# CONTENTS

## Introduction

| What's in this manual | 1 |

## Chapter 1 The run-time library

<p>| Sample procedure | 3 |
| Abs function | 4 |
| Addr function | 4 |
| Append procedure | 5 |
| Arc procedure | 6 |
| ArcTan function | 6 |
| Assign procedure | 7 |
| AssignCrt procedure | 8 |
| Bar procedure | 8 |
| Bar3D procedure | 9 |
| BlockRead procedure | 10 |
| BlockWrite procedure | 11 |
| ChDir procedure | 12 |
| Chr function | 13 |
| Circle procedure | 13 |
| ClearDevice procedure | 14 |
| ClearViewport procedure | 14 |
| Close procedure | 15 |
| CloseGraph procedure | 16 |
| ClrEol procedure | 16 |
| ClrScr procedure | 17 |
| Concat function | 17 |
| Copy function | 18 |
| Cos function | 18 |
| CSeg function | 19 |
| Dec procedure | 19 |
| Delay procedure | 20 |
| Delete procedure | 20 |
| DelLine procedure | 20 |
| DetectGraph procedure | 21 |
| DiskFree function | 22 |
| DiskSize function | 22 |
| Dispose procedure | 23 |
| DosExitCode function | 24 |
| DosVersion function | 24 |
| DrawPoly procedure | 25 |
| DSeg function | 26 |
| Ellipse procedure | 26 |
| EnvCount function | 27 |
| EnvStr function | 27 |
| Eof function (text files) | 28 |
| Eof function (typed, untyped files) | 28 |
| Eoln function | 29 |
| Erase procedure | 29 |
| Exec procedure | 30 |
| Exit procedure | 31 |
| Exp function | 32 |
| FExpand function | 32 |
| FilePos function | 33 |
| FileSize function | 33 |
| FillChar procedure | 34 |
| FillEllipse procedure | 34 |
| FillPoly procedure | 35 |
| FindFirst procedure | 36 |
| FindNext procedure | 37 |
| FloodFill procedure | 37 |
| Flush procedure | 39 |
| Frac function | 39 |
| FreeMem procedure | 40 |
| FSearch function | 40 |
| FSplit procedure | 41 |
| GetArcCoords procedure | 42 |
| GetAspectRatio procedure | 43 |
| GetBackColor function | 44 |
| GetBackgroundColor function | 45 |
| GetBorderColor function | 45 |
| GetColor function | 45 |
| GetDate procedure | 46 |
| GetDefaultPalette function | 46 |</p>
<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDir procedure</td>
<td>47</td>
</tr>
<tr>
<td>GetDriverName function</td>
<td>48</td>
</tr>
<tr>
<td>GetEnv function</td>
<td>48</td>
</tr>
<tr>
<td>GetFAAttr procedure</td>
<td>49</td>
</tr>
<tr>
<td>GetFillPattern procedure</td>
<td>50</td>
</tr>
<tr>
<td>GetFillSettings procedure</td>
<td>51</td>
</tr>
<tr>
<td>GetFTime procedure</td>
<td>52</td>
</tr>
<tr>
<td>GetGraphMode function</td>
<td>52</td>
</tr>
<tr>
<td>GetImage procedure</td>
<td>54</td>
</tr>
<tr>
<td>GetIntVec procedure</td>
<td>55</td>
</tr>
<tr>
<td>GetLineSettings procedure</td>
<td>55</td>
</tr>
<tr>
<td>GetMaxColor function</td>
<td>56</td>
</tr>
<tr>
<td>GetMaxMode function</td>
<td>56</td>
</tr>
<tr>
<td>GetMaxX function</td>
<td>57</td>
</tr>
<tr>
<td>GetMaxY function</td>
<td>58</td>
</tr>
<tr>
<td>GetMem procedure</td>
<td>58</td>
</tr>
<tr>
<td>GetModeName function</td>
<td>59</td>
</tr>
<tr>
<td>GetModeRange procedure</td>
<td>60</td>
</tr>
<tr>
<td>GetPalette procedure</td>
<td>60</td>
</tr>
<tr>
<td>GetPaletteSize function</td>
<td>61</td>
</tr>
<tr>
<td>GetPixel function</td>
<td>62</td>
</tr>
<tr>
<td>GetTextSettings procedure</td>
<td>62</td>
</tr>
<tr>
<td>GetTime procedure</td>
<td>63</td>
</tr>
<tr>
<td>GetVerify procedure</td>
<td>64</td>
</tr>
<tr>
<td>GetViewSettings procedure</td>
<td>64</td>
</tr>
<tr>
<td>GetX function</td>
<td>65</td>
</tr>
<tr>
<td>GetY function</td>
<td>66</td>
</tr>
<tr>
<td>GotoXY procedure</td>
<td>67</td>
</tr>
<tr>
<td>GraphDefaults procedure</td>
<td>67</td>
</tr>
<tr>
<td>GraphErrorMsg function</td>
<td>68</td>
</tr>
<tr>
<td>GraphErrorMsg function</td>
<td>68</td>
</tr>
<tr>
<td>GraphResult function</td>
<td>68</td>
</tr>
<tr>
<td>Halt procedure</td>
<td>70</td>
</tr>
<tr>
<td>Hi function</td>
<td>70</td>
</tr>
<tr>
<td>HighVideo procedure</td>
<td>71</td>
</tr>
<tr>
<td>Inc procedure</td>
<td>71</td>
</tr>
<tr>
<td>InitGraph procedure</td>
<td>73</td>
</tr>
<tr>
<td>Insert procedure</td>
<td>75</td>
</tr>
<tr>
<td>InsLine procedure</td>
<td>75</td>
</tr>
<tr>
<td>InstallUserDriver function</td>
<td>76</td>
</tr>
<tr>
<td>InstallUserFont function</td>
<td>79</td>
</tr>
<tr>
<td>Int function</td>
<td>79</td>
</tr>
<tr>
<td>Intr procedure</td>
<td>80</td>
</tr>
<tr>
<td>IOResult function</td>
<td>80</td>
</tr>
<tr>
<td>Keep procedure</td>
<td>81</td>
</tr>
<tr>
<td>KeyPressed function</td>
<td>82</td>
</tr>
<tr>
<td>Length function</td>
<td>82</td>
</tr>
<tr>
<td>Line procedure</td>
<td>83</td>
</tr>
<tr>
<td>LineRel procedure</td>
<td>84</td>
</tr>
<tr>
<td>LineTo procedure</td>
<td>84</td>
</tr>
<tr>
<td>Ln function</td>
<td>85</td>
</tr>
<tr>
<td>Lo function</td>
<td>86</td>
</tr>
<tr>
<td>LowVideo procedure</td>
<td>86</td>
</tr>
<tr>
<td>Mark procedure</td>
<td>86</td>
</tr>
<tr>
<td>MaxAvail function</td>
<td>87</td>
</tr>
<tr>
<td>MemAvail function</td>
<td>88</td>
</tr>
<tr>
<td>MkDir procedure</td>
<td>88</td>
</tr>
<tr>
<td>Move procedure</td>
<td>89</td>
</tr>
<tr>
<td>MoveRel procedure</td>
<td>89</td>
</tr>
<tr>
<td>MoveTo procedure</td>
<td>90</td>
</tr>
<tr>
<td>MsDos procedure</td>
<td>91</td>
</tr>
<tr>
<td>New procedure</td>
<td>91</td>
</tr>
<tr>
<td>NormVideo procedure</td>
<td>92</td>
</tr>
<tr>
<td>NoSound procedure</td>
<td>92</td>
</tr>
<tr>
<td>Odd function</td>
<td>92</td>
</tr>
<tr>
<td>Ofs function</td>
<td>93</td>
</tr>
<tr>
<td>Ord function</td>
<td>93</td>
</tr>
<tr>
<td>OutText procedure</td>
<td>93</td>
</tr>
<tr>
<td>OutTextXY procedure</td>
<td>95</td>
</tr>
<tr>
<td>OvrClearBuf procedure</td>
<td>96</td>
</tr>
<tr>
<td>OvrGetBuf function</td>
<td>97</td>
</tr>
<tr>
<td>OvrInit procedure</td>
<td>97</td>
</tr>
<tr>
<td>OvrInitEMS procedure</td>
<td>98</td>
</tr>
<tr>
<td>OvrSetBuf procedure</td>
<td>99</td>
</tr>
<tr>
<td>PackTime procedure</td>
<td>100</td>
</tr>
<tr>
<td>ParamCount function</td>
<td>100</td>
</tr>
<tr>
<td>ParamStr function</td>
<td>101</td>
</tr>
<tr>
<td>Pi function</td>
<td>101</td>
</tr>
<tr>
<td>PieSlice procedure</td>
<td>102</td>
</tr>
<tr>
<td>Pos function</td>
<td>102</td>
</tr>
<tr>
<td>Pred function</td>
<td>103</td>
</tr>
<tr>
<td>Ptr function</td>
<td>103</td>
</tr>
<tr>
<td>PutImage procedure</td>
<td>104</td>
</tr>
<tr>
<td>PutPixel procedure</td>
<td>106</td>
</tr>
<tr>
<td>Random function</td>
<td>107</td>
</tr>
<tr>
<td>Randomize procedure</td>
<td>107</td>
</tr>
<tr>
<td>Read procedure (text files)</td>
<td>108</td>
</tr>
<tr>
<td>Read procedure (typed files)</td>
<td>109</td>
</tr>
<tr>
<td>Function/Procedure</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ReadKey function</td>
<td>109</td>
</tr>
<tr>
<td>Readln procedure</td>
<td>110</td>
</tr>
<tr>
<td>Rectangle procedure</td>
<td>111</td>
</tr>
<tr>
<td>RegisterBGI driver function</td>
<td>112</td>
</tr>
<tr>
<td>RegisterBGI font</td>
<td>113</td>
</tr>
<tr>
<td>Release procedure</td>
<td>116</td>
</tr>
<tr>
<td>Rename procedure</td>
<td>116</td>
</tr>
<tr>
<td>RestoreCrtMode procedure</td>
<td>118</td>
</tr>
<tr>
<td>Rewrite procedure</td>
<td>119</td>
</tr>
<tr>
<td>RmDir procedure</td>
<td>120</td>
</tr>
<tr>
<td>Round function</td>
<td>120</td>
</tr>
<tr>
<td>RunError procedure</td>
<td>121</td>
</tr>
<tr>
<td>Sector procedure</td>
<td>121</td>
</tr>
<tr>
<td>Seek procedure</td>
<td>122</td>
</tr>
<tr>
<td>SeekEof function</td>
<td>123</td>
</tr>
<tr>
<td>SeekEoln function</td>
<td>123</td>
</tr>
<tr>
<td>Seg function</td>
<td>124</td>
</tr>
<tr>
<td>setActivePage procedure</td>
<td>124</td>
</tr>
<tr>
<td>SetAspectRatio procedure</td>
<td>126</td>
</tr>
<tr>
<td>SetBkColor procedure</td>
<td>127</td>
</tr>
<tr>
<td>SetCBreak procedure</td>
<td>128</td>
</tr>
<tr>
<td>SetColor procedure</td>
<td>129</td>
</tr>
<tr>
<td>setDate procedure</td>
<td>129</td>
</tr>
<tr>
<td>SetFAttribute procedure</td>
<td>130</td>
</tr>
<tr>
<td>SetFillPattern procedure</td>
<td>130</td>
</tr>
<tr>
<td>SetFillStyle procedure</td>
<td>132</td>
</tr>
<tr>
<td>SetFTime procedure</td>
<td>133</td>
</tr>
<tr>
<td>SetGraphBufSize procedure</td>
<td>133</td>
</tr>
<tr>
<td>SetGraphMode procedure</td>
<td>134</td>
</tr>
<tr>
<td>SetIntVec procedure</td>
<td>136</td>
</tr>
<tr>
<td>SetLineStyle procedure</td>
<td>136</td>
</tr>
<tr>
<td>SetPalette procedure</td>
<td>137</td>
</tr>
<tr>
<td>SetRGBPalette procedure</td>
<td>139</td>
</tr>
<tr>
<td>SetTextBuf procedure</td>
<td>140</td>
</tr>
<tr>
<td>SetTextStyle procedure</td>
<td>143</td>
</tr>
<tr>
<td>SetTime procedure</td>
<td>144</td>
</tr>
<tr>
<td>SetUserCharSize procedure</td>
<td>145</td>
</tr>
<tr>
<td>SetVerify procedure</td>
<td>146</td>
</tr>
<tr>
<td>SetViewPort procedure</td>
<td>146</td>
</tr>
<tr>
<td>SetVisualPage procedure</td>
<td>148</td>
</tr>
<tr>
<td>SetWriteMode procedure</td>
<td>149</td>
</tr>
<tr>
<td>Sin function</td>
<td>150</td>
</tr>
<tr>
<td>SizeOf function</td>
<td>151</td>
</tr>
<tr>
<td>Sound procedure</td>
<td>151</td>
</tr>
<tr>
<td>SPtr function</td>
<td>152</td>
</tr>
<tr>
<td>Sqr function</td>
<td>152</td>
</tr>
<tr>
<td>Sqrt function</td>
<td>152</td>
</tr>
<tr>
<td>SSeg function</td>
<td>152</td>
</tr>
<tr>
<td>Str procedure</td>
<td>153</td>
</tr>
<tr>
<td>Succ function</td>
<td>153</td>
</tr>
<tr>
<td>Swap function</td>
<td>154</td>
</tr>
<tr>
<td>SwapVectors procedure</td>
<td>154</td>
</tr>
<tr>
<td>TextBackground procedure</td>
<td>155</td>
</tr>
<tr>
<td>TextColor procedure</td>
<td>155</td>
</tr>
<tr>
<td>TextHeight function</td>
<td>156</td>
</tr>
<tr>
<td>TextMode procedure</td>
<td>157</td>
</tr>
<tr>
<td>TextWidth function</td>
<td>159</td>
</tr>
<tr>
<td>Trunc function</td>
<td>160</td>
</tr>
<tr>
<td>Truncate procedure</td>
<td>160</td>
</tr>
<tr>
<td>TypeOf function</td>
<td>160</td>
</tr>
<tr>
<td>UnpackTime procedure</td>
<td>161</td>
</tr>
<tr>
<td>UpCase function</td>
<td>161</td>
</tr>
<tr>
<td>Val procedure</td>
<td>161</td>
</tr>
<tr>
<td>WhereX function</td>
<td>162</td>
</tr>
<tr>
<td>WhereY function</td>
<td>163</td>
</tr>
<tr>
<td>Window procedure</td>
<td>163</td>
</tr>
<tr>
<td>Write procedure (text files)</td>
<td>164</td>
</tr>
<tr>
<td>Write procedure (typed files)</td>
<td>166</td>
</tr>
<tr>
<td>Writeln procedure</td>
<td>166</td>
</tr>
</tbody>
</table>

