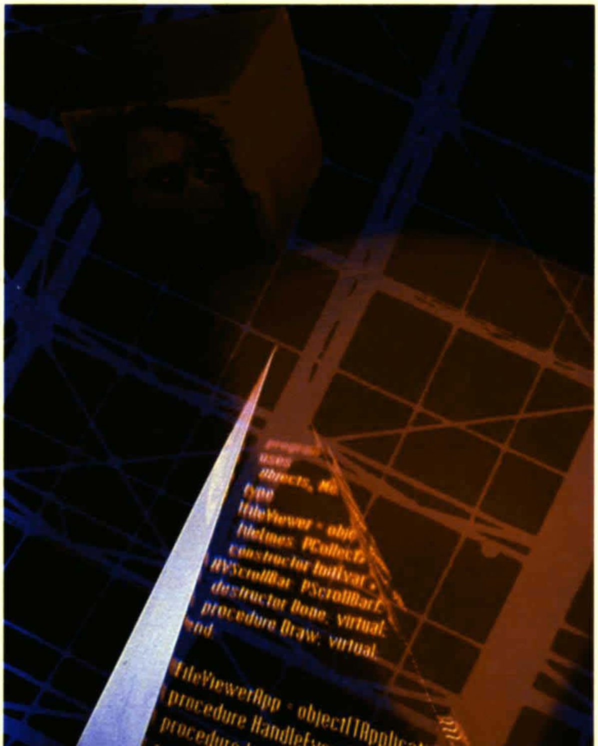
TURBO PASCAL®

6.0

LIBRARY REFERENCE



```
program
uses
  Objects, Ms
type
  FileViewer = obje
    FileLines: PCollect
    constructor Init(var
    HScrollBar: PScrollBar;
    destructor Done; virtual;
    procedure Draw; virtual;
  end;
  FileViewerApp = object(TApplicat
    procedure HandleEv
    procedure
```

BORLAND

Turbo Pascal®

Version 6.0

Library Reference

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

PRINTED IN THE USA.
R2
10 9 8 7 6 5 4 3 2 1

CONTENTS

TABLES

INTRODUCTION

This manual contains definitions of all the Turbo Pascal library routines, along with example program code to illustrate how to use most of these procedures and functions.

The User's Guide tells you how to use this product; the Library Reference and the Programmer's Guide focus on Pascal and programming issues.

If you are new to Pascal programming, you should first read the *User's Guide*. The introduction to that book details the many features of Turbo Pascal and summarizes the contents of all four volumes in this manual set. In the *User's Guide* you'll also find reference information on the IDE, the project manager, the editor, and the command-line compilers.

The *Programmer's Guide* summarizes Turbo Pascal's implementation of the Pascal language and discusses some advanced programming topics. Run-time and compile-time error messages are in Appendix A, "Error messages."

What's in this manual

Chapter 1: Run-time library is an alphabetical reference of all Turbo Pascal library procedures and functions. Each entry gives syntax, an operative description, return values if necessary, together with a reference list of related routines and an example that demonstrates how the routines are used.

CHAPTER

1

The run-time library

This chapter contains a detailed description of all the procedures and functions in Turbo Pascal. The following sample library lookup entry explains where to look for details about each Turbo Pascal procedure and function.

Sample procedure — Unit it occupies

Function	What it does
Declaration	How it's declared; italicized items are user-defined
Result type	What it returns if it's a function
Remarks	General information about the procedure or function
Restrictions	Special requirements or items to watch for
See also	Related procedures and functions
Example	{ Here you'll find a sample program that shows the use of the procedure or function in that entry. }

Abs function

Function Returns the absolute value of the argument.

Declaration `Abs(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 absolute value of *X*.

Example

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

Addr function

Function Returns the address of a specified object.

Declaration `Addr(X)`

Result type 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.

⇨ The @ operator produces the same result as *Addr*.

See also *Ofs, Ptr, Seg*

Example

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

Append procedure

Function Opens an existing file for appending.

Declaration `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*. It is an error if there is no existing external file of the given name. If *F* was already open, it is first closed and then re-opened. 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* will refer 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 a 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.
```

Arc procedure Graph

Function Draws a circular arc from start angle to end angle, using (*X*, *Y*) as the center point.

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

Remarks Draws a circular arc around (*X*, *Y*), with a radius of *Radius*. The *Arc* travels from *StAngle* to *EndAngle* and is drawn in the current drawing color.

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 a complete circle. The angles for *Arc*, *Ellipse*, and *PieSlice* are counter-clockwise 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 with a call to *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

Function Returns the arctangent of the argument.

Declaration `ArcTan(x: Real)`

Result type 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

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

Declaration `Assign(var F; Name: String)`

Remarks *F* is a file variable of any file type, and *Name* is a string-type expression. All further operations on *F* will 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* will refer to the standard input file, and after a call to *Rewrite*(*F*), *F* will refer to the standard output file.

Restrictions *Assign* must never be used on an open file.

See also *Append, Close, 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

Function Associates a text file with the CRT.

Declaration `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.
```

Bar procedure — Graph

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

Declaration `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

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

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

Remarks Draws a filled-in, three-dimensional bar. Uses 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 number of pixels deep of the 3-D outline. If *Top* is True, a 3-D top is put on the bar; if *Top* is False, no top is put on the bar (making it possible to stack several bars on top 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);
```

The following constants are defined:

```
const
  TopOn  = True;
  TopOff = False;
```

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.
```

BlockRead procedure

Function Reads one or more records into a variable.

Declaration `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 less 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 will occur if the number read 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 it was omitted). It's an error if *Count* * *RecSize* is greater than 65,535 (64K).

Result is an optional parameter. Here is how it works: 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 one, *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 the *BlockRead*.

With **{$I-}**, *IOResult* returns a 0 if the operation was successful; 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

Function Writes one or more records from a variable.

Declaration `BlockWrite(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 less 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 will occur 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 it was omitted). It is an error if *Count* * *RecSize* is greater than 65,535 (64K).

Result is an optional parameter. Here is how it works: 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 one, *Result* returns the number of complete records written; that is, it's possible a remaining partial record is not included in *Result*.

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

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

Restrictions File must be open.

See also *BlockRead*

Example See example for *BlockRead*.

ChDir procedure

Function Changes the current directory.

Declaration `ChDir(S: String)`

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

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

See also *GetDir, MkDir, RmDir*

Example

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

Chr function

Function Returns a character with a specified ordinal number.

Declaration `Chr(X: Byte)`

Result type Char

Remarks *X* is an integer-type expression. The result is the character with an ordinal value (ASCII value) of X.

See also *Ord*

Example

```
uses Printer;
begin
  Writeln(Lst, Chr(12));                          { Send formfeed to printer }
end.
```

Circle procedure

Graph

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

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

Remarks The circle is drawn in the current color set by *SetColor*. Each graphics driver contains an aspect ratio that is used by *Circle*, *Arc*, and *PieSlice* to make circles.

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

Function Clears the graphics screen and prepares it for output.

Declaration `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

Function Clears the current viewport.

Declaration `ClearViewPort`

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

Restrictions Must be in graphics mode.

