a call to an I/O procedure.</td>
</tr>
<tr>
<td></td>
<td>Input/output checking off</td>
<td>{$I-}</td>
<td>/$I-</td>
<td>Disables the automatic code generation that checks the result of a call to an I/O procedure.</td>
</tr>
<tr>
<td>Include file</td>
<td>{$I filename}</td>
<td></td>
<td></td>
<td>Includes the named file in the compilation.</td>
</tr>
<tr>
<td>IFDEF</td>
<td>{IFDEF name}</td>
<td></td>
<td></td>
<td>Compiles source text that follows if name is defined.</td>
</tr>
<tr>
<td>IFNDEF</td>
<td>{IFNDEF name}</td>
<td></td>
<td></td>
<td>Compiles the source text that follows if name is not defined.</td>
</tr>
</tbody>
</table>

¹ See \$X on page 274.
<table>
<thead>
<tr>
<th>Directive</th>
<th>Source Code Syntax</th>
<th>Default</th>
<th>Command-line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFOPT</td>
<td><code>{IFOPT switch}</code></td>
<td></td>
<td></td>
<td>Compiles the source text that follows if <code>switch</code> is currently in the specified state.</td>
</tr>
<tr>
<td>Link object file</td>
<td><code>$L filename</code></td>
<td></td>
<td></td>
<td>Links the named object file with the program or unit being compiled.</td>
</tr>
<tr>
<td>Local symbol information on</td>
<td><code>$L+</code></td>
<td>*</td>
<td><code>/L+</code></td>
<td>Generates local symbol information.</td>
</tr>
<tr>
<td>Local symbol information off</td>
<td><code>$L-</code></td>
<td></td>
<td><code>/L-</code></td>
<td>Disables generation of local symbol information.</td>
</tr>
<tr>
<td>Memory allocation sizes</td>
<td><code>[SM]stacksize, heapmin,heapmax</code></td>
<td></td>
<td><code>/SMstacksize, heapmin,heapmax</code></td>
<td>Specifies an application or library's memory allocation parameters.</td>
</tr>
<tr>
<td>Numeric coprocessor on</td>
<td><code>$N+$</code></td>
<td></td>
<td><code>/N+</code></td>
<td>Generates code that performs all real-type calculations using 80x87.</td>
</tr>
<tr>
<td>Numeric coprocessor off</td>
<td><code>$N-$</code></td>
<td>*</td>
<td><code>/N-</code></td>
<td>Generates code that performs all real-type calculations by calling run-time library routines.</td>
</tr>
<tr>
<td>Open parameters on</td>
<td><code>$P+$</code></td>
<td></td>
<td><code>/P+</code></td>
<td>Enables open string and array parameters in procedure and function declarations.</td>
</tr>
<tr>
<td>Open parameters off</td>
<td><code>$P-$</code></td>
<td>*</td>
<td><code>/P-</code></td>
<td>Disables open string and array parameters.</td>
</tr>
<tr>
<td>Overflow checking on</td>
<td><code>$Q+$</code></td>
<td></td>
<td><code>/Q+</code></td>
<td>Enables the generation of overflow-checking code.</td>
</tr>
<tr>
<td>Overflow checking off</td>
<td><code>$Q-$</code></td>
<td>*</td>
<td><code>/Q-</code></td>
<td>Disables the generation of overflow-checking code.</td>
</tr>
<tr>
<td>Overlay code generation</td>
<td><code>$O+$</code></td>
<td></td>
<td><code>/O+</code></td>
<td>Enables overlay code generation.</td>
</tr>
<tr>
<td>Overlay code generation off</td>
<td><code>$O-$</code></td>
<td>*</td>
<td><code>/O-</code></td>
<td>Disables overlay code generation.</td>
</tr>
<tr>
<td>Range checking on</td>
<td><code>$R+$</code></td>
<td></td>
<td><code>/R+</code></td>
<td>Generates range-checking code.</td>
</tr>
<tr>
<td>Range checking off</td>
<td><code>$R-$</code></td>
<td>*</td>
<td><code>/R-</code></td>
<td>Disables generation of range-checking code.</td>
</tr>
<tr>
<td>Stack-overflow checking on</td>
<td><code>$S+$</code></td>
<td>*</td>
<td><code>$S+$</code></td>
<td>Generates stack-overflow checking code.</td>
</tr>
</tbody>
</table>
Table B.1: Compiler directives (continued)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Source Code Syntax</th>
<th>Default</th>
<th>Command-line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack-overflow checking off</td>
<td>{$SS-$}</td>
<td>/$SS-</td>
<td></td>
<td>Disables generation of stack-overflow code.</td>
</tr>
<tr>
<td>Type-checked pointers on</td>
<td>{$ST+}</td>
<td></td>
<td>/$ST+</td>
<td>Enables the generation of type-checked pointers when the @ operator is used.</td>
</tr>
<tr>
<td>Type-checked pointers off</td>
<td>{$ST-}</td>
<td>*</td>
<td>/$ST-</td>
<td>Disables the generation of type-checked pointers when the @ operator is used.</td>
</tr>
<tr>
<td>UNDEF</td>
<td>{UNDEF name}</td>
<td></td>
<td></td>
<td>Undefines a previously defined conditional symbol.</td>
</tr>
<tr>
<td>Var-string checking on</td>
<td>{$SV+}</td>
<td>*</td>
<td>/$SV+</td>
<td>Strict type checking is enabled.</td>
</tr>
<tr>
<td>Var-string checking off</td>
<td>{$SV-$}</td>
<td></td>
<td>/$SV-</td>
<td>Type checking is relaxed.</td>
</tr>
<tr>
<td>Extended syntax on</td>
<td>{$SX+}</td>
<td>*</td>
<td>/$SX+</td>
<td>Enables extended syntax to permit discarding result of a function call and to support null-terminated strings.</td>
</tr>
<tr>
<td>Extended syntax off</td>
<td>{$SX-}</td>
<td></td>
<td>/$SX-</td>
<td>Disables extended syntax.</td>
</tr>
</tbody>
</table>
Reserved words and standard directives

This appendix lists the Turbo Pascal reserved words and standard directives.

Reserved words and standard directives appear in lowercase boldface throughout the manuals. Turbo Pascal isn’t case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

Reserved words have a special meaning to Turbo Pascal; you can’t redefine them.