**Index**

169
1.1: Components of the output string .................. 165
1.2: Components of the fixed-point string ............... 165
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.

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.
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

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

Oids, 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  

\[ \text{Append}(\text{var F: Text}) \]

Remarks  

\( F \) is a text-file variable that must have been associated with an external file using \text{Assign}.  

\text{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 \texttt{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 \texttt{Ctrl-Z} in the block. In this way, text can be appended to a file that terminates with a \texttt{Ctrl-Z}.

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

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

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

\textbf{See also}  

\text{Assign, Close, Reset, Rewrite}

\textbf{Example}  

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

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 counterclockwise with 0 degrees at 3 o’clock, 90 degrees at 12 o’clock, and so on. Information about the last call to Arc can be retrieved 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  \textit{Cos, Sin}

Example \begin{verbatim}
var
    R: Real;
begin
    R := \text{ArcTan}(Pi);
end.
\end{verbatim}

Assign procedure

\begin{tabular}{|l|}
\hline
Function & Assigns the name of an external file to a file variable. \\
Declaraton & Assign(var F; Name: String) \\
Remarks & \begin{itemize}
\item $F$ is a file variable of any file type, and \textit{Name} is a string-type expression. All further operations on \textit{F} will operate on the external file with the file name \textit{Name}.
\item After a call to \textit{Assign}, the association between \textit{F} and the external file continues to exist until another \textit{Assign} is done on \textit{F}.
\item A file name consists of a path of zero or more directory names separated by backslashes, followed by the actual file name:
\begin{verbatim}
Drive:\DirName\...\DirName\FileName
\end{verbatim}
\item If the path begins with a backslash, it starts in the root directory; otherwise, it starts in the current directory.
\item \textit{Drive} is a disk drive identifier \textit{(A-Z)}. If \textit{Drive} and the colon are omitted, the default drive is used. \textit{\DirName\...\DirName} is the root directory and subdirectory path to the file name. \textit{FileName} consists of a name of up to eight characters, optionally followed by a period and an extension of up to three characters.
\item The maximum length of the entire file name is 79 characters.
\item A special case arises when \textit{Name} is an empty string; that is, when \textit{Length(Name)} is zero. In that case, \textit{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 \textit{Reset(F)}, \textit{F} will refer to the standard input file, and after a call to \textit{Rewrite(F)}, \textit{F} will refer to the standard output file.
\end{itemize}
\hline
\end{tabular}

Restrictions Assign must never be used on an open file.

See also Append, Close, Reset, Rewrite
Assign procedure

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

Function
AssignCrt (var F: Text)

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]? ');  { Output to printer }
  if UpCase(ReadKey) = 'P' then
    Assign(F, 'PRN')  
  else
    AssignCrt(F);  
  Rewrite(F);
  Writeln(F, 'Fast output via CRT routines...');
  Close(F);
end.

Bar procedure

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.
Bar3D procedure

**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
Bar3D procedure

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.
BlockRead procedure

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(l));  { Open input file }
    Reset(FromF, 1);
    Assign(ToF, ParamStr(2));   { Open output file }
    Rewrite(ToF, 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.
BlockWrite procedure

The entire block transferred occupies at most $\text{Count} \times \text{RecSize}$ bytes, where \text{RecSize} is the record size specified when the file was opened (or 128 if it was omitted). It is an error if $\text{Count} \times \text{RecSize}$ is greater than 65,535 (64K).

\text{Result} is an optional parameter. Here is how it works: If the entire block was transferred, \text{Result} will be equal to \text{Count} on return. Otherwise, if \text{Result} is less than \text{Count}, the disk became full before the transfer was completed. In that case, if the file's record size is greater than one, \text{Result} returns the number of complete records written; that is, it's possible a remaining partial record is not included in \text{Result}.

The current file position is advanced by \text{Result} records as an effect of the \text{BlockWrite}.

With [{$I-$}], IO\text{Result} returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions
- File must be open.

See also
- \text{BlockRead}

Example
- See example for \text{BlockRead}.

ChDir procedure

Function
Changes the current directory.

Declaration
\text{ChDir}(S:\ \text{String})

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

With [{$I-$}], IO\text{Result} returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also
- \text{GetDir}, \text{MdDir}, \text{RmDir}

Example
\begin{verbatim}
begin
\{$I-$}
\{ Get directory name from command line \}
\text{ChDir}(\text{ParamStr}(1));
\text{if} \ IO\text{Result} <> 0 \text{ then}
\text{Writeln('Cannot find directory');}
end.
\end{verbatim}
### Chr function

**Function**
Returns a character with a specified ordinal number.

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

### Circle procedure

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

**Declaration**
```pascal
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**
```pascal
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;
```

---

**Chapter 1, The run-time library**
Circle procedure

CloseGraph;
end.

ClearDevice procedure

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

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
ClearViewPort procedure

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;
  CloseViewPort;
  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');
  Reset(F, 1);
  Writeln('File size = ', FileSize(F));
  Close(F);
end.

Chapter 1, The run-time library
CloseGraph procedure

**Function**
Shuts down the graphics system.

**Declaration**
```
procedure CloseGraph;
```

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

**Restrictions**
Must be in graphics mode.

**See also**
`DetectGraph`, `GetGraphMode`, `InitGraph`, `RestoreCrtMode`, `SetGraphMode`

**Example**
```
uses Graph;
var
  Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ' ');
  if GraphResult <> grOk then
    Halt(1);
  Line(0, 0, GetMaxX, GetMaxY);
  Readln;
  CloseGraph;
  { Shut down graphics }
```

ClrEol procedure

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

**Declaration**
```
procedure ClrEol;
```

**Remarks**
All character positions are set to blanks with the currently defined text attributes. Thus, if `TextBackground` is not black, the 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

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 60x20 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:

  $S := 'ABC' + 'DEF'$;
Concat function

See also  Copy, Delete, Insert, Length, Pos

Example

```pascal
var
  S: String;
begin
  S := Concat('ABC', 'DEF');
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

```pascal
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

```pascal
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**

```pascal
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Dec(IntVar);
  Dec(LongintVar, 5);
end.
```
Delay procedure

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 Indexth position. If Index is larger than the length of S, no characters are deleted. If Count specifies more characters than remain starting at the Indexth position, the remainder of the string is deleted.