See also *Bar, 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.
```

Close procedure

Function Closes an open file.

Declaration `Close(var F)`

Remarks *F* is a file variable of any file type that was previously opened with *Reset, Rewrite,* or *Append*. The external file associated with *F* is completely updated and then closed, and its DOS file handle is freed for reuse.

With **{$I-}**, *IOResult* returns a 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

Function Shuts down the graphics system.

Declaration `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

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

Declaration `ClrEol`

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

This procedure is window-relative and will clear from the current 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

Function Clears the active window and places the cursor in the upper left-hand corner.

Declaration `ClrScr`

Remarks All character positions are set 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,* as well as empty lines created by scrolling.

This procedure is window-relative and will 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.
```

Concat function

Function Concatenates a sequence of strings.

Declaration `Concat(S1 [ , S2, ..., SN ]: String)`

Result type 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 results 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.
```

Copy function

Function Returns a substring of a string.

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

Result type 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*, an empty string is returned. 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

Function Returns the cosine of the argument.

Declaration `Cos(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the cosine of *X*. *X* is assumed to represent an angle in radians.

See also *ArcTan, Sin*

Example

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

CSeg function

Function Returns the current value of the CS register.

Declaration `CSeg`

Result type Word

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

See also *DSeg, SSeg*

Dec procedure

Function Decrements a variable.

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

Remarks *X* is an ordinal-type variable, 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

Function Delays a specified number of milliseconds.

Declaration

```
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

Function Deletes a substring from a string.

Declaration

```
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

Function Deletes the line containing the cursor.

Declaration

```
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.

This procedure is window-relative and 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

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

Declaration `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 –2 (*grNotDetected*).

The following constants are defined:

```
const
  Detect   =  0;                              { Request autodetection }
  CGA      =  1;
  MCGA     =  2;
  EGA      =  3;
  EGA64    =  4;
  EGAMono  =  5;
  IBM8514  =  6;
  HercMono =  7;
  ATT400   =  8;
  VGA      =  9;
  PC3270   = 10;
```

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. Note that 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, 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.
```

DiskFree function Dos

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

Declaration `DiskFree(Drive: Byte)`

Result type 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;
begin
  Writeln(DiskFree(0) div 1024, ' Kbytes free ');
end.
```

DiskSize function Dos

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

Declaration `DiskSize(Drive: Byte)`

Result type 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;
begin
  Writeln(DiskSize(0) div 1024, ' Kbytes capacity');
end.
```

Dispose procedure

Function Disposes a dynamic variable.

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

Remarks *P* is a pointer variable of any pointer type that was previously assigned by the *New* procedure or was 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 has been extended to allow 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 *Destruct* 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.

Dispose and *FreeMem* cannot be used interchangeably with *Mark* and *Release* unless certain rules are observed. For a complete discussion of this topic, see "The heap manager" in Chapter 16 in the *Programmer's Guide.*

See also *FreeMem, GetMem, Mark, New, Release*

Example

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

DosExitCode function Dos

Function Returns the exit code of a subprocess.

Declaration `DosExitCode`

Result type 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

Function Returns the DOS version number.

Declaration `DosVersion`

Result type Word

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

Example

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

DrawPoly procedure — Graph

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

Declaration `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.
```

DSeg function

Function Returns the current value of the DS register.

Declaration `DSeg`

Result type Word

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

See also *CSeg, SSeg*

Ellipse procedure

Graph

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

Declaration `Ellipse(X, Y: Integer; StAngle, EndAngle: Word; XRadius, YRadius: Word)`

Remarks Draws an elliptical arc using (*X, Y*) as a center point, and *XRadius* and *YRadius* as the horizontal and vertical axes. The ellipse travels from *StAngle* to *EndAngle* and is drawn in the current color.

A start angle of 0 and an end angle of 360 will draw 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 with a call to *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

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

Declaration `EnvCount`

Result type 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, refer to 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

Function Returns a specified environment string.

Declaration `EnvStr(Index: Integer)`

Result type 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, refer to your DOS manuals.

See also *EnvCount, GetEnv*

Eof function (text files)

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

Declaration `Eof [ (var F: Text) ]`

Result type 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 a 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)

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

Declaration `Eof(var F)`

Result type 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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Eoln function

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

Declaration `Eoln [(var F: Text)]`

Result type 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.

When checking *Eoln* on standard input that has not been redirected, the following program will wait for a carriage return to be entered before returning from the call to *Eoln*:

```
begin
{ Tells program to wait for keyboard input }
  Writeln(Eoln);
end.
```

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

See also *Eof, SeekEoln*

Erase procedure

Function Erases an external file.

Declaration `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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions *Erase* must never be used 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.
```

Exec procedure Dos

Function Executes a specified program with a specified command line.

Declaration `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*; possible error codes are 2, 8, 10, and 11. 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; otherwise, there won't be enough memory (*DosError* = 8).

Restrictions Versions of the Novell Network system software earlier than 2.01 or 2.02 do not support a DOS call used by *Exec*. If you are using the IDE to run a program that uses *Exec*, and you have early Novell system software, set **Compile | Destination** to *Disk* and run your program from DOS (you can use the **File | DOS** Shell command to do this).

See also *DosExitCode, 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

Function Exits immediately from the current block.

Declaration `Exit`

Remarks When *Exit* is executed in a subroutine (procedure or function), it causes the subroutine to return. When it is executed in the statement part of a program, it 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

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

begin
  WasteTime;
end.
```

Exp function

Function Returns the exponential of the argument.

Declaration `Exp(X: Real)`

Result type 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*

FExpand function

Dos

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

Declaration `FExpand(Path: PathStr)`

Result type *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'
```

The *FSplit* procedure may be used to split the result of *FExpand* into a drive/directory string, a file-name string, and an extension string.

See also *FindFirst, FindNext, FSplit*

FilePos function

Function Returns the current file position of a file.

Declaration `FilePos(var F)`

Result type 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 a 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*

FileSize function

Function Returns the current size of a file.

Declaration `FileSize(var F)`

Result type 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 a 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.
```

FillChar procedure

Function Fills a specified number of contiguous bytes with a specified value.

Declaration `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.

Whenever possible, use the *SizeOf* function to specify the count parameter. When using *FillChar* on strings, remember to set the length byte after the fill.

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

Function Draws a filled ellipse.

Declaration `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.
```

FillPoly procedure — Graph

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

Declaration `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.
```

FindFirst procedure Dos

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

Declaration `FindFirst(Path: String; Attr: Word; var S: SearchRec)`

Remarks *Path* is the directory mask (for example, `*.*`). The *Attr* parameter specifies the special files to include (in addition to all normal files). Here are the file attributes as they are declared in the *Dos* unit:

```
const
  ReadOnly  = $01;
  Hidden    = $02;
  SysFile   = $04;
  VolumeID  = $08;
  Directory = $10;
  Archive   = $20;
  AnyFile   = $3F;
```

The result of the directory search is returned in the specified search record. *SearchRec* is declared in the *Dos* unit:

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

Errors are reported in *DosError;* possible error codes are 3 ("Directory Not Found") and 18 ("No More Files").

See also *FExpand, FindNext*

Example

```
uses Dos;
var
  DirInfo: SearchRec;

begin
  FindFirst('*.PAS', Archive, DirInfo);                          { Same as DIR *.PAS }
  while DosError = 0 do
  begin
    Writeln(DirInfo.Name);
    FindNext(DirInfo);
  end;
end.
```

FindNext procedure

Dos

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

Declaration `FindNext(var S: SearchRec)`

Remarks *S* must be the same one Passed to *FindFirst* (*SearchRec* is declared in *Dos* unit; see *FindFirst*). Errors are reported in *DosError;* the only possible error code is 18, which indicates no more files.