<table>
<thead>
<tr>
<th>Reserved words</th>
<th>end</th>
<th>mod</th>
<th>shl</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>file</td>
<td>nil</td>
<td>shr</td>
</tr>
<tr>
<td>asm</td>
<td>for</td>
<td>not</td>
<td>string</td>
</tr>
<tr>
<td>begin</td>
<td>function</td>
<td>object</td>
<td>then</td>
</tr>
<tr>
<td>case</td>
<td>goto</td>
<td>of</td>
<td>to</td>
</tr>
<tr>
<td>const</td>
<td>if</td>
<td>or</td>
<td>type</td>
</tr>
<tr>
<td>constructor</td>
<td>implementation</td>
<td>packed</td>
<td>unit</td>
</tr>
<tr>
<td>destructor</td>
<td>in</td>
<td>procedure</td>
<td>until</td>
</tr>
<tr>
<td>div</td>
<td>inherited</td>
<td>program</td>
<td>uses</td>
</tr>
<tr>
<td>do</td>
<td>inline</td>
<td>record</td>
<td>var</td>
</tr>
<tr>
<td>downto</td>
<td>interface</td>
<td>repeat</td>
<td>while</td>
</tr>
<tr>
<td>else</td>
<td>label</td>
<td>set</td>
<td>with</td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following are Turbo Pascal’s standard directives. Unlike reserved words, you may redefine them. It’s advised that you avoid creating user-defined identifiers with the same names as
directives because doing so can produce unexpected results and make it difficult to debug your program.

<table>
<thead>
<tr>
<th>absolute</th>
<th>far</th>
<th>near</th>
<th>resident</th>
</tr>
</thead>
<tbody>
<tr>
<td>assembler</td>
<td>forward</td>
<td>private</td>
<td>virtual</td>
</tr>
<tr>
<td>external</td>
<td>interrupt</td>
<td>public</td>
<td></td>
</tr>
</tbody>
</table>

private and public act as reserved words within object type declarations, but are otherwise treated as directives.
This appendix contains a table that lists the American Standard Code for Information Interchange (ASCII) characters. ASCII is a code that translates alphabetic and numeric characters and symbols and control instructions into 7-bit binary code. Table D.1 shows both printable characters and control characters.
### Table D.1
#### ASCII table

The caret in ^e means to press the Ctrl key and type e.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>20</td>
<td></td>
<td>64</td>
<td>40</td>
<td></td>
<td>96</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>TAB</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>LF</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>VT</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>FF</td>
<td>44</td>
<td>2C</td>
<td></td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>CR</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>SO</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>SI</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>DLE</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>EM</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>SUB</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>ESC</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>FS</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>GS</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>RS</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>US</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>_</td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>Char</td>
<td>Dec</td>
<td>Hex</td>
<td>Char</td>
<td>Dec</td>
<td>Hex</td>
<td>Char</td>
<td>Dec</td>
<td>Hex</td>
<td>Char</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>128</td>
<td>80</td>
<td>Ç</td>
<td>160</td>
<td>A0</td>
<td>à</td>
<td>192</td>
<td>C0</td>
<td>l</td>
<td>224</td>
<td>E0</td>
<td>ð</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>ü</td>
<td>161</td>
<td>A1</td>
<td>ū</td>
<td>193</td>
<td>C1</td>
<td>ǚ</td>
<td>225</td>
<td>E1</td>
<td>B</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>é</td>
<td>162</td>
<td>A2</td>
<td>ë</td>
<td>194</td>
<td>C2</td>
<td>ƙ</td>
<td>226</td>
<td>E2</td>
<td>Γ</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>ä</td>
<td>163</td>
<td>A3</td>
<td>ì</td>
<td>195</td>
<td>C3</td>
<td>ƙ</td>
<td>227</td>
<td>E3</td>
<td>n</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>à</td>
<td>164</td>
<td>A4</td>
<td>ñ</td>
<td>196</td>
<td>C4</td>
<td>ƙ</td>
<td>228</td>
<td>E4</td>
<td>ß</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>ã</td>
<td>165</td>
<td>A5</td>
<td>ñ</td>
<td>197</td>
<td>C5</td>
<td>ƙ</td>
<td>229</td>
<td>E5</td>
<td>σ</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>à</td>
<td>166</td>
<td>A6</td>
<td>à</td>
<td>198</td>
<td>C6</td>
<td>ƙ</td>
<td>230</td>
<td>E6</td>
<td>µ</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>ç</td>
<td>167</td>
<td>A7</td>
<td>ç</td>
<td>199</td>
<td>C7</td>
<td>ƙ</td>
<td>231</td>
<td>E7</td>
<td>τ</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>ñ</td>
<td>168</td>
<td>A8</td>
<td>ç</td>
<td>200</td>
<td>C8</td>
<td>ƙ</td>
<td>232</td>
<td>E8</td>
<td>Ï</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>ñ</td>
<td>169</td>
<td>A9</td>
<td>ç</td>
<td>201</td>
<td>C9</td>
<td>ƙ</td>
<td>233</td>
<td>E9</td>
<td>Ï</td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td>ñ</td>
<td>170</td>
<td>AA</td>
<td>ç</td>
<td>202</td>
<td>CA</td>
<td>ƙ</td>
<td>234</td>
<td>EA</td>
<td>Ω</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>ñ</td>
<td>171</td>
<td>AB</td>
<td>ç</td>
<td>203</td>
<td>CB</td>
<td>ƙ</td>
<td>235</td>
<td>EB</td>
<td>δ</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>ñ</td>
<td>172</td>
<td>AC</td>
<td>ç</td>
<td>204</td>
<td>CC</td>
<td>ƙ</td>
<td>236</td>
<td>EC</td>
<td>Þ</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>ñ</td>
<td>173</td>
<td>AD</td>
<td>ç</td>
<td>205</td>
<td>CD</td>
<td>ƙ</td>
<td>237</td>
<td>ED</td>
<td>Φ</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>ñ</td>
<td>174</td>
<td>AE</td>
<td>ç</td>
<td>206</td>
<td>CE</td>
<td>ƙ</td>
<td>238</td>
<td>EE</td>
<td>Ù</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>ñ</td>
<td>175</td>
<td>AF</td>
<td>ç</td>
<td>207</td>
<td>CF</td>
<td>ƙ</td>
<td>239</td>
<td>EF</td>
<td>Ù</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>ñ</td>
<td>176</td>
<td>B0</td>
<td>ç</td>
<td>208</td>
<td>D0</td>
<td>ƙ</td>
<td>240</td>
<td>F0</td>
<td>=</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>ñ</td>
<td>177</td>
<td>B1</td>
<td>ç</td>
<td>209</td>
<td>D1</td>
<td>ƙ</td>
<td>241</td>
<td>F1</td>
<td>÷</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>ñ</td>
<td>178</td>
<td>B2</td>
<td>ç</td>
<td>210</td>
<td>D2</td>
<td>ƙ</td>
<td>242</td>
<td>F2</td>
<td>÷</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>ñ</td>
<td>179</td>
<td>B3</td>
<td>ç</td>
<td>211</td>
<td>D3</td>
<td>ƙ</td>
<td>243</td>
<td>F3</td>
<td>÷</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>ñ</td>
<td>180</td>
<td>B4</td>
<td>ç</td>
<td>212</td>
<td>D4</td>
<td>ƙ</td>
<td>244</td>
<td>F4</td>
<td>÷</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>ñ</td>
<td>181</td>
<td>B5</td>
<td>ç</td>
<td>213</td>
<td>D5</td>
<td>ƙ</td>
<td>245</td>
<td>F5</td>
<td>÷</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>ñ</td>
<td>182</td>
<td>B6</td>
<td>ç</td>
<td>214</td>
<td>D6</td>
<td>ƙ</td>
<td>246</td>
<td>F6</td>
<td>÷</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>ñ</td>
<td>183</td>
<td>B7</td>
<td>ç</td>
<td>215</td>
<td>D7</td>
<td>ƙ</td>
<td>247</td>
<td>F7</td>
<td>÷</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>ñ</td>
<td>184</td>
<td>B8</td>
<td>ç</td>
<td>216</td>
<td>D8</td>
<td>ƙ</td>
<td>248</td>
<td>F8</td>
<td>÷</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>ñ</td>
<td>185</td>
<td>B9</td>
<td>ç</td>
<td>217</td>
<td>D9</td>
<td>ƙ</td>
<td>249</td>
<td>F9</td>
<td>÷</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>ñ</td>
<td>186</td>
<td>BA</td>
<td>ç</td>
<td>218</td>
<td>DA</td>
<td>ƙ</td>
<td>250</td>
<td>FA</td>
<td>÷</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>ñ</td>
<td>187</td>
<td>BB</td>
<td>ç</td>
<td>219</td>
<td>DB</td>
<td>ƙ</td>
<td>251</td>
<td>FB</td>
<td>÷</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>ñ</td>
<td>188</td>
<td>BC</td>
<td>ç</td>
<td>220</td>
<td>DC</td>
<td>ƙ</td>
<td>252</td>
<td>FC</td>
<td>÷</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>ñ</td>
<td>189</td>
<td>BD</td>
<td>ç</td>
<td>221</td>
<td>DD</td>
<td>ƙ</td>
<td>253</td>
<td>FD</td>
<td>÷</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>ñ</td>
<td>190</td>
<td>BE</td>
<td>ç</td>
<td>222</td>
<td>DE</td>
<td>ƙ</td>
<td>254</td>
<td>FE</td>
<td>÷</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>ñ</td>
<td>191</td>
<td>BF</td>
<td>ç</td>
<td>223</td>
<td>DF</td>
<td>ƙ</td>
<td>255</td>
<td>FF</td>
<td>÷</td>
</tr>
</tbody>
</table>
INDEX