See also Concat, Copy, Insert, Length, Pos

DelLine procedure

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;
DetectGraph procedure

**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:

```delphi
class 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;
end
```

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);
```

Chapter 1, The run-time library
DetectGraph procedure

```pascal
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

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

**Declaration**
```pascal
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**
```pascal
uses Dos;
begin
  Writeln(DiskFree(0) div 1024, ' Kbytes free ');
end.
```

DiskSize function

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

**Declaration**
```pascal
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

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

**Dispose procedure**

<table>
<thead>
<tr>
<th>Function</th>
<th>Disposes a dynamic variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>Dispose(var P: Pointer [ , Destructor ] )</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>P</code> is a pointer variable of any pointer type that was previously assigned by the <code>New</code> procedure or was assigned a meaningful value by an assignment statement. <code>Dispose</code> destroys the variable referenced by <code>P</code> and returns its memory region to the heap. After a call to <code>Dispose</code>, the value of <code>P</code> becomes undefined, and it is an error to subsequently reference <code>P^</code>. <code>Dispose</code> has been extended to allow a destructor call as a second parameter, for disposing a dynamic object type variable. In this case, <code>P</code> is a pointer variable pointing to an object type, and <code>Destructor</code> is a call to the destructor of that object type.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>If <code>P</code> does not point to a memory region in the heap, a run-time error occurs. <code>Dispose</code> and <code>FreeMem</code> cannot be used interchangeably with <code>Mark</code> and <code>Release</code> unless certain rules are observed. For a complete discussion of this topic, see “The heap manager” in Chapter 16 in the <strong>Programmer’s Guide</strong>.</td>
</tr>
<tr>
<td>See also</td>
<td><code>FreeMem</code>, <code>GetMem</code>, <code>Mark</code>, <code>New</code>, <code>Release</code></td>
</tr>
</tbody>
</table>
| Example | `type Str18 = string[18];
       var ^Str18;
       begin
         New(P);
         P^ := 'Now you see it...';
         Dispose(P);
         { Now you don't... }` |

---

Chapter 1, The run-time library
## DosExitCode function

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

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

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

**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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns the number of strings contained in the DOS environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>EnvCount</td>
</tr>
<tr>
<td>Result type</td>
<td>Integer</td>
</tr>
<tr>
<td>Remarks</td>
<td>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.</td>
</tr>
<tr>
<td>See also</td>
<td>EnvStr, GetEnv</td>
</tr>
<tr>
<td>Example</td>
<td>uses Dos;</td>
</tr>
<tr>
<td></td>
<td>var</td>
</tr>
<tr>
<td></td>
<td>I: Integer;</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>for I := 1 to EnvCount do</td>
</tr>
<tr>
<td></td>
<td>Writeln(EnvStr(I));</td>
</tr>
<tr>
<td></td>
<td>end.</td>
</tr>
</tbody>
</table>

### EnvStr function

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns a specified environment string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>EnvStr(Index: Integer)</td>
</tr>
<tr>
<td>Result type</td>
<td>String</td>
</tr>
<tr>
<td>Remarks</td>
<td>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.</td>
</tr>
<tr>
<td>See also</td>
<td>EnvCount, GetEnv</td>
</tr>
</tbody>
</table>
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
```pascal
var
  F: Text;
  Ch: Char;
begin
  { Get file to read from command line }
  Assign(F, ParamStr(1));
  Reset(F);
  while not Eof(F) do
  begin
    Read(F, Ch);
    Write(Ch);
    { 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**  
\[ \text{Eoln} \{ \var F : \text{Text} \} \]

**Result type**  
Boolean

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

When checking \( \text{Eoln} \) 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 \( \text{Eoln} \):

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

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

**See also**  
\( \text{Eof, SeekEoln} \)

### Erase procedure

**Function**  
Erases an external file.

**Declaration**  
\[ \text{Erase} \{ \var F \} \]

**Remarks**  
\( F \) is a file variable of any file type. The external file associated with \( F \) is erased.

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

**Restrictions**  
\( \text{Erase} \) must never be used on an open file.

**See also**  
\( \text{Rename} \)

**Example**  
```pascal
var
    F: file;
    Ch: Char;
```
Erase procedure

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

Function
Execuates 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).
Exec procedure

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

```pascal
{SM $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

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

```pascal
    until False;
end;
begin
    WasteTime;
end.
```

Exp function

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns the exponential of the argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>Exp (X: Real)</td>
</tr>
<tr>
<td>Result type</td>
<td>Real</td>
</tr>
<tr>
<td>Remarks</td>
<td>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.</td>
</tr>
<tr>
<td>See also</td>
<td>Ln</td>
</tr>
</tbody>
</table>

FExpand function

<table>
<thead>
<tr>
<th>Function</th>
<th>Expands a file name into a fully qualified file name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>FExpand (Path: PathStr)</td>
</tr>
<tr>
<td>Result type</td>
<td>PathStr</td>
</tr>
</tbody>
</table>
| 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 FSsplit 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, FSsplit |

Turbo Pascal Library Reference
## 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`

### Example
```pascal
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.
```

## 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**
```pascal
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**  
```pascal
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**  
```pascal
var
  s: string[80];
begin
  { Set a string to all spaces }
  FillChar(S, SizeOf(S), ' ');
  S[0] := #80;
  { Set length byte }
end.
```

---

**FillEllipse procedure**

**Function**  
Draws a filled ellipse.

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

FillPoly procedure

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, '\"');
FillPoly procedure

```pascal
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**
```pascal
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:

```pascal
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:

```pascal
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);
  while DosError = 0 do
    begin
      Writeln(DirInfo.Name);
      FindNext(DirInfo);
    end;
  end.

FindNext procedure

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

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).
FloodFill procedure

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

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

```pascal
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 \textit{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 \textit{exactly} the number of bytes previously allocated to that variable by \textit{GetMem}. \textit{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 \textit{FreeMem}, the value of \( P \) becomes undefined, and it is an error to subsequently reference \( P \).

Restrictions  \textit{Dispose} and \textit{FreeMem} cannot be used interchangeably with \textit{Mark} and \textit{Release} unless certain rules are observed. For a complete discussion of this topic, see “The heap manager” in Chapter 16 of the \textit{Programmer’s Guide}.

See also  \textit{Dispose, GetMem, Mark, New, Release}

FSearch function

Function  Searches for a file in a list of directories.

Declaration  FSearch(Path: PathStr; DirList: String)

Result type  \textit{PathStr}

Remarks  Searches for the file given by \textit{Path} in the list of directories given by \textit{DirList}. The directories in \textit{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 \textit{PathStr} type is defined in the \textit{Dos} unit as \textit{string}[79].

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

The result of \textit{FSearch} can be passed to \textit{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

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:

```pascal
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;

Chapter 1, The run-time library
FSplit procedure

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

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 predefined as follows:

```pascal
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
```pascal
uses Graph;
var
  Gd, Gm: Integer;
  ArcCoords: ArcCoordsType;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
```
### GetArcCoords procedure

```pascal
Arc(100, 100, 0, 270, 30);
GetArcCoords(ArcCoords);
with ArcCoords do
    Line(Xstart, Ystart, Xend, Yend);
Readln;
CloseGraph;
end.
```

### GetAspectRatio procedure

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

**Declaration**
```pascal
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**
```pascal
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

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
        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

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

**Declaration**
```pascal
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

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

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

---

**Chapter 1. The run-time library** 45
GetColor function

    LineTo(Random(GetMaxX), Random(GetMaxY));
    until KeyPressed;
    CloseGraph;
end.

GetDate procedure

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

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);  
MyPal := OldPal;  
{ 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

**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**
```pascal
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

**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**
```pascal
{$M 8192,0,

uses Dos;
```
GetEnv function

```pascal
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

**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 **anding** them with the file attribute masks defined as constants in the Dos unit:

```pascal
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;
```
GetFAttr procedure

begin
{ Get file name from command line }
Assign(F, ParamStr(l));
GetFAttr(F, Attr);
Writeln(ParamStr(l));
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

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  

**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:

```pascal
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**

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

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

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:

<table>
<thead>
<tr>
<th>Graphics driver</th>
<th>Constant name</th>
<th>Value</th>
<th>Column x row</th>
<th>Palette</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGA</td>
<td>CGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAHi</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>MCGA</td>
<td>MCGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAMed</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAHi</td>
<td>5</td>
<td>640x480</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA</td>
<td>EGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>EGAHi</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td>EGA64</td>
<td>EGA64Lo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EGA64Hi</td>
<td>1</td>
<td>640x350</td>
<td>4 color</td>
<td>1</td>
</tr>
</tbody>
</table>
## GetGraphMode function

<table>
<thead>
<tr>
<th>Graphics driver</th>
<th>Constant name</th>
<th>Value</th>
<th>Column x row</th>
<th>Palette</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGA-MONO</td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>1*</td>
</tr>
<tr>
<td></td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>2**</td>
</tr>
<tr>
<td>HERC</td>
<td>HercMonoHi</td>
<td>0</td>
<td>720x348</td>
<td>2 color</td>
<td>2</td>
</tr>
<tr>
<td>ATT400</td>
<td>ATT400C0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400Med</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400Hi</td>
<td>5</td>
<td>640x400</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>VGA</td>
<td>VGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAMed</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAHi</td>
<td>2</td>
<td>640x480</td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td>PC3270</td>
<td>PC3270Hi</td>
<td>0</td>
<td>720x350</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>IBM8514</td>
<td>IBM8514Lo</td>
<td>0</td>
<td>640x480</td>
<td>256 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IBM8514Hi</td>
<td>0</td>
<td>1024x768</td>
<td>256 color</td>
<td>1</td>
</tr>
</tbody>
</table>

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

### Restrictions
Must be in graphics mode.

### See also
`ClearDevice`, `DetectGraph`, `InitGraph`, `RestoreCrtMode`, `SetGraphMode`

### Example
```pascal
uses Graph;

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

---

Chapter 1, The run-time library  53
GetImage procedure

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);
  GetImage(10, 20, 30, 40, P);
  Readln;
  ClearDevice;
  PutImage(100, 100, P, NormalPut);
  Readln;
  CloseGraph;
end.
GetIntVec procedure

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

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:

```plaintext
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 }
  NormWidth = 1;
  ThickWidth = 3;
```

Restrictions
Must be in graphics mode.