See also *FindFirst, FExpand*

Example See the example for *FindFirst.*

FloodFill procedure

Graph

Function Fills a bounded region with the current fill pattern.

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

Remarks This procedure is called to fill 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 *–7* (*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 that 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

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

Declaration `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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Frac function

Function Returns the fractional part of the argument.

Declaration `Frac(X: Real)`

Result type 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.
```

FreeMem procedure

Function Disposes a dynamic variable of a given size.

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

Remarks *P* is a pointer variable of any pointer type that was previously assigned by the *GetMem* procedure or was assigned a meaningful value by an assignment statement. *Size* is an expression of type Word, specifying the size in bytes of the dynamic variable to dispose; 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 it is an error to subsequently reference *P^*.

Restrictions *Dispose* and *FreeMem* cannot be used interchangeably with *Mark* and *Release* unless certain rules are observed. For a complete discussion of this topic, see "The heap manager" in Chapter 16 of the *Programmer's Guide*.

See also *Dispose, GetMem, Mark, New, Release*

FSearch function Dos

Function Searches for a file in a list of directories.

Declaration `FSearch(Path: PathStr; DirList: String)`

Result type *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.

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

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

Function Splits a file name into its three components.

Declaration `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 may possibly be empty, if *Path* contains no such component.

The *PathStr, DirStr, NameStr,* and *ExtStr* types are defined in the *Dos* unit as follows:

```
type
  PathStr = string[79];
  DirStr  = string[67];
  NameStr = string[8];
  ExtStr  = string[4];
```

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 also *FExpand, FindFirst, FindNext*

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

Function Allows the user to inquire about the coordinates of the last *Arc* command.

Declaration `GetArcCoords(var ArcCoords: ArcCoordsType)`

Remarks *GetArcCoords* returns a variable of type *ArcCoordsType. ArcCoordsType* is predeclared as follows:

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

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.
```

GetAspectRatio procedure Graph

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

Declaration `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

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

Declaration `GetBkColor`

Result type Word

Remarks Background colors can 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

Function Returns the state of *Ctrl-Break* checking in DOS.

Declaration `GetCBreak(var Break: Boolean)`

Remarks *GetCBreak* returns 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

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

Declaration `GetColor`

Result type Word

Remarks Drawing colors can 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.
```

GetDate procedure — Dos

Function Returns the current date set in the operating system.

Declaration `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

Function Returns the palette definition record.

Declaration `GetDefaultPalette (var Palette: PaletteType)`

Result type *PaletteType*

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

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

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

Function Returns the current directory of a specified drive.

Declaration `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 *per se*. If the drive specified by *D* is invalid, *S* returns '\', as if it were the root directory of the invalid drive.

See also *ChDir, DiskFree, DiskSize, MkDir, RmDir*

GetDriverName function Graph

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

Declaration `GetDriverName`

Result type 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

Function Returns the value of a specified environment variable.

Declaration `GetEnv(EnvVar: String)`

Result type String

Remarks *GetEnv* returns the value of a specified variable. 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, *GetEnv* returns an empty string.

For more information about the DOS environment, refer to 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.
```

GetFAttr procedure Dos

Function Returns the attributes of a file.

Declaration `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:

```
const
  ReadOnly  = $01;
  Hidden    = $02;
  SysFile   = $04;
  VolumeID  = $08;
  Directory = $10;
  Archive   = $20;
  AnyFile   = $3F;
```

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

- 3 (Invalid Path)
- 5 (File Access Denied)

Restrictions *F* cannot be open.

See also *GetFTime, SetFAttr, SetFTime*

Example

```
uses Dos;
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 file attribute type using flags in Dos 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

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

Declaration `GetFillPattern(var FillPattern: FillPatternType);`

Remarks *FillPatternType* is declared in the *Graph* unit:

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

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

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

Declaration `GetFillSettings(var FillInfo: FillSettingsType)`

Remarks *GetFillSettings* returns a variable of type *FillSettingsType. FillSettingsType* is predeclared as follows:

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

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

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

Declaration `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 may 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 *PackTime, SetFAttr, SetFTime, UnpackTime*

GetGraphMode function Graph

Function Returns the current graphics mode.

Declaration `GetGraphMode`

Result type 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

Graphics driver	Constant name	Value	Column x row	Palette	Pages
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

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

Declaration `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

Function Returns the address stored in a specified interrupt vector.

Declaration `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

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

Declaration `GetLineSettings(var LineInfo: LineSettingsType)`

Remarks The following type and constants are defined:

```
type
  LineSettingsType = record
    LineStyle: Word;
    Pattern: Word;
    Thickness: Word;
  end;
const
  { Line styles }
  SolidLn    = 0;
  DottedLn   = 1;
  CenterLn   = 2;
  DashedLn   = 3;
  UserBitLn  = 4;                                  { User-defined line style }
  { Line widths }
  NormWidth  = 1;
  ThickWidth = 3;
```

Restrictions Must be in graphics mode.

See also *DrawPoly, 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

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

Declaration `GetMaxColor`

Result type Word

Remarks As an example, on a 256K EGA, *GetMaxColor* will always return 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 only support draw colors of 0 or 1.

Restrictions Must be in graphics mode.

See also *SetColor*

GetMaxMode function — Graph

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

Declaration `GetMaxMode`

Result type 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 may 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

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

Declaration `GetMaxX`

Result type 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

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

Declaration `GetMaxY`

Result type 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

Function Creates a new dynamic variable of the specified size, and puts the address of the block in a pointer variable.

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

Remarks *P* is a pointer variable of any pointer type. *Size* is an expression of type Word 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 16 of the *Programmer's Guide.*)

Restrictions The largest block that can be allocated on the heap at one time is 65,521 bytes (64K-$F). If the heap is not fragmented, for example at the beginning of a program, successive calls to *GetMem* returns neighboring blocks of memory.

See also *Dispose, FreeMem, Mark, New, Release*

GetModeName function Graph

Function Returns a string containing the name of the specified graphics mode.

Declaration `GetModeName(ModeNumber: Integer)`

Result type String

Remarks The mode names are embedded in each driver. The return values (320×200 CGA P1, 640×200 CGA, etc.) 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

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

Declaration `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 return parameters are set to –1.

See also *DetectGraph, GetGraphMode, InitGraph, SetGraphMode*

GetPalette procedure — Graph

Function Returns the current palette and its size.

Declaration `GetPalette(var Palette: PaletteType)`

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

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

Function Returns the the size of the palette color lookup table.

Declaration `GetPaletteSize`

Result type 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

Function Gets the pixel value at *X*, *Y*.

Declaration `GetPixel(X, Y: Integer)`

Result type Word

Remarks Gets the pixel color 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

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

Declaration `GetTextSettings(var TextInfo: TextSettingsType)`

Remarks The following type and constants are defined:

```
type
  TextSettingsType = record
    Font: Word;
    Direction: Word;
    CharSize: Word;
    Horiz: Word;
    Vert: Word;
  end;
const
```