80286 code generation compiler switch 215
286 Code option 215
80x87 code option 218

A
$A compiler directive 211
Abs function 6
absolute value of an argument 6
access code passed to DOS 45
Addr function 6
address
  converting to pointer 129
  of object 6
Align Data command 211
aligning data 211
AndPut constant 13
AnyFile constant 43
Append procedure 6
Arc procedure 8
ArcCoordsType type 7
Archive constant 43
ArcTan function 8
arctangent of an argument 8
argument
  arctangent of 8
  predecessor of 129
  size 176
  successor of 188
arguments
  command-line compiler 227
  number passed 59
ASCII characters 277
aspect ratio 59
  correction factor, changing 154
assembly language 216
assign buffer to text file 168
assign file name to file variable 9
Assign procedure 9
AssignCrt procedure 10
Assigned function 10
autoindent mode 267, 269

B
/B command-line option (TPC) 231
$B compiler directive 211
bar constants 11
Bar3D procedure 12
Bar procedure 11
BGI, Zenith Z-449 and 91
binary format 234
BINOBJ 137, 139
bit images 70
BitBlt operators 12, 130
BIX, Borland information 2
BkSlashFill constant 48
Black constant 22
Black text color constant 192
Blink text color constant 192
block
  copy 267, 268
    Borland-style 269
cut 268
delete 267, 268
extend 267
hide and show 266
  Borland-style 269
indent 267
move 267, 268
  Borland-style 269
to beginning of 266
to end of 266
print 266
read from disk 267, 268
set beginning of 266
  Borland-style 269
set end of 266
Borland-style 269
unindent 267
write to disk 267, 268
BlockRead procedure 13
BlockWrite procedure 14
Blue constant 22
Blue text color constant 192
Boolean evaluation
compiler switch 211
complete 211
short circuit 211
Borland, contacting 2-4
BottomText constant 99
Break procedure 15
Brown constant 22
Brown text color constant 192
buffer, assigning to text file 168
buffers, flushing 54
Build command 231
build command-line option 231
BW40 mode constant 26
BW80 mode constant 26
bytes, copying 108

C
C040 mode constant 26
C080 mode constant 26
C40 mode constant 26
C80 mode constant 26
CenterLn constant 101
CenterText constant 99
CGA 25
changing directories 15
characters
convert to uppercase 200
delete 266
filling string with 49
ordinal number of 17
ChDir procedure 15
CheckBreak variable 16
CheckEof variable 16
CheckSnow variable 16
Chr function 17
Circle procedure 17
Clear command 268
ClearDevice procedure 18
clearing the overlay buffer 116
ClearViewPort procedure 19
Clipboard
copy to 267
cut to 267
paste from 267, 268
ClipOff constant 19
ClipOn constant 19
clipping constants 19
clipping parameters 79
Close procedure 20
CloseDotFill constant 49
CloseGraph procedure 20
ClrEol procedure 21
ClrScr procedure 21
colors 67
background 60
changing text 192
constants 22
constants for SetRGBPalette 22
drawing 62
cmd-line
compiler reference 227-236
options 229-235
/B 231
/D 230
debug 234
/E 234
/F 231
/G 234
/GD 234
/GP 234
/GR 234
/I 234
/L 232
/M 231
mode 230
/O 234
/Q 232
switching directive defaults (/$) 229
/T 233
/U 234
/V 235