See also DrawPoly, SetLineStyle

Example
uses Graph;
var
  Gd, Gm: Integer;
  OldStyle: LineSettingsType;

Chapter 1, The run-time library
GetLineSettings procedure

```
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ',');
  if GraphResult <> grOk then
    Halt(1);
  Line(0, 0, 100, 0);
  GetLineSettings(OldStyle);
  SetLineStyle(DottedLn, 0, ThickWidth);
  Line(0, 10, 100, 10);
  with OldStyle do
    SetLineStyle(LineStyle, Pattern, Thickness);
  Line(0, 20, 100, 20);
  Readln;
  CloseGraph;
end.
```

GetMaxColor function

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

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 \texttt{GetMaxMode} is the maximum value that may be passed to \texttt{SetGraphMode}. Every driver supports modes 0..\texttt{GetMaxMode}.

\textbf{Restrictions} \quad \textbf{Must be in graphics mode.}

\textbf{See also} \quad \texttt{GetModeRange}, \texttt{SetGraphMode}

\textbf{Example}
\begin{verbatim}
uses Graph;
var
  Driver, Mode: Integer;
  I: Integer;
begin
  Driver := Detect;
  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.
\end{verbatim}

\textbf{GetMaxX function}

\textbf{Function} \quad Returns the rightmost column (x resolution) of the current graphics driver and mode.

\textbf{Declaration} \quad \texttt{GetMaxX}

\textbf{Result type} \quad \texttt{Integer}

\textbf{Remarks} \quad Returns the maximum X value for the current graphics driver and mode. On a CGA in 320x200 mode; for example, \texttt{GetMaxX} returns 319.

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

\textbf{Restrictions} \quad \textbf{Must be in graphics mode.}

\textbf{See also} \quad \texttt{GetMaxY}, \texttt{GetX}, \texttt{GetY}, \texttt{MoveTo}

\textbf{Example}
\begin{verbatim}
uses Graph;
var
  Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
\end{verbatim}
GetMaxX function

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

GetMaxY function

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)
GetMem procedure

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 \( P^\wedge \).

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

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;
  InitGraph(Driver, Mode, ’’);  \{ Put in graphics 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

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 \texttt{Lowest} = 0 and \texttt{Highest} = 1:

```pascal
uses Graph;
var
  Lowest, Highest: Integer;
begin
  GetModeRange(EGA64, Lowest, Highest);
  Write('Lowest = ', Lowest);
  Write(' Highest = ', Highest);
end.
```

If the value of \texttt{GraphDriver} is invalid, the return parameters are set to -1.

See also \textit{DetectGraph}, \textit{GetGraphMode}, \textit{InitGraph}, \textit{SetGraphMode}

GetPalette procedure

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 \textit{PaletteType}. \textit{PaletteType} is defined as follows:

```pascal
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. \texttt{Colors} contains the actual colors 0..\texttt{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 \textit{GetDefaultPalette}, \textit{GetPaletteSize}, \textit{SetAllPalette}, \textit{SetPalette}
Example

```pascal
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(O, Color * 5, 100, Color * 5);
      end
  else
    Line(0, 0, 100, 0);
  Readln;
  CloseGraph;
end.
```

GetPaletteSize function

**Function**
Returns 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

Chapter 1, The run-time library
GetPixel function

**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**

```pascal
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

**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:

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

const
```

Turbo Pascal Library Reference
GetTextSettings procedure

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 Sec100 (hundredths of seconds) 0..99.

See also GetDate, setDate, SetTime, UnpackTime

Chapter 1, The run-time library
### GetVerify procedure

**Function**
Returns the state of the verify flag in DOS.

**Declaration**
```pascal
function 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

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

**Declaration**
```pascal
function GetViewSettings(var ViewPort: ViewPortType)
```

**Remarks**
GetViewSettings returns a variable of type ViewPortType.

ViewPortType is predeclared as follows:

```pascal
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**
```pascal
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns the X coordinate of the current position (CP).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>GetX</td>
</tr>
<tr>
<td>Result type</td>
<td>Integer</td>
</tr>
<tr>
<td>Remarks</td>
<td>GetX is viewport-relative. In the following example,</td>
</tr>
<tr>
<td></td>
<td>1. SetViewPort(0, 0, GetMaxX, GetMaxY, True);</td>
</tr>
<tr>
<td></td>
<td>2. MoveTo(5, 5);</td>
</tr>
<tr>
<td></td>
<td>3. SetViewPort(10, 10, 100, 100, True);</td>
</tr>
<tr>
<td></td>
<td>4. MoveTo(5, 5);</td>
</tr>
</tbody>
</table>

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.'); |
GetX function

X := GetX;
Y := GetY;
OutTextXY(20, 10, 'Now over here...');
OutTextXY(X, Y, 'Now back over here.');
Readln;
CloseGraph;
end.

GetY function

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns the Y coordinate of the current position (CP).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>GetY</td>
</tr>
<tr>
<td>Result type</td>
<td>Integer</td>
</tr>
<tr>
<td>Remarks</td>
<td>GetY is viewport-relative. In the following example,</td>
</tr>
<tr>
<td></td>
<td>1. SetViewPort(0, 0, GetMaxX, GetMaxY, True);</td>
</tr>
<tr>
<td></td>
<td>2. MoveTo(5, 5);</td>
</tr>
<tr>
<td></td>
<td>3. SetViewPort(10, 10, 100, 100, True);</td>
</tr>
<tr>
<td></td>
<td>4. MoveTo(5, 5);</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Must be in graphics mode.</td>
</tr>
<tr>
<td>See also</td>
<td>GetViewSettings,GetX, InitGraph, MoveTo, SetViewPort</td>
</tr>
</tbody>
</table>
| Example    | uses Graph;
             | var
             |     Gd, Gm: Integer;
             |     X, Y: Integer;
             | begin
             |     Gd := Detect;
             |     InitGraph(Gd, Gm, '');
             |     if GraphResult <> grOk then
             |         Halt(1);
             |     OutText('Starting here. ');
             |     X := GetX;
             |     Y := GetY;
             |     OutTextXY(20, 10, 'Now over here...'); |
OutTextXY(X, Y, 'Now back over here.');
Readln;
CloseGraph;
end.

**GotoXY procedure**

**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**

**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

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

### GraphResult function

**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:
GraphResult function

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

The following routines set GraphResult:

- Bar
- Bar3D
- ClearViewPort
- CloseGraph
- DetectGraph
- DrawPoly
- FillPoly
- FloodFill
- GetGraphMode
- ImageSize
- InitGraph
- InstallUserDriver
- InstallUserFont
- PieSlice
- RegisterBGIdriver
- RegisterBGlfont
- 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

Chapter 1, The run-time library 69
GraphResult function

```pascal
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**  

```pascal
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**  

```pascal
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**

```pascal
var W: Word;
begin
  W := Hi($1234); { $12 }
end.
```
HighVideo procedure

**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;
end. { Color is now white }

ImageSize function

**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);
  GetImage(10, 20, 30, 40, P);
  Readln;
  ClearDevice;
  PutImage(100, 100, P, NormalPut);
  Readln;
  CloseGraph;
end.

{ Allocate memory on heap }

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);
  Inc(LongintVar, 5);
end.

{ IntVar := IntVar + 1 }
{ LongintVar := LongintVar + 5 }
InitGraph procedure

InitGraph procedure

**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
InitGraph procedure

-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:

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

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
    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 Indexth 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

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).
InsLine procedure

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

| 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 }
  ...  { Override autodetection }
  InitGraph(Driver, Mode, '');  { Do graphics ... }
end.
```
The nil value for the AutoDetectPtr parameter in the InstallUserDriver call indicates there isn’t an autodetect function for the SGA.

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

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

```pascal
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-}
begi
{ 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

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, ' '); { Put in graphics 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.
Int function

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

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:

```pascal
type
  Registers = record
    case Integer of
      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
      1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
    end;
```

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

```pascal
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: ', FSize(F))
  else
    Writeln('File not found');
end.
```

Keep procedure

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

**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');
    until KeyPressed;
end.  
{ Fill the screen until a key is typed }

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**

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

**Declaration**
```pascal
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
```pascal
MoveTo(100, 100);
LineTo(200, 200);
```
is equivalent to
```pascal
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**
```pascal
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

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

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

\[ \text{MoveTo}(100, 100); \]
\[ \text{LineTo}(200, 200); \]

is equivalent to

\[ \text{Line}(100, 100, 200, 200); \]

The first method is slower and uses more code. Use \text{LineTo} only when the current pointer is at one endpoint of the line. Use \text{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.

\text{LineTo} moves the current pointer to \((X, Y)\).

**Restrictions**

Must be in graphics mode.

**See also**

\text{GetLineStyle}, \text{Line}, \text{LineRel}, \text{MoveRel}, \text{MoveTo}, \text{SetLineStyle}, \text{SetWriteMode}

**Example**

\begin{verbatim}
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.
\end{verbatim}

---

**Ln function**

**Function**

Returns the natural logarithm of the argument.

