7.0

TURBO PASCAL®

PROGRAMMER'S REFERENCE

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

BORLAND

Turbo Pascal®

Version 7.0

Programmer's Reference

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.

R1

C O N T E N T S

T A B L E S

I N T R O D U C T I O N

See the User's Guide for an overview of the entire Turbo Pascal documentation set. Read its introduction to find out how to use the Turbo Pascal manuals most effectively.

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?

See the Language Guide for an overview of the units found in Turbo Pascal's run-time library.

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.

Online information services

Service	Command
CompuServe	`GO BORLAND`
BIX	`JOIN BORLAND`
GEnie	`BORLAND`

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

408-461-9144
6 a.m. to 5 p.m. PT

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

900-555-1001
6 a.m. to 5 p.m. PT

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

408-461-9000 (voice)
7 a.m. to 5 p.m. PT

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.

C H A P T E R

1

Library reference

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 — Unit it occupies

Purpose Description of purpose.

Declaration 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.

Remarks Specific information about this entry.

Restrictions Special requirements that relate to this entry.

See also Related *variables*, *constants*, *types*, *procedures*, and *functions* that are also described in this chapter.

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

Abs function System

Purpose Returns the absolute value of the argument.

Declaration `function Abs(X);`

Remarks *X* is an integer-type or real-type expression. The result, of the same type as *X*, is the absolute value of *X*.

Example

```
var
  r: Real;
  i: Integer;
begin
  r := Abs(-2.3);                                                    { 2.3 }
  i := Abs(-157);                                                    { 157 }
end.
```

Addr function System

Purpose Returns the address of a specified object.

Declaration `function Addr(X): Pointer;`

Remarks *X* is any variable, or a procedure or function identifier. The result is a pointer that points to X. Like **nil**, the result of *Addr* is assignment compatible with all pointer types.

See also *Ofs*, *Ptr*, *Seg*

Example

```
var P: Pointer;
begin
  P := Addr(P);                                       { Now points to itself }
end.
```

Append procedure System

Purpose Opens an existing file for appending.

Declaration `procedure Append(var F: Text);`

Remarks *F* is a text file variable that must have been associated with an external file using *Assign*.

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

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 — Graph

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 — System

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*

Example

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

Assign procedure — System

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

Declaration

```
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:

```
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

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

AssignCrt procedure Crt

Purpose Associates a text file with the CRT.

Declaration `procedure AssignCrt(var F: Text);`

Remarks *AssignCrt* works exactly like the *Assign* 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).

Example

```
uses Crt;
var F: Text;
begin
  Write('Output to screen or printer [S, P]? ');
  if UpCase(ReadKey) = 'P' then
    Assign(F, 'PRN')                                      { Output to printer }
  else
    AssignCrt(F);                     { Output to screen, use fast CRT routines }
  Rewrite(F);
  Writeln(F, 'Fast output via CRT routines...');
  Close(F);
end.
```

Assigned function System

Purpose Tests to determine if a pointer or procedural variable is **nil**.

Declaration `function Assigned(var P): Boolean;`

Remarks *P* must be a variable reference of a pointer or procedural type. *Assigned* returns *True* if *P* is not **nil**, or *False* if **nil**. *Assigned(P)* corresponds to the test *P* <> **nil** for a pointer variable, and @*P* <> **nil** for a procedural variable.

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 — Graph

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.

Constant	Value
TopOn	*True*
TopOff	*False*

See also *Bar3D*

Bar procedure — Graph

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

Declaration

```
procedure Bar(X1, Y1, X2, Y2: 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 — Graph

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 — Graph

Purpose BitBlt operators used with *PutImage* and *SetWriteMode.*

Remarks The following constant values represent the indicated logical operations.

Constant	Value
CopyPut	0 (**mov**)
XORPut	1 (**xor**)
These BitBlt constants are used by PutImage *only*:	
OrPut	2 (**or**)
AndPut	3 (**and**)
NotPut	4 (**not**)

BlockRead procedure — System

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

```
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 — System

Purpose Writes one or more records from a variable.

Declaration

```
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 *Result* records as an effect of the *BlockWrite*.

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

Restrictions File must be open.

See also *BlockRead*

Example See example for *BlockRead*.

Break procedure — System

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

Declaration `procedure Break;`

Remarks *Break* exits the innermost enclosing **for**, **while**, or **repeat** statement immediately. *Break* is analogous to a **goto** statement addressing a label just after the end of the innermost enclosing repetitive statement. The compiler reports an error if *Break* is not enclosed by a **for**, **while**, or **repeat** statement.

See also *Continue*, *Exit*, *Halt*

Example

```
var S: string;
begin
  while True do
  begin
    Readln(S);
    if S = '' then Break;
    Writeln(S);
  end;
end.
```

ChDir procedure — System

Purpose Changes the current directory.

Declaration `procedure ChDir(S: String);`

Remarks The string-type expression changes the current directory to the path specified by *S*. If *S* specifies a drive letter, the current drive is also changed.

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

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

Example

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

CheckBreak variable — Crt

Purpose Enables and disables checks for *Ctrl+Break*.

Declaration `var CheckBreak: Boolean;`

Remarks When *CheckBreak* is *True*, pressing *Ctrl+Break* aborts the program when it next writes to the display. When *CheckBreak* is *False*, pressing *Ctrl+Break*, has no effect. *CheckBreak* is *True* by default. (At run time, *Crt* stores the old *Ctrl+Break* interrupt vector, $1B, in a global pointer variable called *SaveInt1B*.)

See also *KeyPressed*, *ReadKey*, *SaveInt1B*

CheckEOF variable — Crt

Purpose Enables and disables the end-of-file character.

Declaration `var CheckEOF: Boolean;`

Remarks When *CheckEOF* is *True*, an end-of-file character is generated if you press *Ctrl+Z* while reading from a file assigned to the screen. When *CheckEOF* is *False*, pressing *Ctrl+Z* has no effect. *CheckEOF* is *False* by default.

CheckSnow variable — Crt

Purpose Enables and disables "snow-checking" on CGA video adapters.

Declaration `var CheckSnow: Boolean;`

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 — System

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 — Graph

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*

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 — Graph

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 — Graph

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 — Graph

Purpose Constants that control clipping; used with *SetViewPort*.

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

Constant	Value
ClipOn	*True*
ClipOff	*False*

See also *SetViewPort*

Close procedure — System

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');                                    { Open file }
  Reset(F, 1);
  Writeln('File size = ', FileSize(F));
  Close(F);                                                     { Close file }
end.
```

CloseGraph procedure — Graph

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);
  Readln;
  CloseGraph;                                          { Shut down graphics }
end.
```

ClrEol procedure — Crt

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 — Crt

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 is window-relative. The following program lines define a text window and clear a 60×20 rectangle beginning at (1, 1).

```
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 — Graph

Purpose Color constants used by *SetPalette* and *SetAllPalette.*

Remarks

Constant	Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
LightGray	7
DarkGray	8
LightBlue	9
LightGreen	10
LightCyan	11
LightRed	12
LightMagenta	13
Yellow	14
White	15

See also *SetAllPalette, SetPalette, SetColor*

Color constants for SetRGBPalette — Graph

Purpose 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:

Constant	Value
EGABlack	0 (dark colors)
EGABlue	1
EGAGreen	2
EGACyan	3
EGARed	4
EGAMagenta	5
EGABrown	20
EGALightGray	7
EGADarkGray	56 (light colors)
EGALightBlue	57
EGALightGreen	58
EGALightCyan	59
EGALightRed	60
EGALightMagenta	61
EGAYellow	62
EGAWhite	63

See also *SetRGBPalette*

Concat function — System

Purpose Concatenates a sequence of strings.

Declaration `function Concat(`S_1` [ , `S_2`, ..., `S_N` ]: 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

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

Continue procedure — System

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

Declaration `procedure Continue;`

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

```
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 — System

Purpose Returns a substring of a string.

Declaration `function Copy(S: String; Index: Integer; Count: Integer): String;`

Remarks *S* is a string-type expression. *Index* and *Count* are integer-type expressions. *Copy* returns a string containing *Count* characters starting with the *Index*th 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 *Index*th position, only the remainder of the string is returned.

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

Example

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

Cos function — System

Purpose Returns the cosine of the argument.

Declaration `function Cos(X: Real): Real;`

Remarks *X* is a real-type expression. The result is the cosine of *X* where *X* represents an angle in radians.

See also *ArcTan, Sin*

Example

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

CreateDir procedure — WinDos

Purpose Creates a new subdirectory.

Declaration `procedure CreateDir(Dir: PChar);`

Remarks The subdirectory to be created is specified in *Dir*. Errors are reported in *DosError*. *MkDir* performs the same function as *CreateDir*, but it takes a Pascal-style string as an argument rather than a null-terminated string.

See also *GetCurDir, SetCurDir, RemoveDir*

Crt mode constants — Crt

Purpose Used to represent Crt text and line modes.

Remarks *BW40, C040, BW80,* and *C080* represent the four color text modes supported by the IBM PC Color/Graphics Adapter (CGA). The *Mono* constant represents the single black-and-white text mode supported by the IBM PC Monochrome Adapter. *Font8x8* represents EGA/VGA 43- and 50-line modes and is used with *C080* or *LastMode*. *LastMode* returns to the last active text mode after using graphics.

Constant	Value	Description
BW40	0	40x25 B/W on color adapter
BW80	2	80x25 B/W on color adapter
Mono	7	80x25 B/W on monochrome adapter
C040	1	40x25 color on color adapter
C080	3	80x25 color on color adapter
Font8x8	256	For EGA/VGA 43 and 50 line
C40	C040	For Turbo Pascal 3.0 compatibility
C80	C080	For Turbo Pascal3.0 compatibility

See also *TextMode*

CSeg function — System

Purpose Returns the current value of the CS register.

Declaration
```
function CSeg: Word;
```

Remarks The result of type *Word* is the segment address of the code segment within which *CSeg* was called.

See also *DSeg, SSeg*

DateTime type — Dos

Purpose *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.

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

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 *FindFirst, FindNext, GetFTime, SetFTime*

Dec procedure — System

Purpose Decrements a variable.

Declaration
```
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
```
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Dec(IntVar);                                        { IntVar := IntVar - 1 }
  Dec(LongintVar, 5);                        { LongintVar := LongintVar - 5 }
end.
```

Delay procedure — Crt

Purpose Delays a specified number of milliseconds.

Declaration
```
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 — System

Purpose Deletes a substring from a string.

Declaration
```
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* specifies more characters than remain starting at the *Index*th position, the remainder of the string is deleted.

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

DelLine procedure — Crt

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 — Graph

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.

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

```
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 — Crt

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, Writes* and *Writelns* 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
```
uses Dos;                                                      { or WinDos }
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
```
uses Dos;                                                      { or WinDos }
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 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

```
type Str18 = string[18];
var P: ^Str18;
begin
  New(P);
  P^ := 'Now you see it...';
  Dispose(P);                                                { Now you don't... }
end.
```

DosError variable — Dos, WinDos

Purpose Used by many *Dos* and *WinDos* routines to report errors.

Declaration `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:

DOS error code	Meaning
2	File not found
3	Path not found
5	Access denied
6	Invalid file handle
8	Not enough memory
10	Invalid environment
11	Invalid format
18	No more files

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 — Dos

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 — Dos, WinDos

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;                                                  { or WinDos }
var Ver: Word;
begin
  Ver := DosVersion;
  Writeln('This is DOS version ', Lo(Ver), '.', Hi(Ver));
end.
```

See also *Hi, Lo*

DrawPoly procedure — Graph

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 uses the current line style and color. Use *SetWriteMode* to determine whether the polygon is copied to or **XOR**ed to the screen.