command-line compiler
arguments 227
compiling and linking with 227
extended syntax 228
options 227
286 code generation 228
align data 228
append debug information to EXE 235
Boolean evaluation 228
build all 230, 231
build all units 229
debug 234
debug information 228, 235
in EXE 229
define conditional symbol 229
eulation 228
EXE & TPU directory 229, 234
find error 231
find run-time error 229
force far calls 228
I/O checking 228
include directories 229, 234
link buffer 230, 232
link buffer on disk 229
list of 228
local symbols 228, 235
make 231
make modified units 229
map file 229, 234
memory sizes 228
numeric coprocessor 228
object directories 229, 234
open parameters 228
overflow checking 228
overlay code generation 228
quiet compile 229, 230, 232
range checking 228
stack checking 228
TPL & CFG directories 229, 233
type-checked pointers 228
unit directories 229, 234
var-string checking 228
command-line parameter 59, 127
commands, editor
block operations 266
cursor movement 266
insert and delete 266
compilation, conditional 223
compiler
directives 268
$A 211
$B 211
change state of 229
conditional 210
$D 212, 231
$DEFINE 212, 224, 230
$E 213
$ELSE 213
$ENDIF 213
$F 214
$G 215
$I 98, 216
$IFDEF 215
$IFNDEF 215
$IFOPT 215
$L 216, 217
$M 99, 106, 124, 218, 230
$N 218
$O 219, 220
$P 218
parameter 210
$R 220
$S 221
switch 209
$UNDEF 222, 224
$V 222
$W 214
$Y 221
error messages 237
compiling to .EXE file 231
Complete Boolean Eval option 211
CompuServe, Borland information 2
ComStr type 44
Concat function 23
concatenating strings 23
conditional
defines (command-line option) 230
symbols 224
CONFIG.SYS 121
configuration file
TPC.CFG 235
Continue procedure 23
control characters 277
insert 267
copy block 267
Borland-style 269
Copy function 24
copy to Clipboard 267
copying
  bytes 108
  substrings 24
CopyPut constant 13
Cos function 25
cosine of an argument 25
CPU symbols 225
CreateDir procedure 25
critical errors 261
Crt unit
  AssignCrt procedure 10
  ClrEol procedure 21
  ClrScr procedure 21
  Delay procedure 27
  DelLine procedure 28
  GotoXY procedure 81
  HighVideo procedure 88
  InsLine procedure 93
  KeyPressed function 99
  LowVideo procedure 105
  NormVideo procedure 111
  NoSound procedure 112
  ReadKey function 135
  Sound procedure 27
  TextBackground procedure 192
  TextColor procedure 193
  TextMode procedure 194
  WhereX function 202
  WhereY function 202
  Window procedure 203
CS register value 26
CSeg function 26
cursor position
  reading 202
  setting 81
cursor through tabs 267, 269
customer assistance 2-4
cut to Clipboard 267
Cyan constant 22
Cyan text color constant 192
data alignment 211
date and time procedures
  GetDate 63
  GetFTime 68
  GetTime 78
  SetDate 157
  SetFTime 160
  SetTime 171
DateTime type 26
Debug Information
  command 212
  option 212
debugging
  command-line option 235
  information switch 212
  options, command-line 234
  range-checking switch 220
  run-time error messages 257
  stack overflow switch 221
Dec procedure 27
DefaultFont constant 55
$DEFINE compiler directive 212, 224, 230
Delay procedure 27
delete
  block 267
  characters 266
  lines 266
  words 266
Delete procedure 27
deleting files 38
deleting substrings 27
DelLine procedure 28
DetectGraph procedure 28
devices 18
  installing drivers 94
directives 275
directories 64
  changing 15, 156
  command-line options 233
  creating 25, 108
  procedures 145
  searching 46, 51, 57
directory
  current 62, 64
  remove a 141, 145
Directory constant 43
DirectVideo variable 29
DirStr type 44
DiskFree function 30
disks, space 30
DiskSize function 30
Dispose procedure 30
DOS
  Pascal functions for 110
verify flag 79
  setting 172
Dos unit
  DiskFree function 30
  DiskSize function 30
  DosExitCode function 32
  DosVersion function 32
  EnvCount function 36
  EnvStr function 36
  Exec procedure 39
  FExpand function 42
  FindFirst procedure 51
  FindNext procedure 52
  FSearch function 57
  FSplit procedure 57
  GetCBreak procedure 61
  GetDate procedure 63
  GetEnv function 65
  GetF Attr procedure 66
  GetFTime procedure 68
  GetIntVec procedure 71
  GetTime procedure 78
  GetVerify procedure 79
  Intr procedure 97
  Keep procedure 99
  MsDos procedure 110
  PackTime procedure 125
  SetCBreak procedure 155
  SetFTime procedure 160
  SetIntVec procedure 161
  SetTime procedure 171
  SetVerify 172
  SwapVectors procedure 189
  UnpackTime procedure 200
DosError variable 31, 39, 160
DosExitCode function 32
DosVersion function 32
DottedLn constant 101
DrawPoly procedure 32
driver constants 33

drivers
  active 64
  maximum mode number 72
DS register value 35
DSeg function 35
dynamic variables, creating 74

E
/E command-line option 234
$E compiler directive 213
edit windows, moving cursor in 266
editing
  block operations 266
  deleting 268
  reading and writing 268
commands
  cursor movement 266
  insert and delete 266
EGABlack constant 23
EGABlue constant 23
EGABrown constant 23
EGACyan constant 23
EGADarkGray constant 23
EGAGreen constant 23
EGALightBlue constant 23
EGALightCyan constant 23
EGALightGray constant 23
EGALightGreen constant 23
EGALightMagenta constant 23
EGALightRed constant 23
EGAMagenta constant 23
EGAREd constant 23
EGAW hite constant 23
EGAYellow constant 23
ellipse, drawing 49
Ellipse procedure 35
elliptical sector, drawing and filling 148
$ELSE compiler directive 213
EmptyFill constant 48
emulating numeric coprocessor (8087)
  compiler switch 213
Emulation
  command 213
  option 213
end-of-file
  error messages 239
  status 36, 37, 149
end-of-line status 37, 149
$SENDIF compiler directive 213
EnvCount function 36
environment variable 66
EnvStr function 36
Eof function 36, 37
Eoln function 37
Erase procedure 38
error messages 237
  fatal 261
  searching 231
ErrorAdr variable 39
errors
  codes for graphics operations 82, 84
  messages 82
  range 220
Exclude procedure 39
EXE & TPU directory command-line option 234
.EXE files, creating 231
Exec procedure 39
exit
  codes 32
  procedures 40
  the IDE 267
ExitCode variable 41
ExitProc variable 41
Exp function 42
exponential of an argument 42
extend block 267
extended syntax 214
Extended Syntax option 214
external
  declarations 216
  procedure errors 243
ExtStr definition errors 244
ExtStr type 44