```
      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 }
```

Restrictions Must be in graphics mode.

See also *InitGraph, SetTextJustify, SetTextStyle, TextHeight, 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

Function Returns the current time set in the operating system.

Declaration `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

Function Returns the state of the verify flag in DOS.

Declaration `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

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

Declaration `GetViewSettings(var ViewPort: ViewPortType)`

Remarks *GetViewSettings* returns a variable of type *ViewPortType*. *ViewPortType* is predeclared as follows:

```
type
  ViewPortType = record
    X1, Y1, X2, Y2: Integer;
    Clip: Boolean;
  end;
```

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

Restrictions Must be in graphics mode.

See also *ClearViewPort*, *SetViewPort*

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

Function Returns the X coordinate of the current position (CP).

Declaration `GetX`

Result type Integer

Remarks *GetX* is viewport-relative. In the following example,

1. `SetViewPort(0, 0, GetMaxX, GetMaxY, True);`
2. `MoveTo(5, 5);`
3. `SetViewPort(10, 10, 100, 100, True);`
4. `MoveTo(5, 5);`

Line 1 moves CP to absolute (0, 0), and *GetX* would also return a value of 0. Line 2 moves CP to absolute (5, 5), and *GetX* would also return a value of 5. Line 3 moves CP to absolute (10, 10), but *GetX* would return a value of 0. Line 4 moves CP to absolute (15, 15), but *GetX* would return 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

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

Declaration `GetY`

Result type Integer

Remarks *GetY* is viewport-relative. In the following example,

1. `SetViewPort(0, 0, GetMaxX, GetMaxY, True);`
2. `MoveTo(5, 5);`
3. `SetViewPort(10, 10, 100, 100, True);`
4. `MoveTo(5, 5);`

Line 1 moves CP to absolute (0, 0), and *GetY* would also return a value of 0. Line 2 moves CP to absolute (5, 5), and *GetY* would also return a value of 5. Line 3 moves CP to absolute (10, 10), but *GetY* would return a value of 0. Line 4 moves CP to absolute (15, 15), but *GetY* would return 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

Function Positions the cursor.

Declaration `GotoXY(X, Y: Byte)`

Remarks The cursor is moved 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 and will move 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

Function Resets the graphics settings.

Declaration `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

Function Returns an error message string for the specified *ErrorCode*.

Declaration `GraphErrorMsg(ErrorCode: Integer)`

Result type 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.
```

GraphResult function Graph

Function Returns an error code for the last graphics operation.

Declaration `GraphResult`

Result type Integer

Remarks Returns an error code for the last graphics operation. The following error return codes are defined:

Error code	Graphics error constant	Corresponding error message string
0	*grOk*	No error
–1	*grNoInitGraph*	(BGI) graphics not installed (use *InitGraph*)
–2	*grNotDetected*	Graphics hardware not detected
–3	*grFileNotFound*	Device driver file not found
–4	*grInvalidDriver*	Invalid device driver file
–5	*grNoLoadMem*	Not enough memory to load driver
–6	*grNoScanMem*	Out of memory in scan fill
–7	*grNoFloodMem*	Out of memory in flood fill
–8	*grFontNotFound*	Font file not found
–9	*grNoFontMem*	Not enough memory to load font
–10	*grInvalidMode*	Invalid graphics mode for selected driver
–11	*grError*	Graphics error
–12	*grIOerror*	Graphics I/O error
–13	*grInvalidFont*	Invalid font file
–14	*grInvalidFontNum*	Invalid font number

The following routines set *GraphResult*:

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

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

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

See also *GraphErrorMsg*

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.
```

Halt procedure

Function Stops program execution and returns to the operating system.

Declaration `Halt [ ( ExitCode: Word ) ]`

Remarks *ExitCode* is an optional expression of type Word that specifies the exit code of the program. *Halt* without a parameter corresponds to *Halt(0)*. The exit code can be examined by a parent process using the *DosExitCode* function in the *Dos* unit or through an *ERRORLEVEL* test in a DOS batch file.

Note that *Halt* will initiate execution of any unit *Exit* procedures (see Chapter 18 in the *Programmer's Guide*).

See also *Exit, RunError*

Hi function

Function Returns the high-order byte of the argument.

Declaration `Hi(X)`

Result type 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 W: Word;
begin
  W := Hi($1234);   { $12 }
end.
```

HighVideo procedure — Crt

Function Selects high-intensity characters.

Declaration `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

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

Declaration `ImageSize(X1, Y1, X2, Y2: Integer)`

Result type 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 three words. The first stores the width of the region, the second stores the height, and the third 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

Function Increments a variable.

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

Remarks *X* is an ordinal-type variable, 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 for use 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.
```

InitGraph procedure Graph

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

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

Remarks Both *GraphDriver* and *GraphMode* are **var** parameters.

If *GraphDriver* is equal to *Detect*(0), 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* will be set to the current graphics driver, and *GraphMode* will be set to the current graphics mode.

If an error occurred, both *GraphDriver* and *GraphResult* (a function) returns one of the following values:

–2	Cannot detect a graphics card
–3	Cannot find driver file
–4	Invalid driver

–5	Insufficient memory to load driver
–10	Invalid graphics mode for selected driver

InitGraph resets all graphics settings to their defaults (current pointer, palette, color, viewport, etc.).

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

Several useful constants are defined for each graphics driver supported:

Error code	Graphics error constant	Corresponding error message string
0	*grOk*	No error
–1	*grNoInitGraph*	(BGI) graphics not installed (use *InitGraph*)
–2	*grNotDetected*	Graphics hardware not detected
–3	*grFileNotFound*	Device driver file not found
–4	*grInvalidDriver*	Invalid device driver file
–5	*grNoLoadMem*	Not enough memory to load driver
–6	*grNoScanMem*	Out of memory in scan fill
–7	*grNoFloodMem*	Out of memory in flood fill
–8	*grFontNotFound*	Font file not found
–9	*grNoFontMem*	Not enough memory to load font
–10	*grInvalidMode*	Invalid graphics mode for selected driver
–11	*grError*	Graphics error
–12	*grIOerror*	Graphics I/O error
–13	*grInvalidFont*	Invalid font file
–14	*grInvalidFontNum*	Invalid font number

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, 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.
```

Insert procedure

Function Inserts a substring into a string.

Declaration `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

Function Inserts an empty line at the cursor position.

Declaration `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.

This procedure is window-relative and will insert a line 60 columns wide at absolute coordinates (1, 10):

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

See also *DelLine, Window*

InstallUserDriver function Graph

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

Declaration `InstallUserDriver(Name: String; AutoDetectPtr: Pointer)`

Result type Integer

Remarks *InstallUserDriver* allows you to 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 may 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 is not 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;
    { Discard SGA driver #; trust autodetection }
    InitGraph(Driver, Mode, '');
    ...
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 may 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

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

Declaration `function InstallUserFont(FontFileName: String)`

Result type 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 0 (*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

Function Returns the integer part of the argument.

Declaration `Int(X: Real)`

Result type 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

Function Executes a specified software interrupt.

Declaration `Intr(IntNo: Byte; var Regs: Registers)`

Remarks *IntNo* is the software interrupt number (0..255). *Registers* is a record defined in DOS:

```
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;
```

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, refer to the section "Interrupt handling" in Chapter 18 in the *Programmer's 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 *MsDos*

IOResult function

Function Returns an integer value that is the status of the last I/O operation performed.

Declaration `IOResult`

Result type Word

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 its internal error flag.