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

PolyPoints[*N* + 1] = *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

```
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 — Graph

Purpose Used with routines that call graphics drivers and color palettes.

Remarks The following tables list graphics drivers and color palettes.

Table 1.1 Graph unit driver constants

Driver Constant	Value	Meaning
Detect	0	Requests autodetection
CGA	1	CGA mode
MCGA	2	MCGA mode
EGA	3	EGA mode
EGA64	4	EGA64 mode
EGAMono	5	EGAMono mode
IBM8514	6	IBM8514 mode
HercMono	7	HercMono mode

Table 1.1: Graph unit driver constants (continued)

ATT400	8	ATT400 mode
VGA	9	VGA mode
PC3270	10	PC3270 mode
CurrentDriver	–128	Passed to *GetModeRange*

Table 1.2: Graph unit mode constants

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

See also *DetectGraph, GetModeRange, InitGraph*

DSeg function — System

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 — Graph

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;
   YRadius, YRadius: Word);
```

Remarks Draws an elliptical arc in the current color using (*X, Y*) as a center point, and *XRadius* and *YRadius* 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 Dos

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

Declaration `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

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

EnvStr function Dos

Purpose Returns a specified environment string.

Declaration `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) System

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

Declaration `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 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 *Eoln*, *SeekEof*

Example

```
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);                                          { Dump text file }
  end;
end.
```

Eof function (typed, untyped files) — System

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

Declaration `function Eof(var F): Boolean;`

Remarks *F* is a file variable. *Eof*(*F*) returns *True* if the current file position is beyond the last component 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.

Eoln function — System

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

Declaration `function Eoln [(var F: Text)]: Boolean;`

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

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

See also *Eof, SeekEoln*

Erase procedure — System

Purpose Erases an external file.

Declaration `procedure Erase(var F);`

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 — System

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

Declaration `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 — System

Purpose Excludes an element from a set.

Declaration `procedure Exclude(var S: set of T; I: T);`

Remarks *S* is a set type variable, and *I* is an expression of a type compatible with the base type of *S*. The element given by *I* is excluded 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 — Dos

Purpose Executes a specified program with a specified command line.

Declaration `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');
```

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

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

Exit procedure — System

Purpose Exits immediately from the current block.

Declaration `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.

See also *Halt*

Example

```
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 *ErrorAddr*, *ExitProc*, and Chapter 20, "Control issues," in the *Language Guide* for more information about exit procedures.

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 *ErrorAddr*, *ExitCode*, and Chapter 20, "Control issues," in the *Language Guide* for more information about exit procedures.

Exp function — System

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 — WinDos

Purpose Return flags used by the function *FileSplit*.

Remarks The following *fxXXXX* constants are defined:

Constant	Value
fcExtension	$0001
fcFileName	$0002
fcDirectory	$0004
fcWildcards	$0008

See also *FileSplit*

FExpand function — Dos

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('..\*.TPU')          = 'C:\SOURCE\*.TPU'
FExpand('c:\bin\turbo.exe') = 'C:\BIN\TURBO.EXE'
```

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.

Dos Constant	WinDos Constant	Value
ReadOnly	*faReadOnly*	$01
Hidden	*faHidden*	$02
SysFile	*faSysFile*	$04
VolumeID	*faVolumeID*	$08
Directory	*faDirectory*	$10
Archive	*faArchive*	$20
AnyFile	*faAnyFile*	$3F

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

Constant	Value
fsPathName	79
fsDirectory	67
fsFileName	8
fsExtension	4

See also *FileSearch, FileSplit*

FileExpand function WinDos

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

Declaration `function FileExpand(Dest, Name: PChar): PChar;`

Remarks 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.

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 Dos

Purpose String types are used by various procedures and functions in the *Dos* unit.

Remarks The following string types are defined:

```
ComStr  = string[127];                    { Command-line string }
PathStr = string[79];                   { Full file path string }
DirStr  = string[67];               { Drive and directory string }
NameStr = string[8];                          { File-name string }
ExtStr  = string[4];                     { File-extension string }
```

See also *FExpand, FSplit*

FileMode variable — System

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 — System

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 — Dos

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 — WinDos

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

```
uses WinDos;
var
  S: array[0..fsPathName] of Char;
begin
  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 — System

Purpose Returns the current size of a file.

Declaration `function FileSize(var F): Longint;`

Remarks *F* is a file variable. *FileSize*(*F*) returns the number of components in *F*. If the file is empty, *FileSize*(*F*) returns 0.

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 *FilePos*

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 — WinDos

Purpose Splits a file name into its three components.

Declaration `function FileSplit(Path, Dir, Name, Ext: PChar): Word;`

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 — Graph

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.

Constant	Value	Description
EmptyFill	0	Fills area in background color
SolidFill	1	Fills area in solid fill color
LineFill	2	—— fill
LtSlashFill	3	/// fill
SlashFill	4	/// fill with thick lines
BkSlashFill	5	\\\ fill with thick lines
LtBkSlashFill	6	\\\ fill

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 — System

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 — Graph

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.

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

Example

```
uses Graph;
const R = 30;
var
  Driver, Mode: Integer;
  Xasp, Yasp: Word;
begin
  Driver := Detect;                                    { Put in graphics mode }
  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 — Graph

Purpose Record that defines a user-defined fill pattern.

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

See also *Fill pattern constants, GetFillPattern, SetFillPattern*

FillPoly procedure — Graph

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

Declaration `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

```
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 type — Graph

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

Declaration

```
type
  FillSettingsType = record
     Pattern: Word;
     Color: Word;
  end;
```

See also *GetFillSettings*

FindFirst procedure — Dos, WinDos

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

Declaration

```
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.*

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;                                                      { or WinDos }
var DirInfo: SearchRec;                                    { or TSearchRec }
begin                                                       { or faArchive }
  FindFirst('*.PAS', Archive, DirInfo);               { Same as DIR *.PAS }
  while DosError = 0 do
  begin
    Writeln(DirInfo.Name);
    FindNext(DirInfo);
  end;
end.
```

FindNext procedure — Dos, WinDos

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

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 *Dos Error, File attribute, FExpand, FileExpand, FindFirst, SearchRec, TSearchRec*

Example See the example for *FindFirst*.

Flag constants — Dos, WinDos

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

Remarks

Constants	Value
FCarry	$0001
FParity	$0004
FAuxiliary	$0010
FZero	$0040
FSign	$0080
FOverflow	$0800

For instance, if *R* is a register record, the tests

```
R.Flags and FCarry <> 0
R.Flags and FZero = 0
```

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

See also *Intr, MsDos*

FloodFill procedure — Graph

Purpose Fills a bounded region with the current fill pattern.

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

Remarks Fills an enclosed area on bitmap devices. (*X, Y*) is a seed within the enclosed area to be filled. The current fill 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:

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

```
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 — System

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:

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

See also *TextRec, TFileRec*

Font control constants — Graph

Purpose Constants that identify fonts.

Remarks

Constant	Value
DefaultFont	0 (8x8 bit mapped font)
TriplexFont	1 ("stroked" fonts)
SmallFont	2
SansSerifFont	3
GothicFont	4
HorizDir	0 (left to right)
VertDir	1 (bottom to top)
UserCharSize	0 (user-defined Char size)

See also *GetTextSettings, SetTextStyle, TextSettingsType*

Frac function — System

Purpose Returns the fractional part of the argument.

Declaration `function Frac(X: Real): Real;`

Remarks *X* is a real-type expression. The result is the fractional part of *X*; that is, *Frac*(*X*) = *X* – *Int*(*X*).

See also *Int*

Example

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

FreeList variable — System

Purpose Points to the first free block in the heap.

Declaration `var FreeList: Pointer;`

Remarks The *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, *FreeList* will be equal to *HeapPtr*. See Chapter 12, "Standard procedures and functions," in the *Language Guide* for more information.

See also *Dispose, FreeMem, HeapPtr*

FreeMem procedure — System

Purpose Disposes of a dynamic variable of a given size.

Declaration `procedure FreeMem(var P: Pointer; Size: Word);`

Remarks *P* is a variable of any pointer type previously assigned by the *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 *exactly* the number of bytes previously allocated to that variable by *GetMem*. *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 *FreeMem*, the value of *P* becomes undefined, and an error occurs if you subsequently reference *P*^.

See also *Dispose, FreeMem, HeapError, New*

FSearch function Dos

Purpose Searches for a file in a list of directories.

Declaration
```
function FSearch(Path: PathStr; 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: PathStr;
begin
  S := FSearch('TURBO.EXE', GetEnv('PATH'));
  if S = '' then
    Writeln('TURBO.EXE not found')
  else
    Writeln('Found as ', FExpand(S));
end.
```

FSplit procedure Dos

Purpose Splits a file name into its three components.

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

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 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;
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;
var
  Gd, Gm: Integer;
  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 WinDos

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 WinDos

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 Graph

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);`

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 — Graph

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                                              { Cycle through colors }
      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 — Dos, WinDos

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 — Graph

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

Declaration `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

```
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 — WinDos

Purpose Returns the current directory of a specified drive.

Declaration `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.

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 — Dos, WinDos

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 — Graph

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;                                    { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  GetDefaultPalette(OldPal);                              { Preserve old one }
  MyPal := OldPal;                                     { Duplicate and modify }
  { Display something }
  for I := 0 to MyPal.Size - 1 do
  begin
    SetColor(I);
    OutTextXY(10, I * 10, '...Press any key...');
  end;
  repeat                                    { Change palette until a key is pressed }
    with MyPal do
      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 — System

Purpose Returns the current directory of a specified drive.

Declaration `procedure GetDir(D: Byte; var S: String);`

Remarks *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.

See also *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.

GetDriverName function — Graph

Purpose Returns a string containing the name of the current driver.

Declaration `function GetDriverName: String;`

Remarks After a call to *InitGraph*, returns the name of the active driver.

Restrictions Must be in graphics mode.

See also *GetModeName, InitGraph*

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 — WinDos

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 — Dos, WinDos

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 **and**ing 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));
  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 — Graph

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

Declaration `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 — Graph

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

Declaration `procedure GetFillSettings(var FillInfo: 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 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
    SetFillStyle(Pattern, Color);                         { Restore old fill style }
  Bar(100, 0, 150, 50);
  Readln;
  CloseGraph;
end.
```

GetFTime procedure — Dos, WinDos

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 — Graph

Purpose Returns the current graphics mode.

Declaration `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:

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

PC3270	PC3270Hi	0	720x350	2 color	1
IBM8514	IBM8514Lo	0	640x480	256 color	1
IBM8514	IBM8514Hi	0	1024x768	256 color	1

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

Restrictions Must be in graphics mode.

See also *ClearDevice, DetectGraph, InitGraph, RestoreCrtMode, SetGraphMode*

Example

```
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 — Graph

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

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

Remarks *X1, Y1, X2,* and *Y2* 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*.

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 — Dos, WinDos

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 — Graph

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.