F
/F command-line option 231
$F compiler directive 214
far calls, forcing use of models in 214
fatal run-time errors 261
FAuxiliary constant 53
FCarry constant 53
fcDirectory constant 42
fcExtension constant 42
fcFileName constant 42
fcWildcards constant 42
fcXXXX constants 42
FcXXXX flag constants 48
FExpand function 42
file
  expanding file names 44
  open 267, 269
  position 45
  reading file components 135
  save 267, 269
  size of 47
  split into components 47
file attribute constants 43
file-handling procedures
  Rename 142
  Reset 142
  Rewrite 144
  Seek 149
  SetFAttr 157
  Truncate 198
file-handling string types 44
file mode constants 55
file name length constants 43
file record types 46, 195
FileExpand function 44
 FileMode variable 45
FilePos function 45
FileRec type 46
files
  access-denied error 258
  attributes 66
  closing 20
  creating new 144
  erasing 38
  .MAP 234
  .OBJ 234
    linking with 216
  opening existing 142
  record definition for 197
  text, record definition 199
  truncating 198
  untyped, variable 13, 14
FileSearch function 46
FileSize function 47
FileSplit constants 42
FileSplit function 44, 47
fill pattern constants 48
fill patterns 67
FillChar procedure 49
FillEllipse procedure 49
filling areas 53
FillPatternType type 50
FillPoly procedure 50
FillSettingsType type 51
Find Error command 231, 232
find error command-line option 231
FindFirst procedure 43, 51
FindNext procedure 52
flag constants in Dos unit 53
flag constants in WinDos unit 53
floating-point errors 262
FloodFill procedure 53
Flush procedure 54
fmClosed constant 55
fmInOut constant 55
fmInput constant 55
fmOutput constant 55
fmXXXX constants 55
font control constants 55
Font8x8 mode constant 26
Font8x8 variable 194, 195
fonts
  installing 96
  stroked 171
Force Far Calls
  command 214
  option 214
force far calls compiler switch 214
FOverflow constant 53
FParity constant 53
Frac function 55
fractions, returning 55
FreeList variable 56
FreeMem procedure 56
fsDirectory constant 44
FSearch function 57
fsFileName constant 44
FSign constant 53
fsPathName constant 44
FSplit procedure 43, 57
fsWildCards constant 44
FsXXXX constants 44
FsXXXX constants 48
functions
  discarding results 214
  extended syntax and 214
FXXXX constants 53
fXXXX constants 53
FZero constant 53
G
/G command-line option 234
$G compiler directive 215
/GD command-line option 234
GEnie, Borland information 2
GetArcCoords procedure 58
GetArgCount function 59
GetArgStr function 59
GetAspectRatio procedure 59
GetBkColor function 60
GetCBreak procedure 61
GetColor function 62
GetCurDir function 62
GetDate procedure 63
GetDefaultPalette function 63
GetDir procedure 64
GetDriverName function 64
GetEnv function 65
GetEnvVar function 66
GetFAAttr procedure 43, 66
GetFillPattern procedure 67
GetFillSettings procedure 67
GetFTime procedure 68
GetGraphMode function 69
GetImage procedure 70
GetIntVec procedure 71
GetLineSettings procedure 71
GetMaxColor function 72
GetMaxMode function 72
GetMaxX function 73
GetMaxY function 74
GetMem procedure 74
GetModeName function 75
GetModeRange procedure 75
GetPalette procedure 76
IBM 8514 and 76
GetPaletteSize function 77
GetPixel function 77
GetTextSettings procedure 78
GetTime procedure 78
<table>
<thead>
<tr>
<th>Function/Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVerify procedure</td>
</tr>
<tr>
<td>GetViewSettings procedure</td>
</tr>
<tr>
<td>GetX function</td>
</tr>
<tr>
<td>GetY function</td>
</tr>
<tr>
<td>GothicFont constant</td>
</tr>
<tr>
<td>GotoXY procedure</td>
</tr>
<tr>
<td>/GP command-line option</td>
</tr>
</tbody>
</table>