The codes returned are summarized in Appendix A in the *Programmer's Guide*. 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.
```

Keep procedure Dos

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

Declaration `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. Refer to the MS-DOS technical documentation for more information.

See also *DosExitCode*

KeyPressed function Crt

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

Declaration `KeyPressed`

Result type 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.
```

Length function

Function Returns the dynamic length of a string.

Declaration `Length(S: String)`

Result type Integer

Remarks *S* is a string-type expression. The result is the length of *S*.

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

Example

```
var
  F: Text;
  S: String;
begin
  Assign(F, 'GARY.PAS');
  Reset(F);
  Readln(F, S);
  Writeln('"', S, '"')
  Writeln('length = ', Length(S));
end.
```

Line procedure Graph

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

Declaration `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**'d to the screen.

Note that

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

is equivalent to

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

Use *LineTo* when the current pointer 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.
```

LineRel procedure   Graph

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

Declaration `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**'d 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.
```

LineTo procedure   Graph

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

Declaration `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**'d to the screen.

Note that

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

is equivalent to

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

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

Function Returns the natural logarithm of the argument.

Declaration `Ln(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the natural logarithm of X.

See also *Exp*

Lo function

Function Returns the low-order byte of the argument.

Declaration `Lo(X)`

Result type 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 W: Word;
begin
  W := Lo($1234);   { $34 }
end.
```

LowVideo procedure

Crt

Function Selects low-intensity characters.

Declaration `LowVideo`

Remarks There is a Byte variable in *Crt—TextAttr*—that is used to hold 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.
```

Mark procedure

Function Records the state of the heap in a pointer variable.

Declaration `Mark(var P: Pointer)`

Remarks *P* is a pointer variable of any pointer type. The current value of the heap pointer is recorded in *P*, and can later be used as an argument to *Release*.

Restrictions *Mark* and *Release* cannot be used interchangeably with *Dispose* and *FreeMem* unless certain rules are observed. For a complete discussion of this topic, see "The heap manager" in Chapter 16 of the *Programmer's Guide.*

See also *Dispose, FreeMem, GetMem, New, Release*

MaxAvail function

Function 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 `MaxAvail`

Result type Longint

Remarks This number is calculated by comparing the sizes of all free blocks below the heap pointer to the size of free memory above the heap pointer. To find the total amount of free memory on the heap, call *MemAvail.* Your program can specify minimum and maximum heap requirements using the **$M** compiler directive (see Chapter 21 in the *Programmer's Guide*).

See also *MemAvail*

Example

```
type
  FriendRec = record
    Name: string[30];
    Age: Byte;
  end;
var
  P: Pointer;
begin
  if MaxAvail < SizeOf(FriendRec) then
    Writeln('Not enough memory')
  else
  begin
    { Allocate memory on heap }
    GetMem(P, SizeOf(FriendRec));
    ...
  end;
end.
```

MemAvail function

Function Returns the sum of all free blocks in the heap.

Declaration `MemAvail`

Result type Longint

Remarks This number is calculated by adding the sizes of all free blocks below the heap pointer to the size of free memory above the heap pointer. Note that unless *Dispose* and *FreeMem* were never called, a 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.* Your program can specify minimum and maximum heap requirements using the **$M** compiler directive (see Chapter 21 in the *Programmer's Guide*).

See also *MaxAvail*

Example

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

MkDir procedure

Function Creates a subdirectory.

Declaration `MkDir(S: String)`

Remarks *S* is a string-type expression. A new subdirectory with the path specified by *S* is created. The last item in the path cannot be an existing file name.

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

See also *ChDir, GetDir, RmDir*

Example

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

Move procedure

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

Declaration `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), and they are therefore not detected.

Whenever possible, use the *SizeOf* function to determine the *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

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

Declaration `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

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

Declaration `MoveTo(X, Y: Integer)`

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

ClearDevice
ClearViewPort
GraphDefaults
InitGraph
LineRel
LineTo
MoveRel
MoveTo
OutText
SetGraphMode
SetViewPort

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

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

Example

```
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

Function Executes a DOS function call.

Declaration `MsDos(var Regs: Registers)`

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 declared in the *Dos* unit:

```
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;
```

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*

New procedure

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

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

Remarks *P* is a pointer 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 16 of the *Programmer's Guide*.)

New has been extended to allow 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 *Construct* is a call to the 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*(*P*), *P* can be any pointer type. If the call is of the form

New(P, Init), *P* must point to an object type, and *Init* must be a call to the constructor of that object type. In both cases, the type of the function result is *P*.

See also *Dispose, FreeMem, GetMem, Release*

NormVideo procedure Crt

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

Declaration `NormVideo`

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

See also *HighVideo, LowVideo, TextBackground, TextColor*

NoSound procedure Crt

Function Turns off the internal speaker.

Declaration `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

Function Tests if the argument is an odd number.

Declaration `Odd(X: Longint)`

Result type Boolean

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

Ofs function

Function Returns the offset of a specified object.

Declaration `Ofs(X)`

Result type 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*.

See also *Addr, Seg*

Ord function

Function Returns the ordinal number of an ordinal-type value.

Declaration `Ord(X)`

Result type 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*

OutText procedure Graph

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

Declaration `OutText(TextString: String)`

Remarks *TextString* is output at the current pointer using the current justification settings. *TextString* is always 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 only updated by *OutText* 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

Function Sends a string to the output device.

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

Remarks *TextString* is output at (*X, Y*). *TextString* is always 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.

OutTextXY 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.

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

Function Clears the overlay buffer.

Declaration `OvrClearBuf`

Remarks Upon a call to *OvrClearBuf,* all currently loaded overlays are disposed 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*

OvrGetBuf function — Overlay

Function Returns the current size of the overlay buffer.

Declaration `OvrGetBuf`

Result type 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. A buffer of this size is automatically allocated when an overlaid program is executed. (**Note:** The initial buffer size may be larger than 64K, since it includes both code and fix-up information for the largest overlay.)

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.
```

OvrInit procedure — Overlay

Function Initializes the overlay manager and opens the overlay file.

Declaration `OvrInit(FileName: String)`

Remarks If the file-name parameter 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), 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

Function Loads the overlay file into EMS if possible.

Declaration `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, 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;
```

OvrSetBuf procedure

Overlay

Function Sets the size of the overlay buffer.

Declaration `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*

Example

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

PackTime procedure Dos

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

Declaration `PackTime(var DT: DateTime; var Time: Longint)`

Remarks *DateTime* is a record declared in the *Dos* unit:

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

The fields of the *DateTime* record are not range-checked.

See also *GetFTime, GetTime, SetFTime, SetTime, UnpackTime*

ParamCount function

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

Declaration `ParamCount`

Result type Word

Remarks Blanks and tabs serve as separators.

See also *ParamStr*

Example

```
begin
  if ParamCount < 1 then
    Writeln('No parameters on command line')
  else
    Writeln(ParamCount, ' parameter(s)');
end.
```

ParamStr function

Function Returns a specified command-line parameter.

Declaration `ParamStr(Index)`

Result type 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 zero or greater than *ParamCount*. With DOS 3.0 or later, *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.
```

P-Q

Pi function

Function Returns the value of Pi (3.1415926535897932385).

Declaration `Pi`

Result type Real

Remarks Precision varies, depending on whether the compiler is in 8087 (80287, 80387) or software-only mode.