**Declaration**

\[ \text{Ln}(X: \text{Real}) \]

**Result type**

\text{Real}

**Remarks**

\(X\) is a real-type expression. The result is the natural logarithm of \(X\).

**See also**

\text{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

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;
end. { Color is now light gray }

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.
Mark procedure

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**
```pascal
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

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
MoveRel procedure

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);
  PutPixel(GetX, GetY, GetMaxColor);
  Readln;
  CloseGraph;
end.

MoveTo procedure

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;

**MsDos procedure**

**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:

```plaintext
type
  Registers = record
    case Integer of
      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
      1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
    end;
end;
```

**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 $P^\wedge$. 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 procedure

\[ \text{New}(P, \text{Init}), \ P \text{ must point to an object type, and } \text{Init} \text{ must be a call to the constructor of that object type. In both cases, the type of the function result is } P. \]

See also *Dispose, FreeMem, GetMem, Release*

---

**NormVideo procedure**

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**

Function

Turns off the internal speaker.

Declaration

NoSound

Remarks

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

\[
\text{Sound}(440); \\
\text{Delay}(500); \\
\text{NoSound};
\]

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

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 com-
patibility 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).
OutText procedure

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.

```pascal
program CPupdate;
uses Graph;
var
  Driver, Mode: Integer;
begin
  Driver := Detect;
  InitGraph(Driver, Mode, ");
  if GraphResult < 0 then
    Halt(1);
    { #1 }
  MoveTo(0, 0);
  SetTextStyle(DefaultFont, HorizDir, 1); { CharSize = 1 }
  SetTextJustify(LeftText, TopText);
  OutText('ABC'); { 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 (8x8) is not clipped at the screen edge. Instead, if any part of the string would go off the screen, no text is output. For example, the following statements would have no effect:
OutText procedure

```pascal
SetViewPort(0, 0, GetMaxX, GetMaxY, ClipOn);
SetTextJustify(LeftText, TopText);
OutTextXY(-5, 0);
OutTextXY(GetMaxX - 1, 0, 'ABC');
```

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**
```pascal
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

**Function**
Sends a string to the output device.

**Declaration**
```pascal
OutTextXY(X: 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).
OutTextXY procedure

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(O, 0, 'Perfect!');
  Readln;
  CloseGraph;
end.

OvrClearBuf procedure

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

**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**  
``` Delphi
{$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

**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.
OvrInit procedure

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

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
OvrlnitEMS procedure

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

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.
OvrSetBuf procedure

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

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

PackTime procedure

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:

```pascal
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.
ParamCount function

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 Indexth 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.

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

**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**
```pascal
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
Pos function

**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**  
```pascal
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:

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

**See also**  
Addr, Ofs, Seg
Ptr function

Example

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

PutImage procedure

Function
Puts a bit image onto the screen.

Declaration
```pascal
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:

```pascal
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, P^, NormalPut); { Will barely fit }
    PutImage(GetMaxX - 98, 0, P^, NormalPut); { X + Height > GetMaxX }
    PutImage(-1, 0, P^, NormalPut); { -1,0 not onscreen }
    PutImage(0, -1, P^, NormalPut); { 0,-1 not onscreen }
    PutImage(0, GetMaxY - 30, P^, NormalPut); { Will output 31 "lines" }
    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);
### PutImage procedure

```pascal
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

**Function**
Plots a pixel at X, Y.

**Declaration**
```pascal
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**
```pascal
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 \( \leq X < 1 \). If Range is specified, it must be an expression of type Integer, and the result is a Word random number within the range 0 \( \leq X < \) 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;
begins
  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
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
Read procedure (text files)

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

Function  Reads a character from the keyboard.

Declaration  ReadKey

Offsets 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.
ReadKey function

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:

\[
\text{Ch := ReadKey; if Ch <> #0 then FuncKey := False else begin FuncKey := True; Ch := ReadKey; end;}
\]

The CheckBreak variable controls whether Ctrl-Break should abort the program or be returned like any other key. When CheckBreak is False, ReadKey returns a Ctrl-C (#3) for Ctrl-Break.

See also KeyPressed

---

Readln procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>Executes the Read procedure then skips to the next line of the file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>Readln( [ var F: Text; ] V1 [, V2, ..., VN ] )</td>
</tr>
<tr>
<td>Remarks</td>
<td>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.</td>
</tr>
</tbody>
</table>

Readln(F) with no parameters causes the current file position to advance to the beginning of the next line (if there is one; otherwise, it goes to the end of the file). Readln with no parameter list altogether corresponds to Readln(Input).

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

<table>
<thead>
<tr>
<th>Restrictions</th>
<th>Works only on text files, including standard input. File must be open for input.</th>
</tr>
</thead>
<tbody>
<tr>
<td>See also</td>
<td>Read</td>
</tr>
</tbody>
</table>
### Rectangle procedure

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

**Declaration**
Rectangle(Xl, Yl, 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;
    Xl, Yl, 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

**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:

```pascal
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;
```
RegisterBGIdriver function

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

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:

<table>
<thead>
<tr>
<th>Error code</th>
<th>Error identifier</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-11</td>
<td>grError</td>
<td>There is no room in the font table to register another font. (The font table holds up to 10 fonts, and only 4 are provided, so this error should not occur.)</td>
</tr>
<tr>
<td>-13</td>
<td>grInvalidFont</td>
<td>The font header is not recognized.</td>
</tr>
<tr>
<td>-14</td>
<td>grInvalidFontNum</td>
<td>The font number in the font header is not recognized.</td>
</tr>
</tbody>
</table>

The following program loads the triplex font onto the heap, registers it with the graphics system, and then alternates between using triplex and another stroked font that Graph loads from disk (SansSerifFont):

```pascal
program LoadFont;
uses Graph;
var
  Driver, Mode: Integer;
  FontF: file;
  FontP: Pointer;
begin
  { Open font file, read into memory, register it }
  Assign(FontF, 'TRIP.CHR');
  Reset(FontF, 1);
  GetMem(FontP, FileSize(FontF));
  BlockRead(FontF, FontP, FileSize(FontF));
  if RegisterBGIfont(FontP) < 0 then
    begin
      Writeln('Error registering font: ', GraphErrorMsg(GraphResult));
      Halt(1);
    end;
  { Init graphics }
  Driver := Detect;
  InitGraph(Driver, Mode, '..\');
  if GraphResult < 0 then
    Halt(1);
  Readln;
  { Select registered font }
  SetTextStyle(TriplexFont, HorizDir, 4);
  OutText('Triplex loaded by user program');
  MoveTo(0, TextHeight('a'));
  Readln;
  { Select font that must be loaded from disk }
  SetTextStyle(SansSerifFont, HorizDir, 4);
  OutText('Your disk should be spinning...');
```
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 (8x8 bit-mapped) font is built into GRAPH.TPU, and thus is always in memory. Once a stroked font has been loaded, your program can alternate between the default font and the stroked font without having to reload either one of them.
RegisterBGIfont function

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-$I-$}, *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')
Reset procedure

```pascal
else
   Writeln('File not found');
end.
```

**RestoreCrtMode procedure**

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

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

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

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 Debug 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

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

Chapter 1, The run-time library 121
Sector procedure

Example

uses Graph;
const
  R = 50;
var
  Driver, Mode: Integer;
  Xasp, Yasp: Word;
begin
  Driver := Detect;
  InitGraph(Driver, Mode, '');
  if GraphResult < 0 then
    Halt(1);
  Sector(GetMaxX div 2, GetMaxY div 2, 0, 45, R, R);
  GetAspectRatio(Xasp, Yasp);
  Sector(GetMaxX div 2, GetMaxY div 2, 180, 135, R, R * Longint(Xasp) div Yasp);
  Readln;
  CloseGraph;
end.

Seek procedure

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**

**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

<table>
<thead>
<tr>
<th>Function</th>
<th>Changes all palette colors as specified.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>SetAllPalette(var Palette)</td>
</tr>
<tr>
<td>Remarks</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```plaintext
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;
```

Chapter 1, The run-time library
SetAllPalette procedure

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

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**
```pascal
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**

- **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 procedure

`SetBkColor(N)` makes the Nth color in the palette the new background color. The only exception is `SetBkColor(0)`, which always sets the background color to black.

**Restrictions**
Must be in graphics mode.

**See also**
`GetBkColor, GetColor, GetPalette, SetAllPalette, SetColor, SetPalette, SetRGBPalette`

**Example**
```pascal
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

**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

**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

**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
SetFAAttr procedure

**Function**
Sets the attributes of a file.

**Declaration**
SetFAAttr(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.

```pascal
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**
GetFAAttr, GetFTime, SetFTime

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

SetFillPattern procedure

**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 corre-
sponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is 
valued at 1, a pixel will be plotted. For example, the following pattern 
represents a checkerboard (50% gray scale):

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101010</td>
<td>$AA</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
</tr>
<tr>
<td>10101010</td>
<td>$AA</td>
</tr>
<tr>
<td>01010101</td>
<td>$55</td>
</tr>
</tbody>
</table>

User-defined fill patterns enable you to create patterns different from the 
predefined fill patterns that can be selected with the *SetFillStyle* 
procedure. Whenever you select a new fill pattern with *SetFillPattern* or 
*SetFillStyle*, all fill operations will use that fill pattern. Calling *SetFillStyle* 
(*UserField, SomeColor*) will always select the user-defined pattern. This lets 
you define and use a new pattern using *SetFillPattern*, then switch 
between your pattern and the built-ins by making calls to *SetTextStyle*.

**Restrictions**

Must be in graphics mode.

**See also**

*Bar, Bar3D, FillPoly, GetFillPattern, GetFillSettings, GraphResult, PieSlice*

**Example**