**Graph unit**

<table>
<thead>
<tr>
<th>Function/Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc procedure</td>
</tr>
<tr>
<td>Bar3D procedure</td>
</tr>
<tr>
<td>Bar procedure</td>
</tr>
<tr>
<td>Circle procedure</td>
</tr>
<tr>
<td>ClearDevice procedure</td>
</tr>
<tr>
<td>ClearViewPort procedure</td>
</tr>
<tr>
<td>CloseGraph procedure</td>
</tr>
<tr>
<td>DetectGraph procedure</td>
</tr>
<tr>
<td>DrawPoly procedure</td>
</tr>
<tr>
<td>Ellipse procedure</td>
</tr>
<tr>
<td>FillEllipse procedure</td>
</tr>
<tr>
<td>FillPoly procedure</td>
</tr>
<tr>
<td>FloodFill procedure</td>
</tr>
<tr>
<td>GetArcCoords procedure</td>
</tr>
<tr>
<td>GetAspectRatio procedure</td>
</tr>
<tr>
<td>GetBkColor function</td>
</tr>
<tr>
<td>GetColor function</td>
</tr>
<tr>
<td>GetDefaultPalette function</td>
</tr>
<tr>
<td>GetDriverName function</td>
</tr>
<tr>
<td>GetFillPattern procedure</td>
</tr>
<tr>
<td>GetFillSettings procedure</td>
</tr>
<tr>
<td>GetGraphMode function</td>
</tr>
<tr>
<td>GetImage procedure</td>
</tr>
<tr>
<td>GetLineSettings procedure</td>
</tr>
<tr>
<td>GetMaxColor function</td>
</tr>
<tr>
<td>GetMaxMode function</td>
</tr>
<tr>
<td>GetMaxX function</td>
</tr>
<tr>
<td>GetMaxY function</td>
</tr>
<tr>
<td>GetModeName function</td>
</tr>
<tr>
<td>GetModeRange procedure</td>
</tr>
<tr>
<td>GetPalette procedure</td>
</tr>
<tr>
<td>GetPaletteSize function</td>
</tr>
<tr>
<td>GetPixel function</td>
</tr>
<tr>
<td>GetTextSettings procedure</td>
</tr>
<tr>
<td>GetViewSettings procedure</td>
</tr>
<tr>
<td>GetX function</td>
</tr>
<tr>
<td>GetY function</td>
</tr>
<tr>
<td>GraphDefaults procedure</td>
</tr>
<tr>
<td>GraphError function</td>
</tr>
<tr>
<td>GraphResult function</td>
</tr>
<tr>
<td>ImageSize function</td>
</tr>
<tr>
<td>InitGraph procedure</td>
</tr>
<tr>
<td>InstallUserDriver function</td>
</tr>
<tr>
<td>InstallUserFont function</td>
</tr>
<tr>
<td>Line procedure</td>
</tr>
<tr>
<td>LineRel procedure</td>
</tr>
<tr>
<td>LineTo procedure</td>
</tr>
<tr>
<td>MoveRel procedure</td>
</tr>
<tr>
<td>MoveTo procedure</td>
</tr>
<tr>
<td>OutText procedure</td>
</tr>
<tr>
<td>OutTextXY procedure</td>
</tr>
<tr>
<td>PieSlice procedure</td>
</tr>
<tr>
<td>SetActivePage procedure</td>
</tr>
<tr>
<td>SetAspectRatio procedure</td>
</tr>
<tr>
<td>SetBkColor procedure</td>
</tr>
<tr>
<td>SetColor procedure</td>
</tr>
<tr>
<td>SetFillPattern procedure</td>
</tr>
<tr>
<td>SetFillStyle procedure</td>
</tr>
<tr>
<td>SetGraphBufSize procedure</td>
</tr>
<tr>
<td>SetGraphMode procedure</td>
</tr>
<tr>
<td>SetLineStyle procedure</td>
</tr>
<tr>
<td>SetPalette procedure</td>
</tr>
<tr>
<td>SetRGBPalette procedure</td>
</tr>
<tr>
<td>SetTextJustify procedure</td>
</tr>
<tr>
<td>SetTextStyle procedure</td>
</tr>
<tr>
<td>SetUserCharSize procedure</td>
</tr>
<tr>
<td>SetWriteMode</td>
</tr>
<tr>
<td>TextHeight function</td>
</tr>
<tr>
<td>TextWidth function</td>
</tr>
</tbody>
</table>

**Graph Defaults**

- IBM 8514 and 91
- GraphDriver variable
- GraphError function
- GraphFreeMemPtr variable
- GraphGetMemPtr variable

288 Programmer's Reference
graphics
  bit-image operations 130
  cards 28, 91
  drawing operations 100, 102, 103, 127, 136, 162
  drivers 90
  fill operations 158, 159
  mode 69, 90, 101, 102, 103
  page operations 152, 173
  palette operations 152, 155, 156, 163
  plotting operations 132
  pointer operations 110
  polygons, drawing 32
  resolution 59
  system operations 160
  text operations 113, 115, 169, 193
  video mode operations 143
  viewport operations 172
GraphMode procedure 90
GraphResult function 84
  error codes 84
Green constant 22
Green text color constant 192
grError constant 85
gFileNotFound constant 85
gInvalidDriver constant 85
gInvalidFont constant 85
gInvalidFontNum constant 85
gInvalidMode constant 85
glOError constant 85
gNoFloodMem constant 85
gNoFontMem constant 85
gNoInitGraph constant 85
gNoLoadMem constant 85
gNoScanMem constant 85
gNotDetected constant 85
grOk constant 85
grXXXX constants 85
/GS command-line option 234

H
Halt procedure 85
HatchFill constant 49
heap
  end function, pointer to 86
  error function, pointer to 86
free memory in 107
management sizes 218, 230
org function, pointer to 86
ptr function, pointer to 87
return memory to 30
HeapEnd variable 86
HeapError variable 86
HeapOrg variable 86
HeapPtr variable 87
Help 267
  index 267
  topic search 268
Hi function 87
Hidden constant 43
High function 87
high-intensity characters 88
high-order byte 87
high-order bytes
  swapping 188
HighVideo procedure 88
HorizDir constant 55

I
/I command-line option 234
$IFDEF compiler directive 98, 216
$I compiler directive 98, 216
I/O
  checking 98, 216
  error-checking 216
  errors 260
  operation, status of last 98
I/O Checking
  command 216
  option 216
IBM 8514
  GetPalette procedure and 76
  GraphDriver variable and 91
  InitGraph procedure and 91
  palette entries, modifying 164
  SetAllPalette procedure and 153
  SetPalette procedure and 163
$IFDEF compiler directive 215
$IFNDEF compiler directive 215
$IFOPT compiler directive 215
ImageSize function 88
Inc procedure 89
Include directories command-line option 216, 234

Index
Include files 216
  nesting 216
include files 234
Include procedure 90
indent block 267
InitGraph procedure 90
  SetGraphMode and 160
InOutRes variable 92
input file, name of standard 92
Input variable 92
insert
  lines 266
  mode 266
  substring into a string 93
Insert procedure 93
inserting lines 93
InsLine procedure 93
InstallUserDriver function 94
InstallUserFont function 96
Int function 97
integer part of argument 97
InterleaveFill constant 49
interrupt
  procedures 161
  vectors 71
        swapping 189
Intr procedure 97
invalid typecasting errors 245
IOResult function 98
J
justification, font 78
justification constants 99
K
Keep procedure 99
keyboard operations 99, 135
KeyPressed function 99
L
/L command-line option 232
$L compiler directive 216, 217
LastMode variable 100
LeftText constant 99
Length function 100
length of file name string 43
LightBlue constant 22
LightBlue text color constant 192
LightCyan constant 22
LightCyan text color constant 192
LightGray constant 22
LightGray text color constant 192
LightGreen constant 22
LightGreen text color constant 192
LightMagenta constant 22
LightMagenta text color constant 192
LightRed constant 22
LightRed text color constant 192
line
  drawing, setting writing mode for 174
        mark a 266
        settings 71
Line procedure 100
line style constants 101
LineFill constant 48
LineRel procedure 102
lines
  delete 266
  insert 266
LineSettingsType type 103
LineTo procedure 103
Link Buffer option 232
linking
  buffer option 232
  object files 216
Ln function 104
Lo function 104
local symbol information switch 217
Local Symbols
  command 217
  option 217
logarithm, natural 104
Low function 104
low-order bytes 104
swapping 188
LowVideo procedure 105
Lst variable 106
LtBkSlashFill constant 48
LtSlashFill constant 48
M
/M command-line option 231
$M compiler directive 99, 106, 124, 218, 230
Magenta constant 22
Magenta text color constant 192
Make command 231
make command-line option 231
map file command-line option 234
.MAP files 234
marker
  find 267, 269
  set 268, 270
MaxAvail function 106
MaxColors constant 107
MemAvail function 107
memory 74
  allocation 230
    compiler directive 218
  block, size of largest 106
  error messages 238
  free amount of 107
  freeing 30, 56
  size 218
MkDir procedure 108
mode constants, video 33
Mono mode constant 26
move block 267
Move procedure 108
MoveRel procedure 109
MoveTo procedure 110
MsDos procedure 110
MSDOS symbols 225