PieSlice procedure Graph

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

Declaration `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 –6 (*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.
```

Pos function

Function Searches for a substring in a string.

Declaration `Pos(Substr, S: String)`

Result type 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';
  { Convert spaces to zeroes }
  while Pos(' ', S) > 0 do
    S[Pos(' ', S)] := '0';
end.
```

Pred function

Function Returns the predecessor of the argument.

Declaration `Pred(X)`

Result type Same type as parameter.

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*

Ptr function

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

Declaration `Ptr(Seg, Ofs: Word)`

Result type 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 may be dereferenced and typecast:

```
if Byte(Ptr($40, $49)^) = 7 then
  Writeln('Video mode = mono');
```

See also *Addr, Ofs, Seg*

Example

```
var P: ^Byte;
begin
  P := Ptr($40, $49);
  Writeln('Current video mode is ', P^);
end.
```

PutImage procedure — Graph

Function Puts a bit image onto the screen.

Declaration `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.

The following constants are defined:

```
const
  CopyPut     = 0;     { MOV }
  XORPut      = 1;     { XOR }
  OrPut       = 2;     { OR  }
  AndPut      = 3;     { AND }
  NotPut      = 4;     { NOT }
```

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

PutPixel procedure — Graph

Function Plots a pixel at *X, Y*.

Declaration `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

Function Returns a random number.

Declaration `Random [ ( Range: Word) ]`

Result type Real or Word, depending on the parameter

Remarks If *Range* is not specified, the result is a *Real* random number within the range 0 <= *X* < 1. If *Range* is specified, it must be an expression of type Integer, and the result is a Word random number within the range 0 <= *X* < *Range*. If *Range* equals 0, a value of 0 will be returned.

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

See also *Randomize*

Example

```
uses Crt;
begin
  Randomize;
  repeat
    { Write text in random colors }
    TextAttr := Random(256);
    Write('!');
  until KeyPressed;
end.
```

Randomize procedure

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

Declaration `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*. 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 use data encryption.

See also *Random*

Read procedure (text files)

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

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

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* will start with the next character in the file.

With a type integer variable, *Read* expects a sequence of characters that form a signed number, according to the syntax shown in the section "Numbers" in Chapter 1 of the *Programmer's 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, according to the syntax shown in the section "Numbers" in Chapter 1 of the *Programmer's Guide* (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.

With **{$I-}**, *IOResult* returns a 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, since you will 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, ReadKey, Write, Writeln*

Read procedure (typed files)

Function Reads a file component into a variable.

Declaration `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. It's an error to 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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open.

See also *Write*

ReadKey function Crt

Function Reads a character from the keyboard.

Declaration `ReadKey`

Result type 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. (The extended scan codes are summarized in Appendix B of the *Programmer's Guide.*) 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

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

Declaration `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 altogether corresponds to *Readln(Input).*

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

Restrictions Works only on text files, including standard input. File must be open for input.

See also *Read*

Rectangle procedure Graph

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

Declaration `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*).

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

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

Function Registers a user-loaded or linked-in BGI driver with the graphics system.

Declaration `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.

grInvalidDriver is a possible error return, where the error code equals –4 and 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 may 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, refer to the comments at the top of the BGILINK.PAS example program on the distribution disks. For information on the BINOBJ utility, refer to the file UTIL.DOC (in ONLINE.ZIP) on your distribution disks.

It is also possible to register font files; refer to 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 –11 (*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

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

Declaration `RegisterBGIfont(Font: Pointer): Integer;`

Remarks The return value is less than 0 if an error occurs; otherwise, 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.

Here are some possible error returns:

Error code	Error identifier	Comments
–11	*grError*	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.)
–13	*grInvalidFont*	The font header is not recognized.
–14	*grInvalidFontNum*	The font number in the font header is not recognized.

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;
    { Re-select 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*.

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 may 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 may 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, refer to 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 UTIL.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; refer to the description of *RegisterBGIdriver.*

See also *InitGraph, InstallUserDriver, InstallUserFont, RegisterBGIfont, SetTextStyle*

Release procedure

Function Returns the heap to a given state.

Declaration `Release(var P: Pointer)`

Remarks *P* is a pointer variable of any pointer type that was previously assigned by the *Mark* procedure. *Release* disposes all dynamic variables that were allocated by *New* or *GetMem* since *P* was assigned by *Mark.*

Restrictions *Mark* and *Release* cannot be used interchangeably with *Dispose* and *FreeMem* unless certain rules are observed. For a complete discussion of this topic, refer to the section "The heap manager" in Chapter 16 of the *Programmer's Guide.*

See also *Dispose, FreeMem, GetMem, Mark, New*

Rename procedure

Function Renames an external file.

Declaration `Rename(var F; Newname: String)`

Remarks *F* is a file variable of any file type. *Newname* is a string-type expression. The external file associated with *F* is renamed to *Newname.* Further operations on *F* will 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 *Rename* must never be used on an open file.

See also *Erase*

Reset procedure

Function Opens an existing file.

Declaration `Reset(var F [: file; RecSize: Word)`

Remarks *F* is a file variable of any file type, which must have been 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.

Reset opens the existing external file with the name assigned to *F*. It's an error if no existing external file of the given name exists. If *F* was already open, it is first closed and then re-opened. The current file position is set to the beginning of the file.

If *F* was assigned an empty name, such as *Assign*(*F*, ''), then after the call to *Reset*, *F* will refer 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.

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.

With **{$I-}**, *IOResult* returns a 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
  {$I-}
  Assign(F, FileName);
  Reset(F);
  Close(F);
  {$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

Function Restores the screen mode to its original state before graphics was initialized.

Declaration `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

Function Creates and opens a new file.

Declaration `Rewrite(var F [: file; RecSize: Word ] )`

Remarks *F* is a file variable of any file type, which must have been 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.

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* was 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* will refer 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.

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.

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

See also *Append, Assign, 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

Function Removes an empty subdirectory.

Declaration `RmDir(S: String)`

Remarks *S* is a string-type expression. The subdirectory with the path specified by *S* is removed. If the path does not exist, is non-empty, or is the currently logged directory, an I/O error will occur.

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

See also *MkDir, ChDir, GetDir*

Example

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

Round function

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

Declaration `Round(X: Real)`

Result type 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

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

Declaration `RunError [ ( ErrorCode: Byte ) ]`

Remarks The *RunError* procedure corresponds to the *Halt* procedure except that 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 **D**ebug Information checked (turned 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 had occurred.

See also *Exit, Halt*

Example

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

Sector procedure Graph

Function Draws and fills an elliptical sector.

Declaration `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*. The sector is outlined using the current color, and filled using 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 –6 (*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

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

Declaration `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. In order to expand a file, it is possible to 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 a 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

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

Declaration `SeekEof [ (var F: Text)`

Result type 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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Can only be used on text files. File must be open.

See also *Eof, SeekEoln*

SeekEoln function

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

Declaration `SeekEoln [ (var F: Text) ]`

Result type Boolean

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 a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Can only be used on text files. File must be open.

See also *Eoln, SeekEof*

Seg function

Function Returns the segment of a specified object.

Declaration `Seg(X)`

Result type 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*

SetActivePage procedure — Graph

Function Set the active page for graphics output.

