```
3
10
5
16
8
4
2
1
Ok. Program stopped again.
```

The above program is actually a more complicated version of the following program:

```
program Catch222;
Var
  X: Integer;
begin
  Write('Enter any integer: ');
  Readln(X);
  while X <> 1 do
  begin
    if X mod 2 = 0 then X := X div 2 else X := X*3+1;
    Writeln(X);
  end;
  Write('Ok. Program stopped again.');
end.
```

It may interest you to know that it cannot be proved if this small and very simple program actually **will** stop for any integer!

Chapter 17
INCLUDING FILES

The fact that the TURBO editor performs editing only within memory limits the size of source code handled by the editor. The **I** compiler directive can be used to circumvent this restriction, as it provides the ability to split the source code into smaller 'lumps' and put it back together at compile-time. The include facility also aids program clarity, as commonly used subprograms, once tested and debugged, may be kept as a 'library' of files from which the necessary files can be included in any other program.

The syntax for the **I** compiler directive is:

```
{$I filename}
```

where *filename* is any legal file name. Leading spaces are ignored and lower case letters are translated to upper case. If no file type is specified, the default type **.PAS** is assumed. This directive must be specified on a line by itself.

Examples:

```
{$Ifirst.pas}
{$I COMPUTE.MOD}
{$iStdProc }
```

Notice that a space must be left between the file name and the closing brace if the file does not have a three-letter extension; otherwise the brace will be taken as part of the name.

To demonstrate the use of the include facility, let us assume that in your 'library' of commonly used procedures and functions you have a file called *STUPCASE.FUN*. It contains the function *StUpCase* which is called with a character or a string as parameter and returns the value of this parameter with any lower case letters set to upper case.

ClibPDF - www.fastio.com

File *STUPCASE.FUN:*

```
function StUpCase(St: AnyString): AnyString;
Var I: Integer;
begin
  for I := 1 to Length(St) do
    St[I] := UpCase(St[I]);
  StUpCase := St
end;
```

In any future program you write which requires this function to convert strings to upper case letters, you need only include the file at compile-time instead of duplicating it into the source code:

```
program Include Demo;
type
  InData= string[80];
  AnyString= string[255];
Var
  Answer: InData;
{$I STUPCASE.FUN}
begin
  ReadLn(Answer);
  Writeln(StUpCase(Answer));
end.
```

This method not only is easier and saves space; it also makes program updating quicker and safer, as any change to a 'library' routine will automatically affect all programs including this routine.

Notice that TURBO Pascal allows free ordering, and even multiple occurrences, of the individual sections of the declaration part. You may thus e.g. have a number of files containing various commonly used **type** definitions in your 'library' and include the ones required by different programs.

All compiler directives except **B** and **C** are local to the file in which they appear, i.e. if a compiler directive is set to a different value in an included file, it is reset to its original value upon return to the including file. **B** and **C** directives are always global. Compiler directives are described in Appendix C.

Include files cannot be nested, i.e. one include file cannot include yet another file and then continue processing.

Chapter 18
OVERLAY SYSTEM

The overlay system lets you create programs much larger than can be accommodated by the computer's memory. The technique is to collect a number of subprograms (procedures and functions) in one or more files separate from the main program file, which will then be loaded automatically one at a time into the **same** area in memory.

The following drawing shows a program using one overlay file with five overlay subprograms collected into one **overlay group**, thus sharing the same memory space in the main program:

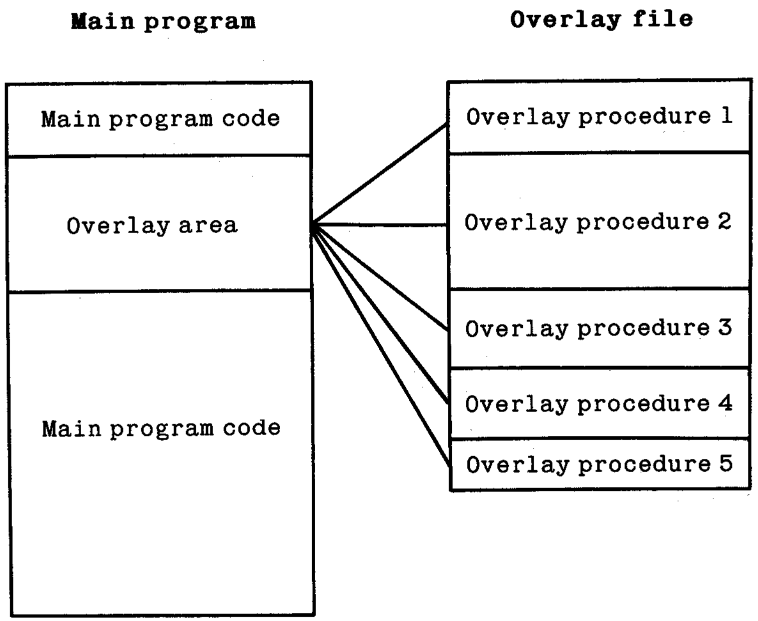

Figure 18-1 Principle of Overlay System

ClibPDF - www.fastio.com

When an overlay procedure is called, it is automatically loaded into the overlay area reserved in the main program. This 'gap' is large enough to accommodate the largest of the overlays in the group. The space required by the main program is thus reduced by roughly the sum of all subprograms in the group less the largest of them.

In the example above, overlay procedure 2 is the largest of the five procedures and thus determines the size of the overlay area in the main code. When it is loaded into memory, it occupies the entire overlay area:

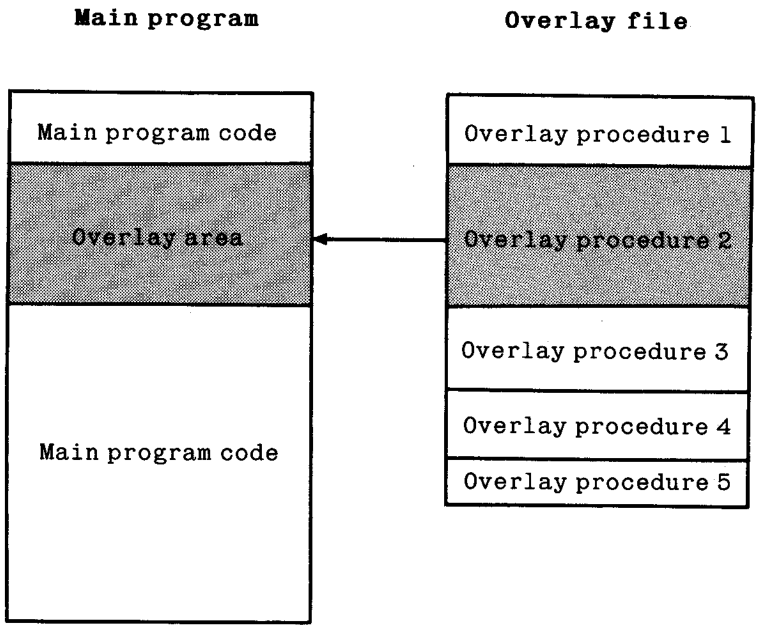

Figure 18-2: Largest Overlay Subprogram Loaded

The smaller subprograms are loaded into the same area of memory, each starting at the first address of the overlay area. Obviously they occupy only part of the overlay area; the remainder is unused:

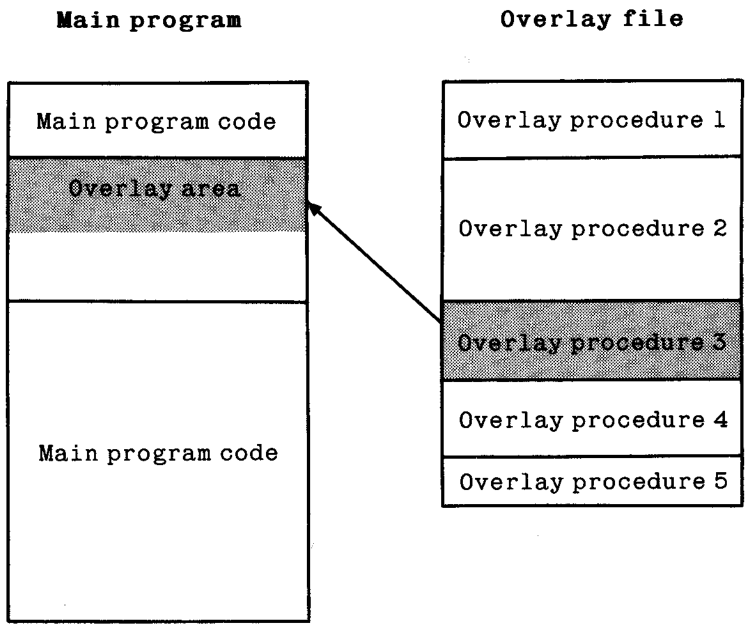

Figure 18-3: Smaller Overlay Subprogram Loaded

As procedures 1, 3, 4, and 5 execute in the same space as used by procedure 2, it is clear that they require no additional space in the main program. It is also clear that none of these procedures must ever call each other, as they are never present in memory simultaneously.