N
$N compiler directive 218
NameStr type 44
nesting files 216
New procedure 111
  extended syntax, constructor passed as parameter 111
NormVideo procedure 111
NormWidth constant 102
NoSound procedure 112
NotPut constant 13
null-terminated strings, compiler directive 214
number, random 132
numeric coprocessor, compiler switch 218
numeric value, convert to string 178

O
/O command-line option 234
$O compiler directive 219, 220
.OBJ files 234
  linking with 216
object
  directories, compiler directive 216
  files, linking with 216
  segment of 150
object directories command-line option 234
Odd function 112
odd number 112
offset of an object 112
Ofs function 112
online information services 2
Open a File dialog box 269
Open command 269
open file 267, 269
open string parameters compiler switch 218
Optimal Fill option 267, 269
Ord function 112
ordinal number of a character 17
ordinal value, of expression 112
OrPut constant 13
out-of-memory errors 238
output file, name of standard 113
Output variable 113
OutText procedure 113
OutTextXY procedure 115
overflow checking 219
overlay buffers
  clearing 116
  returning size 118
  setting size 123
Overlay unit
  OvrClearBuf procedure 116
  OvrGetBuf function 118
  OvrGetRetry function 118
  OvrInit procedure 120
  OvrInitEMS procedure 121
  OvrSetBuf procedure 123
  OvrSetRetry procedure 124
overlay unit name 220
overlays
  code generation, compiler switch 219
  files
    loading into EMS 121
opening 120
manager, initializing 120
Overlays Allowed
   command 219, 220
   option 219
OvrClearBuf procedure 116
OvrCodeList variable 116
OvrDebugPtr variable 117
OvrdosHandle variable 117
OvrEmsHandle variable 117
OvrFileMode variable 118
OvrGetBuf function 118
OvrGetRetry function 118
OvrHeapEnd variable 119
OvrHeapOrg variable 119
OvrHeapPtr variable 120
OvrHeapSize variable 120
OvrInit procedure 120
OvrInitEMS procedure 121
ovrIOError constant 125
OvrLoadCount variable 122
OvrLoadList variable 122
ovrNoEMSDriver constant 125
ovrNoEMSMemory constant 125
ovrNoMemory constant 125
ovrNotFound constant 125
ovrOk constant 125
OvrReadBuf variable 123
OvrResult variable 123
OvrSetBuf procedure 123
OvrSetRetry procedure 124
OvrTrapCount variable 125
ovrXXXX constants 125

P
$P compiler directive 218
PackTime procedure 125
pair matching 267
palette
color lookup table, returning size 77
definition record 63
PaletteType type 126
ParamCount function 126
parameters
   command-line 59, 127
   number of 126
number passed 59
ParamStr function 127
paste from Clipboard 267, 268
PathStr type 44
Pi function 127
PieSlice procedure 127
pixel values 77
place marker
   find 267, 269
   set 268, 270
PointType type 128
polygons, drawing 32
Pos function 128
Pred function 129
PrefixSeg variable 129
Printer unit, Lst variable 106
process-handling routines 99
Program Segment Prefix, address of 129
programs
   execution, stopping 146
   halting 85
   rebuilding 231
   Ptr function 129
PUBLIC definition errors 244
PutImage procedure 130
PutPixel procedure 132

Q
/Q command-line option 232
quiet mode command-line option 232

R
$R compiler directive 220
Random function 132
random generator
   initialize 133
   seed 133
random number 132
Randomize procedure 133
RandSeed variable 133
Range Checking
   command 220
   option 220
   range checking 229
   compiler switch 220
   Val and 201
read
  a line 136
  block 267
  file component 135
  records into a variable 13
  text file 133
Read procedure
  text files 133
  typed files 135
ReadKey function 135
Readln procedure 136
ReadOnly constant 43
Rectangle procedure 136
Red constant 22
Red text color constant 192
referencing errors 252
RegisterBGldriver function 137
RegisterBGIfont function 139
registers, CS 26
Registers type 141
regular expressions
  searching 270
  wildcards 270
relaxed string parameter checking 222
relocatable reference errors 244
RemoveDir procedure 141
Rename procedure 142
Reset procedure 142
resolution, graphics 59
RestoreCrtMode procedure 143
results, discarding 214
return flags for FileSplit constants 42
Rewrite procedure 144
RightText constant 99
RmDir procedure 145
Round function 145
run-time errors 257
  address of 39
  fatal 261
  Find Error command and 231
  finding 231
  generating 146
RunError procedure 146

$S$ compiler directive 221
SansSerifFont constant 55

save file 267, 269
SaveInt00 variable 146
SaveInt02 variable 146
SaveInt21 variable 146
SaveInt23 variable 146
SaveInt24 variable 146
SaveInt34 variable 146
SaveInt35 variable 146
SaveInt36 variable 146
SaveInt37 variable 146
SaveInt38 variable 146
SaveInt39 variable 146
SaveInt75 variable 147
SaveInt3A variable 146
SaveInt1B variable 146
SaveInt3B variable 147
SaveInt3C variable 147
SaveInt3D variable 147
SaveInt3E variable 147
SaveInt3F variable 147
SaveIntXX variables 146
search for text 267
searching
  directories 51
  run-time error messages 231
SearchRec type 147
Sector procedure 148
Seek procedure 149
SeekEof function 149
SeekEoln function 149
Seg0040 variable 150
Seg function 150
SegA000 variable 150
SegB000 variable 151
SegB800 variable 151
segment of object 150
SelectorInc variable 151
SetActivePage procedure 152
SetAllPalette procedure 152
IBM 8514 and 153
SetColor procedure 156
[...]