Declaration `SetActivePage(Page: Word)`

Remarks Makes *Page* the active graphics page. All graphics output will now be directed to *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 *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

Function Changes all palette colors as specified.

Declaration `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 may 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 on the screen. In the example listed here, several lines are drawn on the screen, then the palette is changed. Each time a palette color is changed, all occurrences of that color on the screen will be changed to the new color value.

The following types and constants are defined:

```
const
  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;
```

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

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 *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

Function Changes the default aspect-ratio correction factor.

Declaration `SetAspectRatio(Xasp, Yasp: Word)`

Result type Word

Remarks *SetAspectRatio* is used to change the default aspect ratio of the current graphics mode. The aspect ratio is used to draw round 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

S

Function Sets the current background color using the palette.

Declaration `SetBkColor(ColorNum: Word)`

Remarks Background colors may 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

Function Sets the state of *Ctrl-Break* checking in DOS.

Declaration `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

Function Sets the current drawing color using the palette.

Declaration `SetColor(Color: Word)`

Remarks *SetColor*(5) makes the fifth color in the palette the current drawing color. Drawing colors may 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.
```

SetDate procedure Dos

Function Sets the current date in the operating system.

Declaration `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*

S

SetFAttr procedure Dos

Function Sets the attributes of a file.

Declaration `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 attribute masks defined as constants in the *Dos* unit.

```
  const
    ReadOnly  = $01;
    Hidden    = $02;
    SysFile   = $04;
    VolumeID  = $08;
    Directory = $10;
    Archive   = $20;
```

Errors are reported in *DosError;* possible error codes are 3 (Invalid Path) and 5 (File Access Denied).

Restrictions *F* cannot be open.

See also *GetFAttr, GetFTime, SetFTime*

Example

```
uses Dos;
var
  F: file;
begin
  Assign(F, 'C:\AUTOEXEC.BAT');
  SetFAttr(F, Hidden);                                          { Uh-oh }
  Readln;
  SetFAttr(F, Archive);                                         { Whew! }
end.
```

SetFillPattern procedure Graph

Function Selects a user-defined fill pattern.

Declaration `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 –11 (*grError*), and the current fill settings will be unchanged. *FillPatternType* is predefined as follows:

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

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, 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

Function Sets the fill pattern and color.

Declaration `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 –11 (*grError*), and the current fill settings will be unchanged. The following constants are defined:

```
const
  { Fill patterns for Get/SetFillStyle: }
  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 }
```

If *Pattern* equals *UserFill,* the user-defined pattern (set by a call to *SetFillPattern*) becomes the active pattern.

Restrictions Must be in graphics mode.

See also *Bar, Bar3D, 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

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

Declaration `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 through a call to *PackTime*. Errors are reported in *DosError*; the only possible error code is 6 (Invalid File Handle).

Restrictions *F* must be open.

See also *GetFTime, PackTime, SetFAttr, UnpackTime*

SetGraphBufSize procedure — Graph

Function Allows you to change the size of the buffer used for scan and flood fills.

Declaration `SetGraphBufSize(BufSize: Word);`

Remarks The internal buffer size is set to *BufSize*, and a buffer is allocated 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, enlarging the buffer may be necessary in order to avoid a buffer overflow.

Restrictions Note that once a call to *InitGraph* has been made, calls to *SetGraphBufSize* are ignored.

See also *FloodFill, FillPoly, InitGraph*

S

SetGraphMode procedure Graph

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

Declaration `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 –10 (*grInvalidMode*).

The following constants are defined:

Graphics driver	Graphics modes	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

Graphics driver	Graphics modes	Value	Column x row	Palette	Pages
	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
514	IBM8514Lo	0	640x480	256 color	1
8514	IBM8514Hi	0	1024x768	256 color	1

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

Restrictions A successful call to *InitGraph* must have been made before calling this routine.

See also *ClearDevice, CloseGraph, DetectGraph, 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.
```

S

SetIntVec procedure Dos

Function Sets a specified interrupt vector to a specified address.

Declaration `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 $1*B* handler and later restores the original handler:

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

See also *GetIntVec*

SetLineStyle procedure Graph

Function Sets the current line width and style.

Declaration `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 –11 (*grError*), and the current line settings will be unchanged. The following constants are declared:

```
const
  SolidLn    = 0;
  DottedLn   = 1;
  CenterLn   = 2;
  DashedLn   = 3;
  UserBitLn  = 4;                                    { User-defined line style }
  NormWidth  = 1;
  ThickWidth = 3;
```

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, 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

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

Declaration `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* may 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 –11 (*grError*), and the palette will be unchanged.

Changes made to the palette are seen immediately on the screen. In the example here, several lines are drawn on the screen, then the palette is changed randomly. Each time a palette color is changed, all occurrences of that color on the screen will be changed to the new color value.

The following constants are defined:

```
const
  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;
```

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 *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
  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

Function Modifies palette entries for the IBM 8514 and VGA drivers.

Declaration `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 only be used with the IBM 8514 driver and the VGA.

See also *GetBkColor, GetColor, GetPalette, GraphResult, SetAllPalette, SetBkColor, SetColor, SetPalette*

Example

```
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}
```

S

```
    (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.
```

SetTextBuf procedure

Function Assigns an I/O buffer to a text file.

Declaration `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, will 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, and then use the file outside the procedure that declared the buffer.

Example

```
var
  F: Text;
  Ch: Char;
  Buf: array[1..10240] of Char;                              { 10K 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

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

Declaration `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 –11 (*grError*), and the current text justification settings will be unchanged.

The following constants are defined:

```
const
  LeftText   = 0;                                  { Horizontal justification }
  CenterText = 1;
  RightText  = 2;
  BottomText = 0;                                    { Vertical justification }
  CenterText = 1;                                        { Not declared twice }
  TopText    = 2;
```

Restrictions Must be in graphics mode.

See also *GetTextSettings, GraphResult, 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

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

Declaration `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 on the screen, 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:

–8	Font file not found
–9	Not enough memory to load the font selected
–11	Graphics error
–12	Graphics I/O error
–13	Invalid font file
–14	Invalid font number

The following type and constants are declared:

```
const
  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 }
```

Restrictions Must be in graphics mode.

See also *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

Function Sets the current time in the operating system.

Declaration `SetTime(Hour, Minute, Second, Sec100: Word)`

Remarks Valid parameter ranges are *Hour* 0..23, *Minute* 0..59, *Second* 0..59, and *Sec*100 (hundredths of seconds) 0..99. If the time is not valid, the request is ignored.

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

SetUserCharSize procedure Graph

Function Allows the user to vary the character width and height for stroked fonts.

Declaration `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).

You don't have to call *SetTextStyle* immediately after calling *SetUserCharSize* to make that character size take effect. Calling *SetUserCharSize* sets the current character size to the values given.

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

Function Sets the state of the verify flag in DOS.

Declaration `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), all disk writes are verified to ensure proper writing.

See also *GetVerify*

SetViewPort procedure — Graph

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

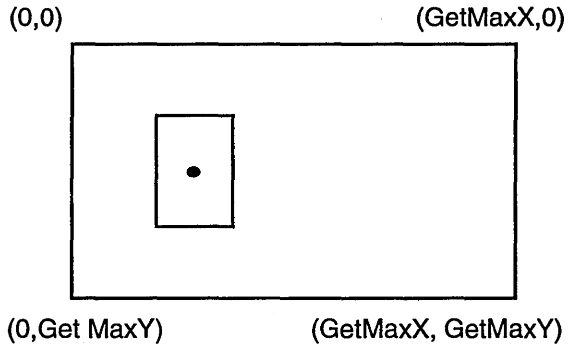
Declaration `SetViewPort(X1, Y1, X2, Y2: Integer; Clip: Boolean)`

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 variable *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 parsed to *SetViewPort*, *GraphResult* returns –11 (*grError*), and the current view settings will be unchanged. The following constants are defined:

```
const
  ClipOn  = True;
  ClipOff = False;
```

All graphics commands (for example, *GetX*, *OutText*, *Rectangle*, *MoveTo*, and so on) are viewport-relative. In the example, note that *MoveTo* moves the current pointer to (5, 5) *inside* the viewport (the absolute coordinates would be (15, 25)).



If the Boolean variable *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 the following global type:

```
type
  ViewPortType = record
    X1, Y1, X2, Y2: Integer;
    Clip: Boolean;
  end;
```

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

Restrictions Must be in graphics mode.

See also *ClearViewPort, GetViewSettings, GraphResult*

S

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

Function Sets the visual graphics page number.

Declaration `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

Function Sets the writing mode for line drawing.

Declaration `SetWriteMode(WriteMode: Integer)`

Remarks The following constants are defined:

```
const
  CopyPut = 0;      { MOV }
  XORPut  = 1;      { XOR }
```

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 *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

Function Returns the sine of the argument.

Declaration `Sin(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the sine of *X*. *X* is assumed to represent an angle in radians.

See also *ArcTan, Cos*

Example

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

SizeOf function

Function Returns the number of bytes occupied by the argument.

Declaration `SizeOf(X)`

Result type 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

Function Starts the internal speaker.

Declaration `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);                                                   { Beep }
  Delay(200);                                                  { Pause }
  NoSound;                                                   { Relief! }
end.
```

S

SPtr function

Function Returns the current value of the SP register.

Declaration `SPtr`

Result type Word

Remarks The result, of type Word, is the offset of the stack pointer within the stack segment.

See also *SSeg*

Sqr function

Function Returns the square of the argument.

Declaration `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

Function Returns the square root of the argument.

Declaration `Sqrt(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the square root of *X*.

SSeg function

Function Returns the current value of the SS register.

Declaration `SSeg`

Result type Word

Remarks The result, of type Word, is the segment address of the stack segment.

See also *SPtr, CSeg, DSeg*

Str procedure

Function Converts a numeric value to its string representation.

Declaration `Str(X [: Width [: Decimals ] ]; var S: String)`

Remarks *X* is an integer-type or real-type expression. *Width* and *Decimals* are integer-type expressions. *S* is a string-type variable. *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.
```

Succ function

Function Returns the successor of the argument.

Declaration `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 *Inc, Pred*

S

Swap function

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

Declaration `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

Function Swaps interrupt vectors.

Declaration `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*'d process does not use any interrupt handlers installed by the current process and vice versa.

See also *Exec*

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.
```

TextBackground procedure Crt

Function Selects the background color.

Declaration `TextBackground(Color: Byte);`

Remarks *Color* is an integer expression in the range 0..7, corresponding to one of the first eight color constants:

```
const
  Black     = 0;
  Blue      = 1;
  Green     = 2;
  Cyan      = 3;
  Red       = 4;
  Magenta   = 5;
  Brown     = 6;
  LightGray = 7;
```

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*

TextColor procedure Crt

Function Selects the foreground character color.

Declaration `TextColor(Color: Byte)`

Remarks *Color* is an integer expression in the range 0..15, corresponding to one of the color constants defined in *Crt*:

```
const
  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;
```

There is a byte 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*

Example

```
TextColor(Green);                                   { Green characters }
TextColor(LightRed + Blink);          { Blinking light-red characters }
TextColor(14);                                     { Yellow characters }
```

TextHeight function — Graph

Function Returns the height of a string in pixels.

Declaration `TextHeight(TextString: String)`

Result type 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

Function Selects a specific text mode.

Declaration `TextMode(Mode: Word)`

Remarks The following constants are defined:

```
const
  BW40  =  0;                        { 40x25 B/W on color adapter }
  BW80  =  2;                        { 80x25 B/W on color adapter }
  Mono  =  7;                   { 80x25 B/W on monochrome adapter }
  CO40  =  1;                      { 40x25 color on color adapter }
  CO80  =  3;                      { 80x25 color on color adapter }
  Font8x8 = 256;                     { For EGA/VGA 43 and 50 line }
  C40 = CO40;                             { For 3.0 compatibility }
  C80 = CO80;                             { For 3.0 compatibility }
```

Other values cause *TextMode* to assume C80.

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-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 on the screen (*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 *RestoreCrtMode*

TextWidth function Graph

Function Returns the width of a string in pixels.

Declaration `TextWidth(TextString: String)`

Result type 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 8x8 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;
```

T-V

```
  CloseGraph;
end.
```

Trunc function

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

Declaration `Trunc(X: Real)`

Result type 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

Function Truncates the file size at the current file position.

Declaration `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*

TypeOf function

Function Returns a pointer to an object's virtual method table.

Declaration `TypeOf(X: object)`

Result type Pointer

Remarks *X* is any object type that declares or inherits virtual methods.

Restrictions If X has no virutal methods, a compiler error occurs.

UnpackTime procedure Dos

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

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

Remarks *DateTime* is a record declared in the *Dos* unit:

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

The fields of the *Time* record are not range-checked.

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

UpCase function

Function Converts a character to uppercase.

Declaration `UpCase(Ch: Char)`

Result type 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

Function Converts the string value to its numeric representation.

Declaration `Val(S: String; var V; var Code: Integer)`

Remarks *S* is a string-type expression. *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 1 of the *Programmer's 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.

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

WhereX function Crt

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

Declaration `WhereX`

Result type Byte

See also *GotoXY, WhereY, Window*

WhereY function — Crt

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

Declaration `WhereY`

Result type Byte

See also *GotoXY, WhereX, Window*

Window procedure — Crt

Function Defines a text window on the screen.

Declaration `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 (refer to the "WindMin and WindMax" section in Chapter 15 of the *Programmer's Guide*). 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)

Function Writes one or more values to a text file.

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

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 type Char 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 1.1:

Table 1.1 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 1.2:

Table 1.2 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 a 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)

Function Writes a variable into a file component.

Declaration

```
Write(F, V1 [, V2,...,VN ] )
```

Remarks *F* is a file variable, and each *V* is a variable of the same type as the component type of *F*. For each variable written, the current file position is advanced to the next component. If the current file position is at the end of the file—that is, if *Eof(F)* is True—the file is expanded.

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

See also *Writeln*

Writeln procedure

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

Declaration

```
Writeln( [ var F: Text; ] V1 [, V2,...,VN ] )
```

Remarks *Writeln* procedure is an extension to the *Write* procedure, as it is defined for text files. After executing the Write, *Writeln* writes an end-of-line marker (carriage-return/line-feed) 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*

I N D E X

6.0

LIBRARY
REFERENCE

TURBO PASCAL®

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300
Offices in: Australia, Denmark, England, France, Germany, Italy, Japan and Sweden ■ Part# 11MN-PAS03-60 ■ BOR 1852