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, 0, ThickWidth);                        { New style }
  Line(0, 10, 100, 10);
  with OldStyle do                                    { Restore old line style }
    SetLineStyle(LineStyle, Pattern, Thickness);
  Line(0, 20, 100, 20);
  Readln;
  CloseGraph;
end.
```

GetMaxColor function — Graph

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 — Graph

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;                                   { 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.
```

GetMaxX function — Graph

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 320×200 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(1);
  Rectangle(0, 0, GetMaxX, GetMaxY);                    { Draw a full-screen box }
  Readln;
  CloseGraph;
end.
```

GetMaxY function — Graph

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 320×200 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);                    { Draw a full-screen box }
  Readln;
  CloseGraph;
end.
```

GetMem procedure — System

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*.)

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 Graph

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

```
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 Graph

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:

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

```
    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 — Graph

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 — Graph

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 — Graph

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 — Graph

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 — Dos, WinDos

Purpose Returns the current time set in the operating system.

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

Remarks Ranges of the values returned are *Hour* 0..23, *Minute* 0..59, *Second* 0..59, and *Sec*100 (hundredths of seconds) 0..99.

See also *GetDate, SetDate, SetTime, UnpackTime*

GetVerify procedure — Dos, WinDos

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 — Graph

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

GetX function Graph

Purpose Returns the *X* coordinate of the current position (CP).

Declaration `function GetX: 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 *GetX* returns a value of 0.
- `MoveTo(5, 5);`
  Moves CP to absolute (5, 5), and *GetX* returns a value of 5.
- `SetViewPort(10, 10, 100, 100, True);`
  Moves CP to absolute (10, 10), but *GetX* returns a value of 0.
- `MoveTo(5, 5);`
  Moves CP to absolute (15, 15), but *GetX* returns a value of 5.

Restrictions Must be in graphics mode.

See also *GetViewSettings, GetY, 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.
```

GetY function — Graph

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 — Crt

Purpose Moves the cursor to the given coordinates.

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

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)):

```
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 — Graph

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 — Graph

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").

See also *DetectGraph, GraphResult, InitGraph*

Example

```
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 — Graph

Purpose Holds the address of the heap deallocation routine.

Declaration `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 — Graph

Purpose Holds the address of the heap allocation routine.

Declaration `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 — Graph

Purpose Returns an error code for the last graphics operation.

Declaration `function GraphResult: Integer;`

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

The following routines set *GraphResult*:

Bar	*GetGraphMode*	*SetAllPalette*
Bar3D	*ImageSize*	*SetFillPattern*
ClearViewPort	*InitGraph*	*SetFillStyle*
CloseGraph	*InstallUserDriver*	*SetGraphBufSize*
DetectGraph	*InstallUserFont*	*SetGraphMode*
DrawPoly	*PieSlice*	*SetLineStyle*
FillPoly	*RegisterBGIdriver*	*SetPalette*
FloodFill	*RegisterBGIfont*	*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

```
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;
  { Do some graphics... }
  ClearDevice;
  Rectangle(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
end.
```

grXXXX constants Graph

Purpose Used by the *GraphResult* function to indicate the type of error that occurred.

Remarks

Constant	Value	Description
grOk	0	No error.
grNoInitGraph	–1	(BGI) graphics not installed (use *InitGraph*).
grNotDetected	–2	Graphics hardware not detected.
grFileNotFound	–3	Device driver file not found.
grInvalidDriver	–4	Invalid device driver file.
grNoLoadMem	–5	Not enough memory to load driver.
grNoScanMem	–6	Out of memory in scan fill.
grNoFloodMem	–7	Out of memory in flood fill.
grFontNotFound	–8	Font file not found.
grNoFontMem	–9	Not enough memory to load font.
grInvalidMode	–10	Invalid graphics mode for selected driver.
grError	–11	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.)
grIOerror	–12	Graphics I/O error.
grInvalidFont	–13	Invalid font file; the font header isn't recognized.
grInvalidFontNum	–14	Invalid font number; the font number in the font header is not recognized.

See also *GraphResult*

Halt procedure System

Purpose Stops program execution and returns to the operating system.

Declaration `procedure Halt [ ( ExitCode: Word ) ];`

Remarks *ExitCode* is an optional expression of type *Word* that specifies the program's exit code. *Halt* without a parameter corresponds to *Halt(0).*

Note that *Halt* initiates execution of any *Exit* procedures. See Chapter 20, "Control issues," in the *Language Guide* for more information.

See also *Exit, RunError*

HeapEnd variable — System

Purpose Points to the end of DOS memory used by programs.

Declaration `var HeapEnd: Pointer;`

Remarks *HeapEnd* is initialized by the system unit when your program begins. See Chapter 19, "Memory issues," in the *Language Guide* for more information.

See also *HeapOrg, HeapPtr*

HeapError variable — System

Purpose Points to the heap error function.

Declaration `var HeapError: Pointer;`

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 Chapter 19, "Memory issues," in the *Language Guide* for more information about using heap error functions.

See also *GetMem, New*

HeapOrg variable — System

Purpose Points to the bottom of the heap.

Declaration `var HeapOrg: Pointer;`

Remarks *HeapOrg* contains the address of the bottom of the heap. See Chapter 19, "Memory issues," in the *Language Guide* for more information.

See also *HeapEnd, HeapPtr*

HeapPtr variable — System

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 — System

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

```
var B: Byte;
begin
  B := Hi($1234);                                        { $12 }
end.
```

High function — System

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

See also *Low*

Example

```
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 — Crt

Purpose Selects high-intensity characters.

Declaration `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

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

ImageSize function — Graph

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

Declaration `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);                                     { Allocate memory on heap }
  GetImage(10, 20, 30, 40, P^);
  Readln;
  ClearDevice;
  PutImage(100, 100, P^, NormalPut);
  Readln;
  CloseGraph;
end.
```

Inc procedure — System

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;
```

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

Include procedure — System

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 — Graph

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

InOutRes variable — System

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 — System

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 Chapter 13, "Input and output," in the *Language Guide* for details about I/O issues.

See also *Output*

Insert procedure — System

Purpose Inserts a substring into a string.

Declaration
```
procedure Insert(Source: String; var S: String; Index: Integer);
```

Remarks *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 *Index*th position. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

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

Example
```
var S: string;
begin
  S := 'Honest Lincoln';
  Insert('Abe ', S, 8);                              { 'Honest Abe Lincoln' }
end.
```

InsLine procedure — Crt

Purpose Inserts an empty line at the cursor position.

Declaration
```
procedure InsLine;
```

Remarks All lines below the inserted line are moved down one line, and the bottom line 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.

Example *InsLine* is window-relative. The following example inserts a line 60 columns wide at absolute coordinates (1, 10):

```
   Window(1, 10, 60, 20);
   InsLine;
```

See also *DelLine, Window*

InstallUserDriver function Graph

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*:

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

```
{$F+}
function DetectSGA: Integer;
var Found: Boolean;
begin
  DetectSGA := grError;                          { Assume it's not there }
  Found := ...                                   { Look for the hardware }
```

```
    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 recommended 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 — Graph

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 — System

Purpose Returns the integer part of the argument.

Declaration `function Int(X: Real): Real;`

Remarks *X* is a real-type expression. The result is the integer part of *X*; that is, *X* rounded toward zero.

See also *Frac, Round, Trunc*

Example

```
var R: Real;
begin
  R := Int(123.456);                                        { 123.0 }
  R := Int(-123.456);                                      { -123.0 }
end.
```

Intr procedure — Dos, WinDos

Purpose Executes a specified software interrupt.

Declaration

```
procedure Intr(IntNo: Byte; var Regs: Registers);                  {Dos}

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

Remarks *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*.

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 — System

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(1));
  {$I-}
  Reset(F);
  {$I+}
  if IOResult = 0 then
    Writeln('File size in bytes: ', FileSize(F))
  else
    Writeln('File not found');
end.
```

See also *InOutRes*

Justification constants — Graph

Purpose Constants that control horizontal and vertical justification.

Remarks

Constant	Value
LeftText	0
CenterText	1
RightText	2
BottomText	0
CenterText	1
TopText	2

See also *SetTextJustify*

Keep procedure — Dos

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 — Crt

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*

Example

```
uses Crt;
begin
  repeat
    Write('Xx');                          { Fill the screen until a key is typed }
  until KeyPressed;
end.
```

LastMode variable — Crt

Purpose Stores current video mode each time *TextMode* is called.

Declaration `var LastMode: Word;`

Remarks At program startup, *LastMode* is initialized to the then-active video mode.

See also *TextMode*

Length function — System

Purpose Returns the dynamic length of a string.

Declaration `function Length(S: String): Integer;`

Remarks Returns the length of the String *S*.

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

Example

```
var S: String;
begin
  Readln(S);
  Writeln('"', S, '"');
  Writeln('length = ', Length(S));
end.
```

Line procedure — Graph

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

Declaration `procedure Line(X1, Y1, X2, Y2: 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 **XOR**ed to the screen.

Note that

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

is equivalent to

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

Use *LineTo* when the current pointer 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 Graph

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

Remarks The following *Line style* constants are defined:

Constant	Value
SolidLn	0
DottedLn	1
CenterLn	2
DashedLn	3

UserBitLn	4 (user-defined line style)
NormWidth	1
ThickWidth	3

See also *LineSettingsType*

LineRel procedure — Graph

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 **XOR**ed 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, '');
  if GraphResult <> grOk then
    Halt(1);
  MoveTo(1, 2);
  LineRel(100, 100);                          { Draw to the point (101,102) }
  Readln;
  CloseGraph;
end.
```

LineSettingsType type Graph

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

Declaration

```
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 Graph

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

Declaration

```
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 **XOR**ed to the screen.

Note that

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

is equivalent to

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

The first method is slower and uses more code. Use *LineTo* only when the current pointer is at one endpoint of the line. Use *LineRel* to draw a line a 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

```
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;
  Readln;
  CloseGraph;
end.
```

Ln function System

Purpose Returns the natural logarithm of the argument.

Declaration `function Ln(X: Real): Real;`

Remarks Returns the natural logarithm of the real-type expression *X*.

See also *Exp*

Lo function System

Purpose Returns the low-order byte of the argument.

Declaration `function Lo(X): Byte;`

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

See also *Hi*, *Swap*

Example

```
var B: Byte;
begin
  B := Lo($1234);   { $34 }
end.
```

Low function System

Purpose Returns the lowest value in the range of the argument.

Declaration `function Low(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, *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

```
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
  DayName: array[TDay] of string[3] = (
    'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');
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 — Crt

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*

Example

```
uses Crt;
begin
  TextAttr := White;
  LowVideo;                                        { Color is now light gray }
end.
```

Lst variable — Printer

Purpose Stores the standard output as a text file.

Declaration `var Lst: Text;`

Remarks Use *Lst* to send the output of your program to the printer.

See also *Assign, Rewrite*

Example

```
program PrintIt;
var
   Lst: Text;                                { Declare Lst as text file variable }
begin
  Assign(Lst, 'LPT1');                        { Assign text file to standard output }
  Rewrite(Lst);                           { Call Rewrite to send text file to printer }
  Writeln(Lst, 'Hello printer.');
  Close(Lst)                                                    { Close text file }
end.
```

MaxAvail function — System

Purpose 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.

Declaration `function MaxAvail: Longint;`

Remarks *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.

See also *MemAvail*

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 — Graph

Purpose The constant that determines the maximum number of colors.

Declaration `const MaxColors = 15;`

See also *GetDefaultPalette, GetPalette, SetAllPalette*

MemAvail function — System

Purpose Returns the amount of free memory in the heap.

Declaration `function MemAvail: Longint;`

Remarks 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 **$M** directive.

See also *MaxAvail*

Example

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

MkDir procedure — System

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));                    { Get directory name from command line }
  if IOResult <> 0 then
    Writeln('Cannot create directory')
  else
    Writeln('New directory created');
end.
```

Move procedure — System

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

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

MoveRel procedure — Graph

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;
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 — Graph

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

Declaration `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	*LineRel*	*OutText*
ClearViewPort	*LineTo*	*SetGraphMode*
GraphDefaults	*MoveRel*	*SetViewPort*
InitGraph	*MoveTo*	

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

```
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 — Dos, WinDos

Purpose Executes a DOS function call.

Declaration

```
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 — System

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^*. 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 — Crt

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*

M

NoSound procedure — Crt

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 — System

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 — System

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 — System

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

Purpose Standard output file.