```pascal
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.
```

Chapter 1, The run-time library 131
SetFillStyle procedure

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, '的专业方式');
    SetFillStyle(SolidFill, 0);
    Bar(0, 0, 50, 50);
    SetFillStyle(XHatchFill, 1);
    Bar(60, 0, 110, 50);
    Readln;
### SetFTime procedure

**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

**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*
SetGraphMode procedure

SetGraphMode procedure

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:

<table>
<thead>
<tr>
<th>Graphics driver</th>
<th>Graphics modes</th>
<th>Value</th>
<th>Column x row</th>
<th>Palette</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGA</td>
<td>CGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CGAHi</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>MCGA</td>
<td>MCGAC0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAC3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAMed</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MCGAHi</td>
<td>5</td>
<td>640x480</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA</td>
<td>EGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>EGAHi</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td>EGA64</td>
<td>EGA64Lo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EGA64Hi</td>
<td>1</td>
<td>640x350</td>
<td>4 color</td>
<td>1</td>
</tr>
<tr>
<td>EGA-MONO</td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>1*</td>
</tr>
<tr>
<td></td>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350</td>
<td>2 color</td>
<td>2**</td>
</tr>
<tr>
<td>HERC</td>
<td>HercMonoHi</td>
<td>0</td>
<td>720x348</td>
<td>2 color</td>
<td>2</td>
</tr>
<tr>
<td>ATT400</td>
<td>ATT400C0</td>
<td>0</td>
<td>320x200</td>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C1</td>
<td>1</td>
<td>320x200</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ATT400C2</td>
<td>2</td>
<td>320x200</td>
<td>C2</td>
<td>1</td>
</tr>
</tbody>
</table>
## SetGraphMode procedure

<table>
<thead>
<tr>
<th>Graphics driver</th>
<th>Graphics modes</th>
<th>Value</th>
<th>Column x row</th>
<th>Palette</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT400C3</td>
<td>3</td>
<td>320x200</td>
<td>C3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ATT400Med</td>
<td>4</td>
<td>640x200</td>
<td>2 color</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ATT400Hi</td>
<td>5</td>
<td>640x400</td>
<td>2 color</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>VGA</td>
<td>VGALo</td>
<td>0</td>
<td>640x200</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAMed</td>
<td>1</td>
<td>640x350</td>
<td>16 color</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VGAHi</td>
<td>2</td>
<td>640x480</td>
<td>16 color</td>
<td>1</td>
</tr>
<tr>
<td>PC3270</td>
<td>PC3270Hi</td>
<td>0</td>
<td>720x350</td>
<td>2 color</td>
<td>1</td>
</tr>
<tr>
<td>514</td>
<td>IBM8514Lo</td>
<td>0</td>
<td>640x480</td>
<td>256 color</td>
<td>1</td>
</tr>
<tr>
<td>8514</td>
<td>IBM8514Hi</td>
<td>0</td>
<td>1024x768</td>
<td>256 color</td>
<td>1</td>
</tr>
</tbody>
</table>

* 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
```pascal
uses Graph;

var
    GraphDriver: Integer;
    GraphMode: Integer;
    LowMode: Integer;
    HighMode: Integer;

begin
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, '');
    if GraphResult <> grOk then
        Halt(1);
    GetModeRange(GraphDriver, LowMode, HighMode);
    SetGraphMode(LowMode); (* Select low-resolution mode *)
    Line(0, 0, GetMaxX, GetMaxY);
    Readln;
    CloseGraph;
end.
```
SetIntVec procedure

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 $1B handler and later restores the original handler:

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

See also
GetIntVec

SetLineStyle procedure

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;
  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:
SetLineStyle procedure

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

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:

```pascal
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**
```pascal
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;
```
SetPalette procedure

else
  Line(0, 0, 100, 0);
Readln;
CloseGraph;
end.

SetRGBPalette procedure

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:$80;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}]
SetRGBPalette procedure

(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:$F;BlueVal:$24), {Yellow EGA 62}
(RedVal:$FC;GreenVal:$F;BlueVal:$FC) {Br. White EGA 63}

var
  Driver, Mode, I: Integer;
begin
  Driver := IBM8514;
  Mode := IBM8514Hi;
  InitGraph(Driver, 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
SetTextBuf procedure

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. CallingSetTextBuf 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

```pascal
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.
end.
```
**SetTextJustify procedure**

**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;
  TopText    = 2;  { Not declared twice }
```

**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, 't');
  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

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 8x8 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 8x8 bit-mapped font in an 8x8 pixel rectangle on the screen, a CharSize value equal to 2 will display the 8x8 bit-mapped font in a 16x16 pixel rectangle and so on (up to a limit of 10 times the normal size). Always use TextHeight and TextWidth to determine the actual dimensions of the text.

The normal size values for text are 1 for the default font and 4 for a stroked font. These are the values that should be passed as the CharSize parameter to SetTextStyle. SetUserCharSize can be used to customize the dimensions of stroked font text.

Normally, stroked fonts are loaded from disk onto the heap when a call is made to SetTextStyle. However, you can load the fonts yourself or link them directly to your .EXE file. In either case, use RegisterBGLfont 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;
```

Chapter 1, The run-time library 143
SetTextStyle procedure

HorizDir = 0;
VertDir = 1;

Restrictions  Must be in graphics mode.

See also  GetTextStyle, 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 Sec100 (hundredths of seconds) 0..99. If the time is not valid, the request is ignored.

See also  GetDate, GetTime, PackTime, SetDate, UnpackTime
SetUserCharSize procedure

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

**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

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

**Declaration**
SetViewPort(Xl, Yl, 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)).
SetViewPort procedure

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:

```plaintext
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:

```plaintext
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
SetViewport procedure

Example

```pascal
uses Graph;
var
  Gd, Gm: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then
    Halt(1);
  if (Gd = HercMono) or (Gd = EGA) or (Gd = EGA64) or (Gd = VGA) then
    begin
      SetVisualPage(0);
      SetActivePage(1);
      Rectangle(10, 20, 30, 40);
      SetVisualPage(1);
    end
  else
    OutText('No paging supported.);
  Readln;
  CloseGraph;
end.
```

SetVisualPage procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>Sets the visual graphics page number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>SetVisualPage(Page: Word)</td>
</tr>
<tr>
<td>Remarks</td>
<td>Makes Page the visual graphics page.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Must be in graphics mode.</td>
</tr>
<tr>
<td>See also</td>
<td>SetActivePage</td>
</tr>
<tr>
<td>Example</td>
<td>uses Graph;</td>
</tr>
<tr>
<td></td>
<td>var</td>
</tr>
<tr>
<td></td>
<td>Gd, Gm: Integer;</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>Gd := Detect;</td>
</tr>
<tr>
<td></td>
<td>InitGraph(Gd, Gm, '');</td>
</tr>
<tr>
<td></td>
<td>if GraphResult &lt;&gt; grOk then</td>
</tr>
<tr>
<td></td>
<td>Halt(1);</td>
</tr>
<tr>
<td></td>
<td>if (Gd = HercMono) or (Gd = EGA) or</td>
</tr>
<tr>
<td></td>
<td>(Gd = EGA64) or (Gd = VGA) then</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>SetVisualPage(0);</td>
</tr>
<tr>
<td></td>
<td>SetActivePage(1);</td>
</tr>
<tr>
<td></td>
<td>Rectangle(10, 20, 30, 40);</td>
</tr>
<tr>
<td></td>
<td>SetVisualPage(1);</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>OutText('No paging supported.');</td>
</tr>
<tr>
<td></td>
<td>Readln;</td>
</tr>
<tr>
<td></td>
<td>CloseGraph;</td>
</tr>
<tr>
<td></td>
<td>end.</td>
</tr>
</tbody>
</table>
SetVisualPage procedure

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

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;
  Xl, Yl, Dx, Dy: Integer;
  FillInfo: FillSettingsType;
begin
  DirectVideo := False;
  { Turn off screen write }
  Randomize;
  { Put in graphics mode }
  Driver := Detect;
  InitGraph(Driver, Mode, '');
SetWriteMode procedure

if GraphResult < 0 then
  Halt(1);
  { Fill screen with background pattern }
GetFillSettings(FillInfo);
SetFillStyle(WideDotFill, FillInfo.Color);
Bar(0, 0, GetMaxX, GetMaxY);
Dx := GetMaxX div 4;
Dy := GetMaxY div 4;
SetLineStyle(SolidLn, 0, ThickWidth);
SetWriteMode(XORPut);
repeat
  X1 := Random(GetMaxX - Ox);
  Y1 := Random(GetMaxY - Dy);
  Rectangle(X1, Y1, X1 + Ox, Y1 + Dy);
  Delay(10);
  Rectangle(X1, Y1, X1 + Ox, Y1 + Dy);
  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

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);
  Delay(200);
  NoSound;
end.
```
**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**  
\[ \text{Str}(X [: \text{Width} [: \text{Decimals} ] ]; \text{var } S : \text{String}) \]

**Remarks**  
\( X \) is an integer-type or real-type expression. \( \text{Width} \) and \( \text{Decimals} \) are integer-type expressions. \( S \) is a string-type variable. \( \text{Str} \) converts \( X \) to its string representation, according to the \( \text{Width} \) and \( \text{Decimals} \) formatting parameters. The effect is exactly the same as a call to the \( \text{Write} \) 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**  \( \text{Val, Write} \)

**Example**  
\[
\begin{align*}
\text{function } \text{IntToStr}(I : \text{Longint}) : \text{String};
\{ \text{Convert any integer type to a string} \} \\
\text{var} \\
\quad S : \text{string}[11]; \\
\begin{align*}
\text{begin} \\
\quad \text{Str}(I, S); \\
\quad \text{IntToStr} := S;
\end{align*} \\
\text{end}; \\
\begin{align*}
\text{begin} \\
\quad \text{WriteLn} \left( \text{IntToStr}(-5322) \right); \\
\text{end}.
\end{align*}
\]

Succ function

**Function**  Returns the successor of the argument.

**Declaration**  \( \text{Succ}(X) \)

**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**  \( \text{Inc, Pred} \)
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  

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

SwapVectors procedure

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  

```pascal
{$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

**Function**
Selects the background color.

**Declaration**

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

**Remarks**

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

```pascal
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

**Function**
Selects the foreground character color.

**Declaration**

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

**Remarks**

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

```pascal
const
  Black    = 0;
  Blue     = 1;
  Green    = 2;
  Cyan     = 3;
  Red      = 4;
  Magenta  = 5;
  Brown    = 6;
  LightGray = 7;
  DarkGray = 8;
  LightBlue = 9;
```
TextColor procedure

LightGreen = 10;
LightCyan = 11;
LightRed = 12;
LightMagenta = 13;
Yellow = 14;
White = 15;

There is a byte variable in Crt—TextColor—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);
TextColor(LightRed + Blink);
TextColor(14);

TextHeight function

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 8x8 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

Turbo Pascal Library Reference
Example