Index 293
SetFillPattern procedure 158
SetFillStyle procedure 159
SetFTime procedure 160
SetGraphBufSize procedure 160
SetGraphMode procedure 160
SetIntVec procedure 161
SetLineStyle procedure 162
SetPalette procedure 163
IBM 8514 and 163
SetRGBPalette procedure 164
SetTextBuf procedure 168
SetTextJustify procedure 169
SetTextStyle procedure 169
  OutText and 113
  OutTextXY and 115
SetTime procedure 171
SetUserCharSize procedure 171
SetVerify procedure 172
SetViewport procedure 172
SetVisualPage procedure 173
SetWriteMode procedure 174
short-circuit Boolean evaluation 211
Sin function 175
sine of argument 175
size
  free memory in heap 107
  largest free block in heap 106
  of argument 176
SizeOf function 49, 176
SlashFill constant 48
SmallFont constant 55
software interrupts 97
SolidFill constant 48
SolidLn constant 101
sound operations
  NoSound 112
  Sound 176
Sound procedure 176
source debugging compiler switch 212
SP register, value of 177
SPtr function 177
Sqr function 177
Sqrt function 177
square of argument 177
square root of argument 177
SS register, value of 177
SSeg function 177
stack
  checking switch directive 221
  overflow, switch directive 221
  size 218
Stack Checking
  command 221
  option 221
Str procedure 178
StrCat function 178
StrComp function 179
StrCopy function 179
StrDispose function 180
StrECopy function 180
StrEnd function 181
StrIComp function 181
strict string parameter checking 222
string
  convert to 178
  convert to number 201
  first occurrence of 186
  first occurrence of character in 187
  last occurrence of character in 187
  length of 100, 183
  point to end of 181
String Var Checking
  command 222
  option 222
strings
  allocating on heap 185
  appending 178, 181
  comparing 179
    up to maximum length 182
    without case sensitivity 181, 183
  comparing up to maximum length 183
  concatenation of 23
  converting null-terminated 185
  converting to lowercase 184
  converting to uppercase 188
  copying 179, 180, 186
  copying characters 182, 184
  deleting 27
  disposing of 180
  initializing 49
  length byte 49
  null-terminated 214
  relaxed parameter checking of 222
  strict parameter checking of 222
StrLCat function 181
StrLComp function 182
StrLCopy function 182
StrLen function 183
StrLIComp function 183
StrLower function 184
StrMove function 184
StrNew function 185
stroked fonts 171
StrPas function 185
StrPCopy function 186
StrPos function 186
StrRScan function 187
StrScan function 187
StrUpper function 188
subdirectory, create a 108
substrings
  copying 24
  deleting 27
  inserting 93
  position of 128
Succ function 188
support, technical 2-4
Swap function 188
swapping bytes 188
SwapVectors procedure 189
switch compiler directives 209
symbol reference, compiler switch 221
symbols
  conditional 224
  CPU 225
    local information 217
syntax, extended 214
SysFile constant 43

T
/T command-line option 233
Tab mode 270
Tabs mode 268
TDateTime type in WinDos unit 189
technical support 2-4
terminating a program 40
  cause of 41
    with exit procedure 41
terminating program 85
Test8086 type in System unit 190
Test8087 variable 190
text attribute variable 191
text attributes 78
text color constants 192
text file, reading from 133
text file record definition 199
text files 36
Textarea variable
  ClrEol and 21
  ClrScr and 21
  HighVideo and 88
  LowVideo and 105
  NormVideo and 111
  TextBackground and 192
  TextColor and 193
Textarea background procedure 192
Textarea color procedure 193
Textarea height function 193
Textarea mode procedure 194
Textarea rec type 46, 195
Textarea settings type 196
Textarea width function 196
TFileRec file record type in WinDos unit 197
ThickWidth constant 102
time procedures
  GetFTime 68
  GetTime 78
  SetFTime 160
  SetTime 171
Topic search in Help 268
TopOff constant 11
TopOn constant 11
TopText constant 99
TPC.CFG file 233, 235
  sample 236
TPL & CFG directory command-line option 233
trapping I/O errors 216
TRegisters type 198
TriplexFont constant 55
Trunc function 198
truncate a value 198
Truncate procedure 198
TSearchRec type 199
TTxtRec file record type 199
TURBO.TPL 233, 234
type checking, strings and 222
typecasting, invalid 245
TypeOf function 200

Index
types
  file record 46, 195
  mismatches, error messages 241

U
/U command-line option 234
$UNDEF compiler directive 222, 224
undo 268
unindent
  block 267
  mode 268, 270
Unit Directories option 234
units, version mismatch errors 246
UnpackTime procedure 200
untyped files, variable 13, 14
UpCase function 200
UserBitLn constant 102
UserCharSize constant 55
UserFill constant 49

V
/V command-line option 235
$V compiler directive 222
Val procedure 201
var, string checking, compiler switch 222
variable
  decrementing a 27
  environment 66
variables
  disposing of 30, 56
  DosError 39, 52, 66, 68, 160
  dynamic, creating 74, 111
  incrementing 89
  untyped file 13, 14
VER70 symbol 225
VertDir constant 55
VGA
  driver, modifying palette entries for 164
    modes, emulated 91
VGAHi 69
VGALo 69
VGAMed 69
video mode constants 33
video operations
  AssignCrt 10
  ClrEol 21

ClrScr 21
DelLine procedure 28
GotoXY 81
HighVideo 88
InsLine 93
LowVideo 105
NormVideo 111
RestoreCrtMode 143
TextColor 193
WhereX 202
WhereY 202
Window 203
Write (text) 204
Write (typed) 206
Writeln 206
viewports 19
  parameter 79
ViewPortType type 202
VMT, pointer to a 200
VolumeID constant 43

W
WhereX function 202
WhereY function 202
White constant 22
White text color constant 192
WideDotFill constant 49
WindMax and WindMin variables 203
WinDos unit
  DiskFree function 30
  DiskSize function 30
  GetCBreak procedure 61
  GetDate procedure 63
  GetFAttr procedure 66
  GetFTime procedure 68
  GetIntVec procedure 71
  GetTime procedure 78
  GetVerify procedure 79
  Intr procedure 97
  MsDos procedure 110
  PackTime procedure 125
  SetCBreak procedure 155
  SetFTime procedure 160
  SetIntVec procedure 161
  SetTime procedure 171
  SetVerify 172
UnpackTime procedure 200
Window procedure 203
word
delete 266
mark 266
write block 267
Write procedure
text files 204
typed files 206
write records from a variable 14
Writeln procedure 206

X
$X compiler directive 214
XHatchFill constant 49
XORPut constant 13

Y
$Y compiler directive 221
Yellow constant 22
Yellow text color constant 192

Z
Zenith Z-449, BGI and 91