Declaration `var Output: Text;`

Remarks *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.

See also *Input*

OutText procedure — Graph

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

Declaration `procedure OutText(TextString: String);`

Remarks 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 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.

```
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');                           { CP is updated }
  OutText('DEF');                           { CP is updated }
  { #2 }
  MoveTo(100, 50);
  SetTextStyle(DefaultFont, HorizDir, 1);   { CharSize = 1 }
  SetTextJustify(RightText, TopText);
  OutText('ABC');                           { CP is updated }
  OutText('DEF');                           { CP is updated }
  { #3 }
  MoveTo(100, 100);
  SetTextStyle(DefaultFont, VertDir, 1);    { CharSize = 1 }
  SetTextJustify(LeftText, TopText);
  OutText('ABC');                           { CP is NOT updated }
  OutText('DEF');                           { CP is NOT updated }
  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:

```
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 — Graph

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.

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 — Overlay

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 — System

Purpose Overlay code segment list.

Declaration `var OvrCodeList: Word;`

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 System

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 System

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 *OverInit*

OvrEmsHandle variable System

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 *OverInitEMS* 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 *OverInit*, *OverInitEMS*

OvrFileMode variable — Overlay

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

Declaration `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 — Overlay

Purpose Returns the current size of the overlay buffer.

Declaration `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

```
{$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 — Overlay

Purpose Returns the current size of the probation area.

Declaration `function OvrGetRetry: Longint;`

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 — System

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 — System

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 — Overlay

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 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 — Overlay

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

```
uses Overlay;
begin
  OvrInit('EDITOR.OVR');
  if OvrResult <> ovrOk then
  begin
    Writeln('Overlay manager initialization failed.');
    Halt(1);
  end;
  OvrInitEMS;
  case OvrResult of
    ovrIOError: 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 — Overlay

Purpose Overlay load count.

Declaration `var OvrLoadCount: Word;`

Remarks The initial value of *OvrLoadCount* is zero. The overlay manager increments *OvrLoadCount* each time an overlay is loaded. By examining *OvrTrapCount* and *OvrLoadCount* in the Debugger's Watch window during identical runs of an application, you can monitor the effect of different probation area sizes (set with *OvrSetRetry*) to find the optimal size for your particular application.

See also *OvrTrapCount*

OvrLoadList variable — System

Purpose Loaded overlays list.

Declaration `var OvrLoadList: Word;`

Remarks *OvrLoadList* is used internally by the overlay manager. You should never modify *OvrLoadList*.

OvrReadBuf variable — Overlay

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 — Overlay

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 — Overlay

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.*

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 — Overlay

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*:

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

OvrTrapCount variable — Overlay

Purpose Overlay call interception count.

Declaration `var OvrTrapCount: Word;`

Remarks 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 *OvrTrapCount* variable is incremented. The initial value of *OvrTrapCount* is zero.

See also *OvrLoadCount*

ovrXXXX constants — Overlay

Purpose Return codes stored in the *OvrResult* variable.

Remarks

Constant	Value	Meaning
ovrOk	0	Success
ovrError	–1	Overlay manager error
ovrNotFound	–2	Overlay file not found
ovrNoMemory	–3	Not enough memory for overlay buffer
ovrIOError	–4	Overlay file I/O error
ovrNoEMSDriver	–5	EMS driver not installed
ovrNoEMSMemory	–6	Not enough EMS memory

PackTime procedure — Dos, WinDos

Purpose Converts a *DateTime* record into a 4-byte, packed date-and-time *Longint* used by *SetFTime*.

Declaration

```
procedure PackTime(var DT: DateTime; var Time: Longint);                {Dos}

procedure PackTime(var DT: TDateTime; var Time: Longint);            {WinDos}
```

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 — Graph

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 — System

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*

Example

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

ParamStr function — System

Purpose Returns a specified command-line parameter.

Declaration `function ParamStr(Index): String;`

Remarks *Index* is an expression of type *Word*. *ParamStr* returns the *Index*th 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 — System

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 — Graph

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*.

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 — Graph

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

Declaration

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

Pos function — System

Purpose 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.

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

Example

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

Pred function — System

Purpose Returns the predecessor of the argument.

Declaration `function Pred(X);`

Remarks *X* is an ordinal-type expression. The result, of the same type as *X*, is the predecessor of *X*.

See also *Dec, Inc, Succ*

PrefixSeg variable — System

Purpose Contains the segment address of the Program Segment Prefix (PSP) created by DOS when the application executes.

Declaration `var PrefixSeg: Word;`

Remarks For a complete description of the Program Segment Prefix, see your DOS manuals.

Ptr function — System

Purpose Converts a segment base and an offset address to a pointer-type value.

Declaration `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:

```
if Byte(Ptr(Seg0040, $49)^) = 7 then
  Writeln('Video mode = mono');
```

See also *Addr, Ofs, Seg*

Example

```
var P: ^Byte;
begin
  P := Ptr(Seg0040, $49);
  Writeln('Current video mode is ', P^);
end.
```

PutImage procedure — Graph

Purpose Puts a bit image onto the screen.

Declaration `procedure PutImage(X, Y: Integer; var BitMap; BitBlt: Word);`

Remarks (*X*, *Y*) is the upper left corner of a rectangular region on the screen. *BitMap* is an untyped parameter that contains the height and width of the region, and the bit image that will be put onto the screen. *BitBlt* specifies which binary operator will be used to put the bit image onto the screen. See page 12 for a list of *BitBlt* operators.

Each constant corresponds to a binary operation. For example, *PutImage(X, Y, BitMap, NormalPut)* puts the image stored in *BitMap* at (*X*, *Y*) using the assembly language **MOV** instruction for each byte in the image.

Similarly, *PutImage(X, Y, BitMap, XORPut)* puts the image stored in *BitMap* at (*X*, *Y*) using the assembly language **XOR** instruction for each byte in the image. This is an often-used animation technique for "dragging" an image around the screen.

PutImage(X, Y, BitMap, NotPut) inverts the bits in *BitMap* and then puts the image stored in *BitMap* at (*X*, *Y*) using the assembly language **MOV** for each byte in the image. Thus, the image appears in inverse video of the original *BitMap*.

Note that *PutImage* 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 *PutImage* statements would have no effect:

```
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^);                     { Width = 100, height = 50 }
  ClearDevice;
  PutImage(GetMaxX - 99, 0,                                  { Will barely fit }
    P^, NormalPut);
  PutImage(GetMaxX - 98, 0,                              { X + Height > GetMaxX }
    P^, NormalPut);
  PutImage(-1, 0,                                         { -1,0 not onscreen }
    P^, NormalPut);
  PutImage(0, -1,                                         { 0,-1 not onscreen }
    P^, NormalPut);
  PutImage(0, GetMaxY - 30,                            { Will output 31 "lines" }
    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);                                 { Allocate memory on heap }
  GetImage(10, 20, 30, 40, P^);
  Readln;
  ClearDevice;
```

P-Q

```
  PutImage(100, 100, P^, NormalPut);
  Readln;
  CloseGraph;
end.
```

PutPixel procedure — Graph

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 — System

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 <= *X* < *Range*. If *Range* equals 0, a value of 0 is returned.

The random number generator should be initialized by making a call to *Randomize*, or by assigning a value to *RandSeed*.

See also *Randomize*

Randomize procedure — System

Purpose Initializes the built-in random generator with a random value.

Declaration `procedure Randomize;`

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 — System

Purpose Stores the built-in random number generator's seed.

Declaration `var RandSeed: Longint;`

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, Randomize*

Read procedure (text files) — System

Purpose Reads one or more values from a text file into one or more variables.

Declaration **procedure** Read([**var** F: Text;] V_1 [, V_2,...,V_N]);

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 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 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) System

Purpose Reads a file component into a variable.

Declaration `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 {**$I-**}, *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 Crt

Purpose Reads a character from the keyboard.

Declaration `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:

```
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 — System

Purpose Executes the *Read* procedure then skips to the next line of the file.

Declaration `procedure Readln( [ var F: Text; ] V1 [, V2,...,VN ] );`

Remarks *Readln* is an extension to *Read*, as it is defined on text files. After executing the *Read*, *Readln* skips to the beginning of the next line of the file.

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 — Graph

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

Declaration `procedure Rectangle(X1, Y1, X2, Y2: Integer);`

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 **XOR**ed 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 — Graph

Purpose 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 function — Graph

Purpose Registers a user-loaded or linked-in BGI font with the graphics system.

Declaration `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 (8×8 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 *RegisterBGIdriver*.

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*):

```
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;
```

```
    { Reselect registered font (already in memory) }
    SetTextStyle(TriplexFont, HorizDir, 4);
    OutText('Back to Triplex');
    Readln;
    CloseGraph;
  end.
```

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 — Dos

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

```
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;
```

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 — WinDos

Purpose Removes an empty subdirectory.

Declaration `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.

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 — System

Purpose Renames an external file.

Declaration `procedure Rename(var F; 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 — System

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*.

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, '');
  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.
```

Rewrite procedure — System

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

```
var F: Text;
begin
  Assign(F, 'NEWFILE.$$$');
  Rewrite(F);
  Writeln(F, 'Just created file with this text in it...');
  Close(F);
end.
```

RmDir procedure System

Purpose Removes an empty subdirectory.

Declaration `procedure RmDir(S: String);`

Remarks 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.

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

See also *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.

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.
```

R

Round function System

Purpose Rounds a real-type value to an integer-type value.

Declaration `function Round(X: Real): Longint;`

Remarks *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 greatest absolute magnitude. A run-time error occurs if the rounded value of *X* is not within the *Longint* range.

See also *Int, Trunc*

RunError procedure — System

Purpose Stops program execution and generates a run-time error.

Declaration
```
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
```
{$IFDEF Debug}
  if P = nil then
    RunError(204);
{$ENDIF}
```

SaveIntXX variables — System

Purpose Stores interrupt vectors.

Declaration The *System* unit declares the following *SaveIntXX* variables.