```pascal
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Selects a specific text mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>TextMode (Mode: Word)</td>
</tr>
<tr>
<td>Remarks</td>
<td>The following constants are defined:</td>
</tr>
</tbody>
</table>

```pascal
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.
TextMode procedure

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

**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**

```pascal
uses Graph;
var
  Gd, Gm: Integer;
  Row: Integer;
  Title: String;
  Size: Integer;
begin
  Gd := Detect;
  InitGraph(Gd, Gm, ");
  if GraphResult <> grOk then
    Halt(1);
  Row := 0;
  Title := 'Turbo Graphics';
  Size := 1;
  while TextWidth(Title) < GetMaxX do
    begin
      OutTextXY(O, Row, Title);
      Inc(Row, TextHeight('M'));
      Inc(Size);
      SetTextStyle(DefaultFont, HorizDir, Size);
    end;
  Readln;
end;
```

Chapter 1, The run-time library
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 virtual methods, a compiler error occurs.
UnpackTime procedure

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.
Val procedure

With range-checking on, \{\texttt{R+}\}, an out-of-range value always generates a run-time error. With range-checking off, \{\texttt{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 \(\text{Code}\) returns a nonzero value. For any other numeric type, \(\text{Code}\) 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 \texttt{Val} a Longint variable and perform range-checking before making an assignment of the returned value:

\begin{verbatim}
{R-}
Val(\texttt{'65536'}, \texttt{LongIntVar, Code})
if (Code <> 0) or (LongIntVar < 0) or (LongIntVar > 65535) then
  ... { Error }
else
  WordVar := LongIntVar;
\end{verbatim}

In this example, \texttt{LongIntVar} would be set to 65,536, and \texttt{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**

\texttt{Str}

**Example**

\begin{verbatim}
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.
\end{verbatim}

WhereX function

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns the X-coordinate of the current cursor position, relative to the current window.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>\texttt{WhereX}</td>
</tr>
<tr>
<td>Result type</td>
<td>\texttt{Byte}</td>
</tr>
<tr>
<td>See also</td>
<td>\texttt{GotoXY, WhereY, Window}</td>
</tr>
</tbody>
</table>
WhereY function

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

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);
end { Clear screen }
Window procedure

ClrScr;
repeat
  X := Succ(Random(80));
  Y := Succ(Random(25));
  Window(X, Y, X + Random(10), Y + Random(8));
  TextBackground(Random(16));
  ClrScr;
until KeyPressed;
end.

Write procedure (text files)

<table>
<thead>
<tr>
<th>Function</th>
<th>Writes one or more values to a text file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>Write( [ var F: Text; ] V1 [, V2,...,VN ] )</td>
</tr>
<tr>
<td>Remarks</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>OutExpr [: MinWidth [: DecPlaces ] ]</td>
</tr>
</tbody>
</table>

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 procedure (text files)

**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>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>- ]</td>
</tr>
<tr>
<td>&lt;digit&gt;</td>
<td>Single digit, &quot;0&quot; only; if OutExpr is 0</td>
</tr>
<tr>
<td>&lt;decimals&gt;</td>
<td>Digit string of MinWidth-7 (but at most 10) digits</td>
</tr>
<tr>
<td>E</td>
<td>Uppercase [E] character</td>
</tr>
<tr>
<td>[ +</td>
<td>- ]</td>
</tr>
<tr>
<td>&lt;exponent&gt;</td>
<td>Two-digit decimal exponent</td>
</tr>
</tbody>
</table>

If DecPlaces is present, a fixed-point decimal string is written. If DecPlaces is larger than 11, it is assumed to be 11. The format of the fixed-point string follows:

```
[ <blanks> ] [ - ]<digits>[ . <decimals> ]
```

The components of the fixed-point string are shown in Table 1.2:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ &lt;blanks&gt; ]</td>
<td>Blanks to satisfy MinWidth</td>
</tr>
<tr>
<td>[ - ]</td>
<td>If OutExpr is negative</td>
</tr>
<tr>
<td>&lt;digits&gt;</td>
<td>At least one digit, but no leading zeros</td>
</tr>
<tr>
<td>[ . &lt;decimals&gt; ]</td>
<td>Decimals if DecPlaces &gt; 0</td>
</tr>
</tbody>
</table>

**Write with a string-type value:** If MinWidth is omitted, the string value of OutExpr is written to the file with no leading blanks. If MinWidth is specified, and its value is larger than the length of OutExpr, enough blanks are written before the decimal string to make the field width MinWidth.
Write procedure (text files)

**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)

<table>
<thead>
<tr>
<th>Function</th>
<th>Writes a variable into a file component.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>Write(F, V1 [, V2, ..., VN ] )</code></td>
</tr>
<tr>
<td>Remarks</td>
<td><code>F</code> is a file variable, and each <code>V</code> is a variable of the same type as the component type of <code>F</code>. 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 <code>Eof(F)</code> is True—the file is expanded. With <code>{I-}</code>, <code>IOResult</code> returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.</td>
</tr>
<tr>
<td>See also</td>
<td><code>Writeln</code></td>
</tr>
</tbody>
</table>

Writeln procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>Executes the <code>Write</code> procedure, then outputs an end-of-line marker to the file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td><code>Writeln( [ var F: Text; ] V1 [, V2, ..., VN ] )</code></td>
</tr>
</tbody>
</table>
| 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).`
Writeln procedure

Restrictions  File must be open for output.
See also  Write
$ See compiler directives
\@ (address-of) operator See address-of (@) operator

A
Abs function 4
Addr function 4
address-of (@) operator versus Addr 4
AndPut constant 104
Append procedure 5
Arc procedure 6
ArcTan function 6
aspect ratio 43
correction factor, changing 126
Assign procedure 7
AssignCrt procedure 8

B
Bar3D procedure 9
Bar procedure 8
BGI, Zenith Z-449 and 74
BINOBJ 112, 113
bit images 54
BitBlt operations 104
BlockRead procedure 10
BlockWrite procedure 11
buffers, flushing 39

C
ChDir procedure 12
Chr function 13
Circle procedure 13
ClearDevice procedure 14
clearing the overlay buffer 96
ClearViewPort procedure 14
clipping parameters 64

Close procedure 15
CloseGraph procedure 16
ClrEol procedure 16
ClrScr procedure 17
colors 50, 51
background 44
drawing 45
command-line parameters 101
compiler directives
$1 81
$M 81, 87, 88, 100
Concat function 17
CONFIG.SYS 98
Copy function 18
Cos function 18
Crt unit
AssignCrt procedure 8
ClrEol procedure 16
ClrScr procedure 17
Delay procedure 20
DelLine procedure 20
GotoXY procedure 67
HighVideo procedure 71
InsLine procedure 75
KeyPressed function 82
LowVideo procedure 86
NormVideo procedure 92
NoSound procedure 92
ReadKey function 109
Sound procedure 151
TextColor procedure 155
TextBackground procedure 155
TextColor procedure 155
TextMode procedure 157
WhereX function 162
WhereY function 163
Window procedure 163
CS register 19
CSeg function 19
D

date and time procedures
  GetDate 46
  GetFTime 52
  GetTime 63
  SetDate 129
  SetFTime 133
  SetTime 144
Dec procedure 19
Delay procedure 20
Delete procedure 20
DelLine procedure 20
DetectGraph procedure 21

devices 14
  drivers, installing 76
directives See compiler, directives
directories 47
  changing 12
  procedures 120
  searching 36, 40
DiskFree function 22
disks, space 22
DiskSize function 22
Dispose procedure 23

DOS
  Pascal functions for 91
  verify flag 64
    setting 146
Dos unit
  DiskFree function 22
  DiskSize function 22
  DosExitCode function 24
  DosVersion function 24
  EnvCount function 27
  EnvStr function 27
  Exec procedure 30
  FExpand function 32
  FindFirst procedure 36
  FindNext procedure 37
  FSearch function 40
  FSplit procedure 41
  GetCBreak procedure 45
  GetDate procedure 46

GetEnv function 48
GetFAttr procedure 49
GetFTime procedure 52
GetIntVec procedure 55
GetTime procedure 63
GetVerify procedure 64
Intr procedure 80
Keep procedure 81
MsDos procedure 91
PackTime procedure 100
SetCBreak procedure 128
SetFTime procedure 133
SetIntVec procedure 136
SetTime procedure 144
SetVerify 146
SwapVectors procedure 154
UnpackTime procedure 161

DosError variable 30, 36, 37, 49, 52, 130, 133
DosExitCode function 24
DosVersion function 24

DrawPoly procedure 25
drivers
  active, returning
    maximum mode number 56
    name 48
DS register 26
DSeg function 26

E

ellipse, drawing 34
Ellipse procedure 26
elliptical sector, drawing and filling 121
doing of file status 28
EnvCount function 27
EnvStr function 27
Eof function 28
Eonl function 29
Erase procedure 29
errors
  codes for graphics operations 68
  messages 68
  run-time See run-time, errors
Exec procedure 30
exit
  codes 24
  procedures 31
Exp function 32
F

FExpand function 32
file-handling procedures
  Rename 116
  Reset 117
  Rewrite 119
  Seek 122
  SetFAttr 130
  Truncate 160
FilePos function 33
files
  Assign procedure 7
  attributes 49
  closing 15
  erasing 29
    untyped, variable 10, 11
FileSize function 33
fill patterns 50
FillChar procedure 34
FillEllipse procedure 34
filling areas 37
FillPoly procedure 35
FindFirst procedure 36
FindNext procedure 37
FloodFill procedure 37
Flush procedure 39
Font8x8 variable 157, 158
fonts
  installing 79
  stroked 145
Frac function 39
fractions, returning 39
FreeMem procedure 40
FSearch function 40
FSplit procedure 32, 41
functions See specific function

G

GetArcCoords procedure 42
GetAspectRatio procedure 43
GetBkColor function 44
GetCBreak procedure 45
GetColor function 45
GetDate procedure 46
GetDefaultPalette function 46
GetDir procedure 47

GetDriverName function 48
GetEnv function 48
GetFAttr procedure 49
GetFillPattern procedure 50
GetFillSettings procedure 51
GetFTime procedure 52
GetGraphMode function 52
GetImage procedure 54
GetIntVec procedure 55
GetLineSettings procedure 55
GetMaxColor function 56
GetMaxMode-function 56
GetMaxX function 57
GetMaxY function 58
GetMem procedure 58
GetModeName function 59
GetModeRange procedure 60
GetPalette procedure 60
  IBM 8514 and 60
GetPaletteSize function 61
GetPixel function 62
GetPixel function 62
GetTextSettings procedure 62
GetTime procedure 63
GetVerify procedure 64
GetViewSettings procedure 64
GetX function 65
GetY function 66
GotoXY procedure 67
Graph unit
  Arc procedure 6
  Bar3D procedure 9
  Bar procedure 8
  Circle procedure 13
  ClearDevice procedure 14
  ClearViewport procedure 14
  CloseGraph procedure 16
  DetectGraph procedure 21
  DrawPoly procedure 25
  Ellipse procedure 26
  FillEllipse procedure 34
  FillPoly procedure 35
  FloodFill procedure 37
  GetArcCoords procedure 42
  GetAspectRatio procedure 43
  GetBkColor function 44
  GetColor function 45
  GetDefaultPalette function 46
  GetDir procedure 47

Index
GetDriverName function 48
GetFillPattern procedure 50
GetFillSettings procedure 51
GetGraphMode function 52
GetImage procedure 54
GetLineSettings procedure 55
GetMaxColor function 56
GetMaxMode function 56
GetMaxX function 57
GetMaxY function 58
GetModeName function 59
GetModeRange procedure 60
GetPalette procedure 60
GetPaletteSize function 61
GetPixel function 62
GetTextSettings procedure 62
GetViewSettings procedure 64
GetX function 65
GetY function 66
GraphDefaults procedure 67
GraphErrorMsg function 68
ImageSize function 71
InitGraph procedure 73
InstallUserDriver function 76
InstallUserFont function 79
Line procedure 83
LineRel procedure 84
LineTo procedure 84
MoveRel procedure 89
MoveTo procedure 90
OutText procedure 93
OutTextXY procedure 95
PieSlice procedure 102
PutImage procedure 104
PutPixel procedure 106
Rectangle procedure 111
RegisterBGIdriver function 112
RegisterBGIfont function 113
Sector procedure 121
SetActivePage procedure 124
SetAllPalette procedure 125
SetAspectRatio procedure 126
SetBkColor procedure 127
SetColor procedure 129
SetFillPattern procedure 130
SetFillStyle procedure 132
SetGraphBufSize procedure 133
SetGraphMode procedure 134
SetLineStyle procedure 136
SetPalette procedure 137
SetRGBPalette procedure 139
SetTextJustify procedure 142
SetTextStyle procedure 143
SetUserCharSize procedure 145
SetViewPort procedure 146
SetVisualPage procedure 148
SetWriteMode 149
TextHeight function 156
TextWidth function 159
GraphDefaults procedure 67
GraphDriver variable
IBM 8514 and 74
GraphErrorMsg function 68
bit-image operations 104
cards 21, 73
drawing operations 83, 84, 102, 111, 136
drivers 73
fill operations 130, 132
mode 52, 73, 83, 84, 85
page operations 124, 148
palette operations 125, 127, 129, 137
plotting operations 106
pointer operations 90
polygons, drawing 25
resolution 43
system operations 134
text operations 93, 95, 142, 156
video mode operations 118
viewport operations 146
GraphMode procedure 73
GraphErrorMsg function 68
error codes 68
grError 69
grInvalidFont 69
grInvalidFontNum 69
grIOerror 69
H
Halt procedure 70
handles, DOS 7
heap management procedures 116
Hi function 70
  high
    intensity characters 71
  order bytes 70
HighVideo procedure 71

I
$I$ compiler directive 81
I/O
  checking 81
  DOS standard 7
IBM 8514
  GetPalette procedure and 60
  GraphDriver variable and 74
  InitGraph procedure and 74
  palette entries, modifying 139
  SetAllPalette procedure and 126
  SetPalette procedure and 138
ImageSize function 71
Inc procedure 72
InitGraph procedure 73
  SetGraphMode and 134
input, DOS standard 7
Insert procedure 75
inserting
  lines 75
  strings 75
InsLine procedure 75
InstallUserDriver function 76
InstallUserFont function 79
Int function 79
interrupt
  procedures 136
  vectors 55
    swapping 154
Intr procedure 80
IOResult function 80

J
justification, font 62

K
Keep procedure 81
keyboard operations 82, 109
KeyPressed function 82

L
Length function 82
line
  drawing, setting writing mode for 149
  settings 55
Line procedure 83
LineRel procedure 84
LineTo procedure 84
Ln function 85
Lo function 86
LowVideo procedure 86

M
$M$ compiler directive 81, 87, 88, 100
Mark procedure 86
MaxAvail function 87
MemAvail function 88
memory 40, 58
MkDir procedure 88
Move procedure 89
MoveRel procedure 89
MoveTo procedure 90
MsDos procedure 91

N
New procedure 91
  extended syntax
    constructor passed as parameter 91
NormalPut constant 104
NormVideo procedure 92
NoSound procedure 92
NotPut constant 104

O
Odd function 92
Ofs function 93
Ord function 93
OrPut constant 104
output, DOS standard 7
OutText procedure 93
OutTextXY procedure 95
Overlay unit
  OvrClearBuf procedure 96
  OvrGetBuf function 97
  OvrInit procedure 97
OvrInitEMS procedure 98
OvrSetBuf procedure 99

overlays
buffers
  clearing 96
  size
    returning 97
    setting 99
files
  loading into EMS 98
  opening 97
  manager, initializing 97
OvrClearBuf procedure 96
OvrGetBuf function 97
OvrInit procedure 97
OvrInitEMS procedure 98
OvrSetBuf procedure 99

P
PackTime procedure 100
palette
  color lookup table, returning size 61
    definition record 46
ParamCount function 100
parameters, command-line 101
ParamStr function 101
Pi function 101
PieSlice procedure 102
pixel values 62
polygons, drawing 25
Pos function 102
Pred function 103
procedures See specific procedure
process-handles routines 81
programs
  execution, stopping 121
    halting 70
Ptr function 103
PutImage procedure 104
PutPixel procedure 106

R
Random function 107
Randomize procedure 107
range-checking, Val and 161
Read procedure
text files 108
typed files 109
reading records 10
ReadKey function 109
Readln procedure 110
Rectangle procedure 111
RegisterBGIdevice function 112
RegisterBGIfont function 113
registers, CS 19
Release procedure 116
Rename procedure 116
Reset procedure 117
resolution, graphics 43
RestoreCrtMode procedure 118
Rewrite procedure 119
Rmdir procedure 120
Round function 120
run-time errors, generating 121
RunError procedure 121

S
searching directories 36
Sector procedure 121
Seek procedure 122
SeekEof function 123
SeekEoln function 123
Seg function 124
SetActivePage procedure 124
SetAllPalette procedure 125
  IBM 8514 and 126
SetAspectRatio procedure 126
SetBkColor procedure 127
SetCBreak procedure 128
SetColor procedure 129
SetDate procedure 129
SetFAttr procedure
  IBM 8514 and 130
SetFillPattern procedure 130
SetFillStyle procedure 132
SetFTime procedure 133
SetGraphBufSize procedure 133
SetGraphMode procedure 134
SetIntVec procedure 136
SetLineStyle procedure 136
SetPalette procedure 136
  IBM 8514 and 137
SetRGBPalette procedure 139
SetTextBuf procedure 140
SetTextJustify procedure 142
SetTextStyle procedure 143
OutText and 93
OutTextXY and 95
SetTime procedure 144
SetUserCharSize procedure 145
SetVerify procedure 146
SetViewPort procedure 146
SetVisualPage procedure 148
SetWriteMode procedure 149
Sin function 150
SizeOf function 34, 151
software interrupts 80
sound operations
NoSound 92
Sound 151
Sound procedure 151
SPtr function 152
Sqr function 152
Sqrt function 152
SSeg function 152
Str procedure 153
strings
concatenation 17
construction 17
deletion 20
initializing 34
length byte 34
procedures 75, 153, 161
stroked fonts 145
substrings
copying 18
deleting 20
inserting 75
position 102
Succ function 153
Swap function 154
SwapVectors procedure 154

T
terminating a program 31
text
attributes 62
files 28
TextAttr variable
ClrEol and 16
ClrScr and 17
HighVideo and 71
LowVideo and 86
NormVideo and 92
TextBackground and 155
TextColor and 156
TextBackground procedure 155
TextColor procedure 155
TextHeight function 156
TextMode procedure 157
TextWidth function 159
time procedures
GetFTime 52
GetTime 63
SetFTime 133
SetTime 144
Trunc function 160
Truncate procedure 160
Turbo Pascal 3.0
conversion, chaining 161
TypeOf function 160

U
UnpackTime procedure 161
untyped files, variable 10, 11
UpCase function 161

V
Val procedure 161
variables
disposing of 23, 40
DosError 30, 36, 37, 49, 52, 130, 133
increasing 72
untyped file 10, 11
VGA
driver, modifying palette entries for 139
modes 21
emulated 74
VGAHi 53, 135
VGALo 53, 135
VGAMed 53, 135
video operations
AssignCrt 8
ClrEol 16
ClrScr 17
DelLine procedure 20
GotoXY 67
HighVideo 71
InsLine 75
LowVideo 86
NormVideo 92
RestoreCrtMode 118
TextColor 155
WhereX 162
WhereY 163
Window 163
Write (text) 164
Write (typed) 166
Writeln 166

W
WhereX function 162
WhereY function 163
Window procedure 163
Write procedure
text files 164
typed files 166
write statements, DOS 7
Writeln procedure 166
writing records 11

X
XORPut constant 104

Z
Zenith Z-449, BGI and 74