Name	Type	Description
SaveInt00	Pointer	{ Saved interrupt $00 }
SaveInt02	Pointer	{ Saved interrupt $02 }
SaveInt1B	Pointer	{ Saved interrupt $1B }
SaveInt21	Pointer	{ Saved interrupt $21 }
SaveInt23	Pointer	{ Saved interrupt $23 }
SaveInt24	Pointer	{ Saved interrupt $24 }
SaveInt34	Pointer	{ Saved interrupt $34 }
SaveInt35	Pointer	{ Saved interrupt $35 }
SaveInt36	Pointer	{ Saved interrupt $36 }
SaveInt37	Pointer	{ Saved interrupt $37 }
SaveInt38	Pointer	{ Saved interrupt $38 }
SaveInt39	Pointer	{ Saved interrupt $39 }
SaveInt3A	Pointer	{ Saved interrupt $3A }

SaveInt3B	Pointer	{ Saved interrupt $3B }
SaveInt3C	Pointer	{ Saved interrupt $3C }
SaveInt3D	Pointer	{ Saved interrupt $3D }
SaveInt3E	Pointer	{ Saved interrupt $3E }
SaveInt3F	Pointer	{ Saved interrupt $3F }
SaveInt75	Pointer	{ Saved interrupt $75 }

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 — Dos

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 — Graph

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;                                      { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  Sector(GetMaxX div 2, GetMaxY div 2, 0, 45, R, R);
  GetAspectRatio(Xasp, Yasp);                           { Draw circular sector }
  Sector(GetMaxX div 2, GetMaxY div 2,                          { Center point }
    180, 135,                                              { Mirror angle above }
    R, R * Longint(Xasp) div Yasp);                                 { Circular }
  Readln;
  CloseGraph;
end.
```

Seek procedure — System

Purpose Moves the current position of a file to a specified component.

Declaration `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 {**$I-**}, *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 — System

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

Declaration `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 {**$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 *Eof*, *SeekEoln*

SeekEoln function — System

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

Declaration `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.

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 System

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 System

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 System

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 — System

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 — System

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, SegB000*

SelectorInc variable — System

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 — Graph

Purpose Set the active page for graphics output.

Declaration `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

```
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 — Graph

Purpose Changes all palette colors as specified.

Declaration `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, SetPalette, 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 Graph

Purpose Changes the default aspect-ratio correction factor.

Declaration `procedure SetAspectRatio(Xasp, Yasp: Word): Word;`

Remarks *SetAspectRatio* is used to change the default aspect ratio of the current graphics mode. The aspect ratio is used to draw circles. If circles appear elliptical, the monitor is not 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 *SetAspectRatio*. To read the current aspect ratio from the system, use *GetAspectRatio*.

Restrictions Must be in graphics mode.

See also *GetAspectRatio*

Example

```
uses Crt, Graph;
const R = 50;
var
  Driver, Mode: Integer;
  Xasp, Yasp: Word;
begin
  DirectVideo := False;
  Driver := Detect;                                          { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  GetAspectRatio(Xasp, Yasp);                                { Get default aspect ratio }
  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 — Graph

Purpose Sets the current background color using the palette.

Declaration `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 *N*th 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

```
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 — Dos, WinDos

Purpose Sets the state of *Ctrl+Break* checking in DOS.

Declaration `procedure SetCBreak(Break: Boolean);`

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 — Graph

Purpose Sets the current drawing color using the palette.

Declaration `procedure SetColor(Color: Word);`

Remarks *SetColor*(5) 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 — WinDos

Purpose Changes the current directory to the path specified by *Dir*.

Declaration `procedure SetCurDir(Dir: PChar);`

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 — Dos, WinDos

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 — Dos, WinDos

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;                                          { or WinDos }
var F: file;
begin
  Assign(F, 'C:\AUTOEXEC.BAT');
  SetFAttr(F, Hidden);                           { or faHidden }
  Readln;
  SetFAttr(F, Archive);                         { or faArchive }
end.
```

S

SetFillPattern procedure — Graph

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):

Binary		Hex	
10101010	=	$AA	(1st byte)
01010101	=	$55	(2nd byte)
10101010	=	$AA	(3rd byte)
01010101	=	$55	(4th byte)
10101010	=	$AA	(5th byte)
01010101	=	$55	(6th byte)
10101010	=	$AA	(7th byte)
01010101	=	$55	(8th byte)

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 — Graph

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 — Dos, WinDos

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

Declaration `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 — Graph

Purpose Lets you change the size of the buffer used for scan and flood fills.

Declaration `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 — Graph

Purpose Sets the system to graphics mode and clears the screen.

Declaration `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:

```
GetIntVec($1B, Int1BSave);
SetIntVec($1B, @Int1BHandler);
   ⋮
SetIntVec($1B, Int1BSave);
```

See also *GetIntVec*

SetLineStyle procedure — Graph

Purpose Sets the current line width and style.

Declaration `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

```
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;
  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 — Graph

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
```

S

```
    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 — Graph

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:

```
uses Graph;
type
  RGBRec = record
    RedVal, GreenVal, BlueVal: Integer;
  end;
const
  EGAColors: array[0..MaxColors] of RGBRec =
    (                                     {NAME       COLOR}
    (RedVal:$00;GreenVal:$00;BlueVal:$00),{Black      EGA  0}
    (RedVal:$00;GreenVal:$00;BlueVal:$FC),{Blue       EGA  1}
    (RedVal:$24;GreenVal:$FC;BlueVal:$24),{Green      EGA  2}
    (RedVal:$00;GreenVal:$FC;BlueVal:$FC),{Cyan       EGA  3}
    (RedVal:$FC;GreenVal:$14;BlueVal:$14),{Red        EGA  4}
    (RedVal:$B0;GreenVal:$00;BlueVal:$FC),{Magenta    EGA  5}
    (RedVal:$70;GreenVal:$48;BlueVal:$00),{Brown      EGA 20}
    (RedVal:$C4;GreenVal:$C4;BlueVal:$C4),{White      EGA  7}
    (RedVal:$34;GreenVal:$34;BlueVal:$34),{Gray       EGA 56}
    (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:$70),{Lt Magenta EGA 61}
    (RedVal:$FC;GreenVal:$FC;BlueVal:$24),{Yellow     EGA 62}
    (RedVal:$FC;GreenVal:$FC;BlueVal:$FC) {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 }
  { Change first 16 8514 palette entries }
  for I := 1 to MaxColors do
  begin
    SetColor(I);
    OutTextXY(10, I * 10, ' ..Press any key.. ');
  end;
  { Restore default EGA colors to 8514 palette }
  for I := 0 to MaxColors do
    with EGAColors[I] do
      SetRGBPalette(I, RedVal, GreenVal, BlueVal);
  Readln;
  CloseGraph;
end.
```

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;
```

```
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
  Halt(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 — System

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 — Graph

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

Declaration `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:

```
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

```
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 — Graph

Purpose Sets the current text font, style, and character magnification factor.

Declaration `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-mapped font in an 8×8 pixel rectangle onscreen, a *CharSize* value equal to 2 will display the 8×8 bit-mapped font in a 16×16 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*, *GetTextSettings*, *GraphResult*, *OutText*, *OutTextXY*, *RegisterBGIfont*, *SetTextJustify*, *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 — Dos, WinDos

Purpose 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 *Sec*100 (hundredths of seconds) 0..99. If the time isn't valid, the request is ignored.

See also *GetDate, GetTime, PackTime, SetDate, UnpackTime*

SetUserCharSize procedure — Graph

Purpose 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;
begin
  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 — Dos, WinDos

Purpose 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 — Graph

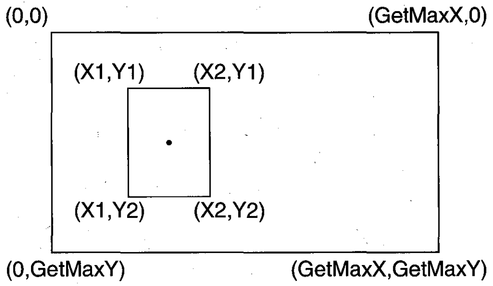
Purpose Sets the current output viewport or window for graphics output.

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.



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:

```
  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 *ClearViewPort*, *GetViewSettings*, *GraphResult*

Example

```
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.
```

SetVisualPage procedure — Graph

Purpose Sets the visual graphics page number.

Declaration `procedure SetVisualPage(Page: Word);`

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;
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.
```

SetWriteMode procedure — Graph

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 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;                               { Turn off screen write }
  Randomize;
  Driver := Detect;                                    { Put in graphics mode }
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  { Fill screen with background pattern }
  GetFillSettings(FillInfo);                           { Get current settings }
  SetFillStyle(WideDotFill, FillInfo.Color);
  Bar(0, 0, GetMaxX, GetMaxY);
  Dx := GetMaxX div 4;                      { Determine rectangle's dimensions }
  Dy := GetMaxY div 4;
  SetLineStyle(SolidLn, 0, ThickWidth);
  SetWriteMode(XORPut);                               { XOR mode for rectangle }
  repeat                                         { Draw until a key is pressed }
    X1 := Random(GetMaxX - Dx);
    Y1 := Random(GetMaxY - Dy);
    Rectangle(X1, Y1, X1 + Dx, Y1 + Dy);                            { Draw it }
    Delay(10);                                                { Pause briefly }
    Rectangle(X1, Y1, X1 + Dx, Y1 + Dy);                           { Erase it }
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

Sin function — System

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

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

SizeOf function — System

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:

```
FillChar(S, SizeOf(S), 0);
GetMem(P, SizeOf(RecordType));
```

Example

```
type
  CustRec = record
    Name: string[30];
    Phone: string[14];
  end;
var P: ^CustRec;
begin
  GetMem(P, SizeOf(CustRec));
end.
```

Sound procedure — Crt

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

```
uses Crt;
begin
  Sound(220);
  Delay(200);
  NoSound;
end.
```

SPtr function — System

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 — System

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 — System

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 — System

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 — System

Purpose Contains the offset of the bottom of the stack in the stack segment.

Declaration `var StackLimit: Word;`

Remarks *StackLimit* returns the lowest value the SP register can contain before it is considered a stack overflow.

See also *SPtr*

Str procedure System

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

```
function IntToStr(I: Longint): String;
{ Convert any integer type to a string }
var S: string[11];
begin
  Str(I, S);
  IntToStr := S;
end;
begin
  Writeln(IntToStr(-5322));
end.
```

StrCat function Strings

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

```
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 — Strings

Purpose Compares two strings.

Declaration `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, StrLIComp*

Example

```
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 — Strings

Purpose Copies one string to another.

Declaration `function StrCopy(Dest, Source: PChar): PChar;`

S

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 — Strings

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 — Strings

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   Strings

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   Strings

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, StrLComp, StrLIComp*

S

StrLCat function   Strings

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;`

Remarks *StrLCat* appends at most *MaxLen – 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

```
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 — Strings

Purpose Compares two strings, up to a maximum length.

Declaration `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

```
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 — Strings

Purpose Copies characters from one string to another.

Declaration `function StrLCopy(Dest, Source: PChar; MaxLen: Word): PChar;`

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 — Strings

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 — Strings

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 — Strings

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 — Strings

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 — Strings

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 — Strings

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 — Strings

Purpose Copies a Pascal-style string into a null-terminated string.

Declaration `function StrPCopy(Dest: PChar; 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 — Strings

Purpose Returns a pointer to the first occurrence of a string in another string.

Declaration `function StrPos(Str1, Str2: PChar): PChar;`

Remarks *StrPos* returns a pointer to the first occurrence of *Str2* in *Str1*. If *Str2* does not occur in *Str1*, *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 — Strings

Purpose Returns a pointer to the last occurrence of a character in a string.

Declaration `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

```
{ 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 — Strings

Purpose Returns a pointer to the first occurrence of a character in a string.

Declaration `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

```
{ 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 — Strings

Purpose Converts a string to uppercase.

Declaration `function StrUpper(Str: PChar): PChar;`

Remarks *StrUpper* converts *Str* to uppercase and returns *Str*.

See also *StrLower*

Example

```
uses Strings;
var
  S: array[0..79] of Char;
begin
  Readln(S);
  Writeln(StrUpper(S));
  Writeln(StrLower(S));
end.
```

Succ function — System

Purpose Returns the successor of the argument.

Declaration `function Succ(X):`

Result type Same type as parameter.

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 — System

Purpose Swaps the high- and low-order bytes of the argument.

Declaration `function Swap(X);`

Result type Same type as parameter.

Remarks *X* is an expression of type *Integer* or *Word*.

See also *Hi, Lo*

Example

```
var X: Word;
begin
  X := Swap($1234);   { $3412 }
end.
```

SwapVectors procedure — Dos

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 *Exec*ed 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.
```

S

TDateTime type — WinDos

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;
```

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 — System

Purpose Identifies the type of 80x86 processor the system contains.

Declaration `var Test8086: Byte;`

Remarks 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:

Value	Definition
0	Processor is an 8086
1	Processor is an 80286
2	Processor is an 80386 or later

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 — System

Purpose Stores the results of the 80x87 autodetection logic and coprocessor classification.

Declaration `var Test8087: Byte;`

Remarks The *Test8087* variable indicates whether floating-point instructions are being emulated or actually executed. The following values stored in *Test8087* are defined.

Value	Definition
0	No coprocessor detected
1	8087 detected
2	80287 detected
3	80387 or later detected

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.

```
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 — Crt

Purpose Stores the currently selected text attribute.

Declaration `var TextAttr: Byte;`

Remarks Although text attributes are normally set through calls to *TextColor* and *TextBackground*, you can also set them by directly storing a value in *TextAttr*. The color information is encoded in *TextAttr* as follows:

bit →	7	6	5	4	3	2	1	0
	B	b	b	b	f	f	f	f

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 — Crt

Purpose Represents the text colors.

Remarks The following constants are used in connection with the *TextColor* and *TextBackground* procedures.

Constant	**Value**
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
LightGray	7
DarkGray	8
LightBlue	9
LightGreen	10
LightCyan	11
LightRed	12
LightMagenta	13
Yellow	14
White	15
Blink	128

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 — Crt

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*.

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

See also *HighVideo, LowVideo, NormVideo, TextColor, Text color*

TextColor procedure — Crt

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 — Graph

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.

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 — Crt

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

```
TextMode(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 *Crt* startup code, and the window variable that keeps track of the maximum number of lines onscreen (*WindMax*) will be initialized correctly.

Here's how to write a "well-behaved" program that will restore the video mode to its original state:

```
program Video;
uses Crt;
var OrigMode: Integer;
begin
  OrigMode := LastMode;                                { Remember original mode }
     :
  TextMode(OrigMode);
end.
```

Note that *TextMode* does not support graphics modes, and therefore *TextMode*(*OrigMode*) will only restore those modes supported by *TextMode*.

See also *Crt mode constants, RestoreCrtMode*

TextRec type — Dos

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

Declaration

```
type
  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;
    FlushFunc: Pointer;
    CloseFunc: 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 — Graph

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 — Graph

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.

See also *OutText, OutTextXY, SetTextStyle, SetUserCharSize, TextHeight*

Example

```
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

```
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.*

T-V

TRegisters type WinDos

Purpose Variables of type *TRegisters* are used by *Intr* and *MsDos* procedures to specify the specify input register contents and examine output register contents of a software interrupt.

Declaration
```
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 System

Purpose Truncates a real-type value to an integer-type value.

Declaration `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 System

Purpose Truncates the file size at the current file position.

Declaration `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 WinDos

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 WinDos

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;
```

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

Purpose Returns a pointer to an object type's virtual method table (VMT).

Declaration `function TypeOf(X): Pointer;`

Remarks *X* is either an object type identifier or an instance of an object type. In either case, *TypeOf* returns the address of the object type's virtual method table. *TypeOf* 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 *Language Guide*.

UnpackTime procedure — Dos, WinDos

Purpose Converts a 4-byte, packed date-and-time *Longint* returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *DateTime* record.

Declaration `procedure UnpackTime(Time: Longint; var DT: DateTime);`

Remarks *DateTime* is a record declared in the *Dos* unit. If you are writing Windows programs, use *TDateTime*. The fields of the *Time* record are not range-checked. See page 26 for the *DateTime* record declaration and page 189 for the *TDateTime* record declaration.

See also *DateTime, GetFTime, GetTime, PackTime, SetFTime, SetTime, TDateTime*

UpCase function — System

Purpose Converts a character to uppercase.

Declaration `function UpCase(Ch: Char): Char;`

Remarks *Ch* is an expression of type *Char*. The result of type *Char* is *Ch* converted to uppercase. Character values not in the range *a..z* are unaffected.

Val procedure — System

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:

```
{$R-}
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*

Example

```
var I, Code: Integer;
begin
  Val(ParamStr(1), I, Code);                       { Get text from command line }
  if code <> 0 then                           { Error during conversion to integer? }
    Writeln('Error at position: ', Code)
  else
    Writeln('Value = ', I);
end.
```

ViewPortType type — Graph

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 — Crt

Purpose Returns the *X*-coordinate of the current cursor position, relative to the current window.

Declaration `function WhereX: Byte;`

See also *GotoXY, WhereY, Window*

WhereY function — Crt

Purpose Returns the *Y*-coordinate of the current cursor position, relative to the current window.

Declaration `function WhereY: Byte;`

See also *GotoXY, WhereX, Window*

WindMax and WindMin variables — Crt

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 — Crt

Purpose Defines a text window onscreen.

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

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

Example

```
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) — System

Purpose Writes one or more values to a text file.

Declaration **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

```
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 1.3 The components of the output string

[| –]	" " or "-", according to the sign of *OutExpr*
<digit>	Single digit, "0" only if *OutExpr* is 0
<decimals>	Digit string of *MinWidth*-7 (but at most 10) digits
E	Uppercase [E] character
[+ | –]	According to sign of exponent
<exponent>	Two-digit decimal exponent

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 1.4 The components of the fixed-point string

[<blanks>]	Blanks to satisfy *MinWidth*
[–]	If *OutExpr* is negative
<digits>	At least one digit, but no leading zeros
[. <decimals>]	Decimals if *DecPlaces* > 0

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) System

Purpose Writes a variable into a file component.

Declaration `procedure Write(F, V`$_1$` [, V`$_2$`,...,V`$_N$` ] );`

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 System

Purpose Executes the *Write* procedure, then outputs an end-of-line marker to the file.

Declaration `procedure Writeln( [ var F: Text; ] P`$_1$` [, P`$_2$`,...,P`$_N$` ] );`

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 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*

C H A P T E R

2

Compiler directives

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.

See Appendix B for a table summarizing the compiler directives.

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.

See page 223 for more information about using conditional compilation directives.

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}
{$I TYPES.INC}
{$O EdFormat}
{$M 65520,8192,655360}
{$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 /$R+ 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 O*. Turbo Pascal will insert the current settings at the top of your edit window.

Align data — Switch

Syntax `{$A+}` or `{$A-}`

Default `{$A+}`

Type Global

Remarks 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.

Boolean evaluation — Switch

Syntax `{$B+}` or `{$B-}`

Default `{$B-}`

Type Local

Remarks 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.

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 — Switch

Syntax `{$D+}` or `{$D-}`

Default `{$D+}`

Type Global

Remarks The **$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 {**$D+**} state, Turbo Pascal's integrated debugger lets you single-step and set breakpoints in that module.

The Standalone debugging (Options | Debugger) and Map file (Options | Linker) options produce complete line information for a given module only if you've compiled that module in the {**$D+**} state.

The **$D** switch is usually used in conjunction with the **$L** switch, which enables and disables the generation of local symbol information for debugging.

☞ 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 — Conditional compilation

Syntax `{$DEFINE name}`

Remarks Defines a conditional symbol with the given *name*. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {**$UNDEF** *name*} directive. The {**$DEFINE** *name*} directive has no effect if *name* is already defined.

ELSE directive — Conditional compilation

Syntax `{$ELSE}`

Remarks Switches between compiling and ignoring the source text delimited by the last {**$IF***xxx*} and the next {**$ENDIF**}.

Emulation — Switch

Syntax `{$E+}` or `{$E-}`

Default `{$E+}`

Type Global

Remarks The **$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 {**$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 {**$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 {**$N-**} state, and if all the units used by the program were compiled with {**$N-**}, then an 80x87 run-time library is not required, and the 80x87 emulation switch is ignored.

ENDIF directive — Conditional compilation

Syntax `{$ENDIF}`

Remarks Ends the conditional compilation initiated by the last {**$IF***xxx*} directive.

Extended syntax — Switch

Syntax `{$X+}` or `{$X-}`

Default `{$X+}`

Type Global

Remarks The **$X** directive enables or disables Turbo Pascal's extended syntax:

*The **{$X+}** directive does not apply to built-in functions (those defined in the System unit).*

- Function statements. In the {**$X+**} mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. However, in certain cases a function can carry out multiple operations based on its parameters, and some of those cases may not produce a useful result. In such cases, the {**$X+**} extensions allow the function to be treated as a procedure.
- Null-terminated strings. A {**$X+**} compiler directive enables Turbo Pascal's support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. For more details about null-terminated strings, see Chapter 16, "Using null-terminated strings," in the *Language Guide.*

Force far calls — Switch

Syntax `{$F+}` or `{$F-}`

Default `{$F-}`

Type Local

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 — Switch

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 — Conditional compilation

Syntax `{$IFDEF name}`

Remarks Compiles the source text that follows it if *name* is defined.

IFNDEF directive — Conditional compilation

Syntax `{$IFNDEF name}`

Remarks Compiles the source text that follows it if *name* is not defined.

IFOPT directive — Conditional compilation

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

```
{$IFOPT N+}
  type Real = Extended;
{$ENDIF}
```

will compile the type declaration if the **$N** option is currently active.

Include file — Parameter

Syntax `{$I filename}`

Type Local

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 — Switch

Syntax `{$I+}` or `{$I-}`

Default `{$I+}`

Type Local

Remarks 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*.

Link object file — Parameter

Syntax `{$L filename}`

Type Local

Remarks 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).

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 — Switch

Syntax `{$L+}` or `{$L-}`

Default `{$L+}`

Type Global

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.

Memory allocation sizes — Parameter

Syntax `{$M stacksize,heapmin,heapmax}`

Default `{$M 16384,0,655360}`

Type Global

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 — Switch

Syntax `{$N+}` or `{$N-}`

Default `{$N-}`

Type Global

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 — Switch

Syntax `{$P+}` or `{$P-}`

Default `{$P-}`

Type Local

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 — Switch

Syntax `{$Q+}` or `{$Q-}`

Default `{$Q-}`

Type Local

Remarks 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.

Overlay code generation — Switch

Syntax `{$O+}` or `{$O-}`

Default `{$O-}`

Type Global

Remarks 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.

☞ 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 — Parameter

Syntax `{$O unitname}`

Type Local

Remarks The Overlay unit name directive turns a unit into an overlay.

The {**$O** *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.

{**$O** *unitname*} directives must be placed after the program's **uses** clause. Turbo Pascal reports an error if you attempt to overlay a unit that wasn't compiled in the {**$O+**} state.

Range checking — Switch

Syntax `{$R+}` or `{$R-}`

Default `{$R-}`

Type Local

Remarks 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.

Stack-overflow checking — Switch

Syntax `{$S+}` or `{$S-}`

Default `{$S+}`

Type Local

Remarks 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 {**$S+**} causes the program to terminate and display a runtime error message. In the {**$S-**} state, such a call is likely to cause a system crash.

Symbol reference information — Switch

Syntax `{$Y+}` or `{$Y-}`

Default `{$Y+}`

Type Global

Remarks 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.

Type-checked pointers — Switch

Syntax `{$T+}` or `{$T-}`

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 — Conditional compilation

Syntax `{$UNDEF name}`

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 — Switch

Syntax `{$V+}` or `{$V-}`

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,

```
{$IFxxx}
   ⋮
{$ENDIF}
```

causes the source text between {**$IF***xxx*} and {**$ENDIF**} to be compiled only if the condition specified in {**$IF***xxx*} is *True*. If the condition is *False*, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFxxx}
   ⋮
{$ELSE}
   ⋮
{$ENDIF}
```

causes either the source text between {**$IF***xxx*} and {**$ELSE**} or the source text between {**$ELSE**} and {**$ENDIF**} to be compiled, depending on the condition specified by the {**$IF***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 {**$IF***xxx*}, the corresponding {**$ENDIF**} must be found within the same source file—which means there must be an equal number of {**$IF***xxx*}'s and {**$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

```
{$DEFINE name}
{$UNDEF name}
```

You can also use the **/D** switch in the command-line compiler (or place it in the Conditional Defines input box from within **O**ptions | **C**ompiler of the IDE).

Conditional symbols are best compared to Boolean variables: They are either *True* (defined) or *False* (undefined). The {**$DEFINE**} directive sets a given symbol to *True*, and the {**$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

```
const
  Debug = True;
begin
  {$IFDEF Debug}
   Writeln('Debug is on');
  {$ENDIF}
end;
```

will *not* compile the *Writeln* statement. Likewise, the construct

```
{$DEFINE Debug}
begin
  if Debug then
   Writeln('Debug is on');
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 `{$IFDEF CPU87} {$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.

C H A P T E R

3

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.

If you need help using the command-line compiler, you can get online help by typing THELP at the 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 3.1
Command-line options

If you type TPC alone at the command line, a list of command-line compiler options appears on your screen.

Option	Description
/$A+	Align data on word boundaries
/$A-	Align data on byte boundaries
/$B+	Complete Boolean evaluation
/$B-	Short circuit Boolean evaluation
/$D+	Debugging information on
/$D-	Debugging information off
/$E+	Emulation on
/$E-	Emulation off
/$F+	Force far calls on
/$F-	Force far calls off
/$G+	286 code generation on
/$G-	286 code generation off
/$I+	I/O checking on
/$I-	I/O checking off
/$L+	Local symbols on
/$L-	Local symbols off
/$M*stack,min,max*	Memory sizes
/$N+	Numeric coprocessor on
/$N-	Numeric coprocessor off
/$O+	Overlay code generation on
/$O-	Overlay code generation off
/$P+	Open parameters on
/$P-	Open parameters off
/$Q+	Overflow checking on
/$Q-	Overflow checking off
/$R+	Range checking on
/$R-	Range checking off
/$S+	Stack checking on
/$S-	Stack checking off
/$T+	Type-checked pointers on
/$T-	Type-checked pointers off
/$V+	Strict var-string checking
/$V-	Relaxed var-string checking
/$X+	Extended syntax support on

Table 3.1: Command-line options (continued)

/$X-	Extended syntax support off
/B	Build all units
/D*defines*	Define conditional symbol
/E*path*	EXE and TPU directory
/F*segment:offset*	Find run-time error
/GS	Map file with segment
/GP	Map file with publics
/GD	Detailed map file
/I*path*	Include directories
/L	Link buffer on disk
/M	Make modified units
/O*path*	Object directories
/Q	Quiet compile
/T*path*	TPL & CFG directories
/U*path*	Unit directories
/V	EXE debug information

Compiler directive options

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 {**$R+**} or {**$R-**} 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.*

```
TPC MYSTUFF /$R-,I-,V-,F+
```

In addition to changing switch directives, **/$** also lets you specify a program's memory allocation parameters, using the following format:

```
/$Mstacksize,heapmin,heapmax
```

where *stacksize* is the stack size, *heapmin* is the minimum heap size, and *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:

```
TPC MYSTUFF /$M16384,256,4096
TPC MYSTUFF /$M$4000,$100,$1000
```

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

```
TPC MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, *iocheck*, *debug*, and *list*, for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple **/D** directives, you can concatenate the symbol lists. Therefore,

```
TPC MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

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

*You can't use **/M** and **/B** at the same time.*

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 /**F***segment*:*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:

```
program Test;
var
  x : Real;
begin
  x := 0;
  x := x / x;                        { Force a divide by zero error }
end.
```

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 {**$D-**} compiler directive or a **/$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:\TP\INCLUDE;D:\INC
```

If you specify multiple directives, the directory lists are concatenated. Therefore,

```
TPC MYSTUFF /IC:\TP\INCLUDE /ID:\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:\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 {**$I** *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 {**$L** *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

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:

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.

- 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 {**$D+,L+**} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the {**$D+,L-**} state, only symbols defined in a unit's **interface** part are listed in the Publics section. For modules compiled in the {**$D-**} state, there are no entries in the Line Numbers section.

The debugging (/V) option

When you specify the **/V** 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 **/V** 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 **$D** and **$L** compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the {**$D+,L+**} state, which is the default, *all* constant, variable, type, procedure, and function symbols become known to the debugger. In the {**$D+,L-**} state, only symbols defined in a unit's **interface** section become known to the debugger. In the {**$D-**} 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
```

C H A P T E R

4

Error messages

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 4.1
Error message types

Type of message	Page
Compiler	See page 238.
DOS	See page 258.
I/O	See page 260.
Critical	See page 261.
Fatal	See page 261.

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 **HIGH** and **LOW** operators with relocatable symbols or if you refer to relocatable symbols in **DB** directives.
- An operand refers to a relocatable symbol that was not defined in the **CODE** segment or in the **DATA** segment.
- An operand refers to an **EXTRN** procedure or function with an offset, for example, **CALL** *SortProc+8*.

57 THEN expected.

The reserved word **then** does not appear where it should.

58 TO or DOWNTO expected.

The reserved word **to** or **downto** does not appear where it should.

59 Undefined forward.

- The procedure or function was declared in the **interface** part of a unit, but its definition never occurred in the **implementation** part.
- The procedure or function was declared with **forward**, but its definition was never found.

61 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 Division by zero.

The preceding operand attempts to divide by zero.

63 Invalid file type.

The file type is not supported by the file-handling procedure; for example, *Readln* with a typed file or *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 **C**ompile | **M**ake or **C**ompile | **B**uild 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 ENDIF directive missing.

The source file ended within a conditional compilation construct. There must be an equal number of {**$IF*xxx***}s and {**$ENDIF**}s in a source file.

130 Error in initial conditional defines.

The initial conditional symbols specified in **O**ptions | **C**ompiler | **C**onditional 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.

Integrated debugger only

Turbo Pascal expects either an operator or the end of the expression at this point, but found neither.

135 Invalid format specifier.

Integrated debugger only

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.

Integrated debugger only

Your TURBO.TPL library must contain the *System* unit for the debugger to be able to evaluate expressions.

139 Cannot access this symbol.

Integrated debugger only

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 {**$I+**} state. In the {**$I-**} state, the program continues to execute, and the error is reported by the *IOResult* function.

100 Disk read error.

Reported by *Read* on a typed file if you attempt to read past the end of the file.

101 Disk write error.

Reported by *Close, Write, Writeln,* or *Flush* if the disk becomes full.

102 File not assigned.

Reported by *Reset, Rewrite, Append, Rename,* and *Erase* if the file variable has not been assigned a name through a call to *Assign.*

103 File not open.

Reported by *Close, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead,* or *BlockWrite* if the file is not open.

104 File not open for input.

Reported by *Read, Readln, Eof, Eoln, SeekEof,* or *SeekEoln* on a text file if the file is not open for input.

105 File not open for output.

Reported by *Write* and *Writeln* on a text file if the file is not open for output.

106 Invalid numeric format.

Reported by *Read* or *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 {**$R+**} 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 {**$S+**} state when there is not enough stack space to allocate the subprogram's local variables. Increase the size of the stack by using the **$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 unmasks 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 *Sqrt* function was negative.
- The argument passed to the *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 *OvrInit*, or the call to *OvrInit* failed. Note that, it you have initialization code in any of your overlaid units, you must create an additional non-overlaid unit which calls *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 *Abstract* procedure in the *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 *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 *Compare* method of the *TSortedCollection* type in the *Objects* unit is abstract, indicating that to implement a sorted collection you must create an object type that inherits from *TSortedCollection* and overrides the *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.

A P P E N D I X

A

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

*A word is defined as a sequence of characters separated by one of the following: space < > , ; . ( ) [ ] ^ ' * + − / $ # = | ~ ? ! " % & ' : @ \, and all control and graphic characters.*

Command	**Keys**	**Alternate Keys**
Cursor movement commands		
Character left	←	*Ctrl+S*
Character right	→	*Ctrl+D*
Word left	*Ctrl+* ←	*Ctrl+A*
Word right	*Ctrl+* →	*Ctrl+F*
Line up	↑	*Ctrl+E*
Line down	↓	*Ctrl+X*
Scroll up one line	*Ctrl+W*	
Scroll down one line	*Ctrl+Z*	
Page up	*PgUp*	*Ctrl+R*
Page down	*PgDn*	*Ctrl+C*
Beginning of line	*Home*	
	Ctrl+Q S	
End of line	*End*	
	Ctrl+Q D	
Top of window	*Ctrl+Q E*	*Ctrl+Home*
Bottom of window	*Ctrl+Q X*	*Ctrl+End*
Top of file	*Ctrl+Q R*	*Ctrl+PgUp*
Bottom of file	*Ctrl+Q C*	*Ctrl+PgDn*
Move to previous position	*Ctrl+Q P*	
Insert and delete commands		
Delete character	*Del*	*Ctrl+G*
Delete character to left	*Backspace*	*Ctrl+H*
	Shift+Tab	
Delete line	*Ctrl+Y*	
Delete to end of line	*Ctrl+Q Y*	
Delete to end of word	*Ctrl+T*	
Insert newline	*Ctrl+N*	
Insert mode on/off	*Ins*	*Ctrl+V*
Block commands		
Move to beginning of block	*Ctrl+Q B*	
Move to end of block	*Ctrl+Q K*	
Set beginning of block §	*Ctrl+K B*	
Set end of block §	*Ctrl+K K*	
Exit to menu bar	*Ctrl+K D*	
Hide/Show block §	*Ctrl+K H*	
Mark line	*Ctrl+K L*	
Print selected block	*Ctrl+K P*	
Mark word	*Ctrl+K T*	
Delete block	*Ctrl+K Y*	

* *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)

Command	Keys	Alternate Keys
Copy block §	*Ctrl+K C*	
Move block §	*Ctrl+K V*	
Copy to Clipboard ‡	*Ctrl+Ins*	
Cut to Clipboard ‡	*Shift+Del*	
Delete block ‡	*Ctrl+Del*	
Indent block	*Ctrl+K I*	
Paste from Clipboard ‡	*Shift+Ins*	
Read block from disk ‡	*Ctrl+K R*	
Unindent block	*Ctrl+K U*	
Write block to disk ‡	*Ctrl+K W*	
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*	*Shift+Ctrl+PgDn*
Beginning of file	*Shift+Ctrl+Home*	*Shift+Ctrl+PgUp*
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 [,* *Ctrl+Q]*	
Save file #	*Ctrl+K S*	*F2*
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)

Command	Keys	Alternate Keys
Set marker #	*Ctrl+K n* *	
Tabs mode on/off #	*Ctrl+O T*	
Topic search help	*Ctrl+F1*	
Undo	*Alt+Backspace*	
Unindent mode on/off #	*Ctrl+O U*	
Display compiler directives	*Ctrl+O O*	

* *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

Command	Keys	Function
Copy to Clipboard and Paste from Clipboard	*Ctrl+Ins, Shift+Ins*	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.
Copy to Clipboard	*Ctrl+Ins*	Copies selected text to the Clipboard.
Cut to Clipboard	*Shift+Del*	Cuts selected text to the Clipboard.
Delete block	*Ctrl+Del*	Deletes a selected block. You can "undelete" a block with Undo.
Cut to Clipboard and Paste from Clipboard	*Shift+Del, Shift+Ins*	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.
Paste from Clipboard	*Shift+Ins*	Pastes the contents of the Clipboard.
Read block from disk	*Ctrl+K R*	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.
Write block to disk	*Ctrl+K W*	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.

If you have used Borland editors in the past, you might prefer to use the block commands listed in the following table.

Table A.3 Borland-style block commands

Selected text is highlighted only if both the beginning and end have been set and the beginning comes before the end.

Command	**Keys**	**Function**
Set beginning of block	*Ctrl+K B*	Begin selection of text.
Set end of block	*Ctrl+K K*	End selection of text.
Hide/show block	*Ctrl+K H*	Alternately displays and hides selected text.
Copy block	*Ctrl+K C*	Copies the selected text to the position of the cursor. Useful only with the Persistent Block option.
Move block	*Ctrl+K V*	Moves the selected text to the position of the cursor. Useful only with the Persistent Block option.

Editor commands in depth

The next table describes certain editing commands in more detail. The table is arranged alphabetically by command name.

Table A.4: Other editor commands in depth

Command	**Keys**	**Function**
Autoindent mode on/off	*Ctrl+O I*	Toggles the automatic indenting of successive lines. You can also use Options I Editor Autoindent in the IDE to turn automatic indenting on and off.
Cursor through tabs on/off	*Ctrl+O R*	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+O R* is a toggle.
Find place marker	*Ctrl+Q n*[*]	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.
Open file	*F3*	Lets you load an existing file into an edit window.
Optimal fill mode on/off	*Ctrl+O F*	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.
Save file	*F2*	Saves the file and returns to the editor.

[*] *n* represents a number from 0 to 9.

Table A.4: Other editor commands in depth (continued)

Command	Keys	Function
Set marker	*Ctrl+K n*[*]	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.
Tabs mode on/off	*Ctrl+O T*	Toggles Tab mode. You can specify the use of true tab characters in the IDE with the Options | Editor Use Tab Character option.
Unindent mode on/off	*Ctrl+O U*	Toggles Unindent. You can turn Unindent on and off from the IDE with the Options | Editor Backspace Unindents option.

[*] *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

Expression	Function
^	A circumflex at the start of the string matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any character.
*****	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*.
+	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*.
[ ]	Characters in brackets match any one character that appears in the brackets but no others. For example *[bot]* matches *b*, *o*, or *t*.
[^]	A circumflex at the start of the string in brackets means *not*. Hence, *[^bot]* matches any character except *b*, *o*, or *t*.
[-]	A hyphen within the brackets signifies a range of characters. For example, *[b-o]* matches any character from *b* through *o*.
\	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.

A P P E N D I X

B

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

Directive	Source Code Syntax	Default	Command-line	Description
Align data word	{**$A+**}	*	/**$A+**	Aligns variables and typed constants on word boundaries.
Align data byte	{**$A-**}		/**$A-**	Aligns variables and typed constants on byte boundaries.
Boolean evaluation – complete	{**$B+**}		/**$B+**	Complete Boolean expression evaluation.
Boolean evaluation – short circuit	{**$B-**}	*	/**$B-**	Short circuit Boolean expression evaluation.
Debug information on	{**$D+**}	*	/**$D+**	Generates debug information.
Debug information off	{**$D-**}		/**$D-**	Turns off debug information.

Table B.1: Compiler directives (continued)

Directive	Source Code Syntax	Default	Command-line	Description
DEFINE	{**DEFINE** *name*}		/**D***name*	Defines a conditional symbol of *name*.
ELSE	{**ELSE**}			Switches between compiling and ignoring source delimited by {**$IF***xxx*} and {**$ENDIF**}.
Emulation on	{**$E+**}	*	**/$E+**	Enables linking with a run-time library that emulates the 80x87 numeric coprocessor.
Emulation off	{**$E-**}		**/$E-**	Disables linking with a run-time library that emulates the 80x87 numeric coprocessor.
ENDIF	{**$ENDIF**}			Ends conditional compilation started by last {**$IF***xxx*}.
Extended syntax[1]				
Force far calls on	{**$F+**}		/**$F+**	Procedures and functions compiled always use far call model.
Force far calls off	{**$F-**}	*	/**$F-**	Compiler selects appropriate model: far or near.
80286 code generation on	{**$G+**}		/**$G+**	Generates 80286 instructions to improve code generation.
80286 code generation off	{**$G-**}	*	/**$G-**	Generates only generic 8086 instructions.
Input/output checking on	{**$I+**}	*	/**$I+**	Enables the automatic code generation that checks the result of a call to an I/O procedure.
Input/output checking off	{**$I-**}		/**$I-**	Disables the automatic code generation that checks the result of a call to an I/O procedure.
Include file	{**$I** *filename*}			Includes the named file in the compilation.
IFDEF	{**IFDEF** *name*}			Compiles source text that follows if *name* is defined.
IFNDEF	{**IFNDEF** *name*}			Compiles the source text that follows if *name* is *not* defined.

[1] See **$X** on page 274.

Table B.1: Compiler directives (continued)

Directive	Source Code Syntax	Default	Command-line	Description
IFOPT	{**IFOPT** *switch*}			Compiles the source text that follows if *switch* is currently in the specified state.
Link object file	{**$L** *filename*}			Links the named object file with the program or unit being compiled.
Local symbol information on	{**$L+**}	*	/**$L+**	Generates local symbol information.
Local symbol information off	{**$L-**}		/**$L-**	Disables generation of local symbol information.
Memory allocation sizes	{**$M**}*stacksize, heapmin,heapmax*		/**$M***stacksize, heapmin,heapmax*	Specifies an application or library's memory allocation parameters.
Numeric coprocessor on	{**$N+**}		/**$N+**	Generates code that performs all real-type calculations using 80x87.
Numeric coprocessor off	{**$N-**}	*	/**$N-**	Generates code that performs all real-type calculations by calling run-time library routines.
Open parameters on	{**$P+**}		/**$P+**	Enables open string and array parameters in procedure and function declarations.
Open parameters off	{**$P-**}	*	/**$P-**	Disables open string and array parameters.
Overflow checking on	{**$Q+**}		/**$Q+**	Enables the generation of overflow-checking code.
Overflow checking off	{**$Q-**}	*	/**$Q-**	Disables the generation of overflow-checking code.
Overlay code generation	{**$O+**}		/**$O+**	Enables overlay code generation
Overlay code generation	{**$O-**}	*	/**$O-**	Disables overlay code generation.
Range checking on	{**$R+**}		/**$R+**	Generates range-checking code.
Range checking off	{**$R-**}	*	/**$R-**	Disables generation of range-checking code.
Stack-overflow checking on	{**$S+**}	*	/**$S+**	Generates stack-overflow checking code.

Table B.1: Compiler directives (continued)

Directive	Source Code Syntax	Default	Command-line	Description
Stack-overflow checking off	**{$S-}**		**/$S-**	Disables generation of stack-overflow code.
Type-checked pointers on	**{$T+}**		**/$T+**	Enables the generation of type-checked pointers when the @ operator is used.
Type-checked pointers off	**{$T-}**	*	**/$T-**	Disables the generation of type-checked pointers when the @ operator is used.
UNDEF	{**UNDEF** *name*}			Undefines a previously defined conditional symbol.
Var-string checking on	**{$V+}**	*	**/$V+**	Strict type checking is enabled.
Var-string checking off	**{$V-}**		**/$V-**	Type checking is relaxed.
Extended syntax on	**{$X+}**	*	**/$X+**	Enables extended syntax to permit discarding result of a function call and to support null-terminated strings.
Extended syntax off	**{$X-}**		**/$X-**	Disables extended syntax.

A P P E N D I X

C

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 C.1
Turbo Pascal reserved words

and	**end**	**mod**	**shl**
array	**file**	**nil**	**shr**
asm	**for**	**not**	**string**
begin	**function**	**object**	**then**
case	**goto**	**of**	**to**
const	**if**	**or**	**type**
constructor	**implementation**	**packed**	**unit**
destructor	**in**	**procedure**	**until**
div	**inherited**	**program**	**uses**
do	**inline**	**record**	**var**
downto	**interface**	**repeat**	**while**
else	**label**	**set**	**with**
			xor

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 C.2
Turbo Pascal standard directives

absolute	**far**	**near**	**resident**
assembler	**forward**	**private**	**virtual**
external	**interrupt**	**public**	

private and **public** act as reserved words within object type declarations, but are otherwise treated as directives.

A P P E N D I X

D

ASCII characters

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 ^@ means to press the Ctrl key and type @.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	^@ NUL	32	20		64	40	@	96	60	'
1	1	☺ SOH	33	21	!	65	41	A	97	61	a
2	2	☻ STX	34	22	"	66	42	B	98	62	b
3	3	♥ ETX	35	23	#	67	43	C	99	63	c
4	4	♦ EOT	36	24	$	68	44	D	100	64	d
5	5	♣ ENQ	37	25	%	69	45	E	101	65	e
6	6	♠ ACK	38	26	&	70	46	F	102	66	f
7	7	• BEL	39	27	'	71	47	G	103	67	g
8	8	◘ BS	40	28	(	72	48	H	104	68	h
9	9	○ TAB	41	29	)	73	49	I	105	69	i
10	A	◙ LF	42	2A	*	74	4A	J	106	6A	j
11	B	♂ VT	43	2B	+	75	4B	K	107	6B	k
12	C	♀ FF	44	2C	,	76	4C	L	108	6C	l
13	D	♪ CR	45	2D	-	77	4D	M	109	6D	m
14	E	♫ SO	46	2E	.	78	4E	N	110	6E	n
15	F	☼ SI	47	2F	/	79	4F	O	111	6F	o
16	10	► DLE	48	30	0	80	50	P	112	70	p
17	11	◄ DC1	49	31	1	81	51	Q	113	71	q
18	12	↕ DC2	50	32	2	82	52	R	114	72	r
19	13	‼ DC3	51	33	3	83	53	S	115	73	s
20	14	¶ DC4	52	34	4	84	54	T	116	74	t
21	15	§ NAK	53	35	5	85	55	U	117	75	u
22	16	■ SYN	54	36	6	86	56	V	118	76	v
23	17	↨ ETB	55	37	7	87	57	W	119	77	w
24	18	↑ CAN	56	38	8	88	58	X	120	78	x
25	19	↓ EM	57	39	9	89	59	Y	121	79	y
26	1A	→ SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	← ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	∟ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	↔ GS	61	3D	=	93	5D	]	125	7D	}
30	1E	▲ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	▼ US	63	3F	?	95	5F	_	127	7F	_

Table D.1: ASCII table (continued)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	•
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

I N D E X

7.0

TURBO PASCAL®

BORLAND