There could be many more overlay procedures in this group of overlays; in fact the total size of the overlay procedures could substantially exceed the size of the main program. And they would still require only the space occupied by the largest of them.

The tradeoff for this extra room for program code is the addition of disk access time each time a procedure is read in from the disk. With good planning, as discussed on page 155, this time is negligible.

ClibPDF - www.fastio.com

Creating Overlays

Overlay subprograms are created automatically, simply by adding the reserved word **overlay** to the declaration of any procedure or function:

```
overlay procedure Initialize;
```
and
```
overlay function TimeOfDay: Time;
```

When the compiler meets such a declaration, code is no longer output to the main program file, but to a separate overlay file. The name of this file will be the same as that of the main program, and the type will be a number designating the overlay group, ranging form 000 through 099.

Consecutive overlay subprograms will be grouped together. I.e. as long as overlay subprograms are not separated by any other declaration, they belong to the same group and are placed in the same overlay file.

Example 1:
```
overlay procedure One;
begin
  :
end;

overlay procedure Two;
begin
  :
end;

overlay procedure Three;
begin
  :
end;
```

These three overlay procedures will be grouped together and placed in the same overlay file. If they are the first group of overlay subprograms in a program, the overlay file will be no. *000.*

The three overlay procedures in the following example will be placed in consecutive overlay files, *.000* and *.001*, because of the declaration of a non-overlay procedure *Count* separating overlay procedures *Two* and *Three*.

The separating declaration may be any declaration, for example a dummy **type** declaration, if you want to force a separation of overlay areas.

Example 2:
```
overlay procedure One;
begin
  :
end;

overlay procedure Two;
begin
  :
end;

procedure Count;
begin
  :
end

overlay procedure Three;
begin
  :
end;
```

A separate overlay area is reserved in the main program code for each overlay group, and the following files will be created:

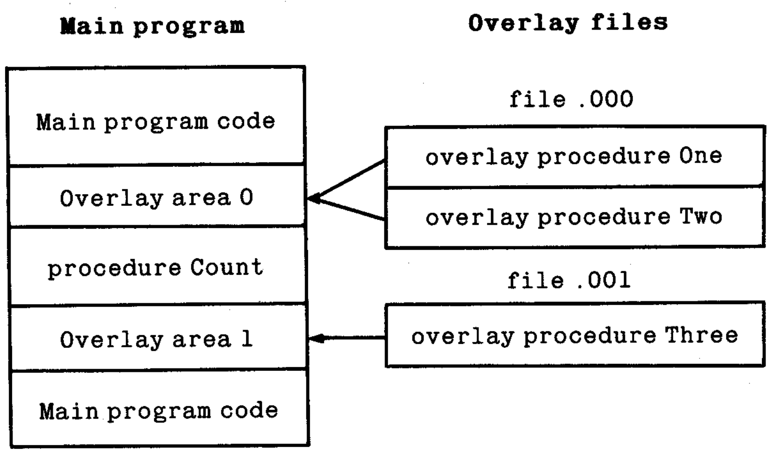

Figure 18-4: Multiple Overlay Files

ClibPDF - www.fastio.com

Nested Overlays

Overlay subprograms may be nested, i.e. an overlay subprogram may itself contain overlay subprograms which may contain overlay subprograms, etc.

Example 3:

```
program OverlayDemo;
  :
  :
overlay procedure One;
begin
  :
end;

overlay procedure Two;
  overlay procedure Three;
  begin
    :
  end;
begin
  :
end;
  :
```

In this example, two overlay files will be created. File *.000* contains overlay procedures *One* and *Two*, and an overlay area is reserved in the main program to accommodate the largest of these. Overlay file *.001* contains overlay procedure *Three* which is local to overlay procedure *Two*, and an overlay area is created in the code of overlay procedure *Two*:

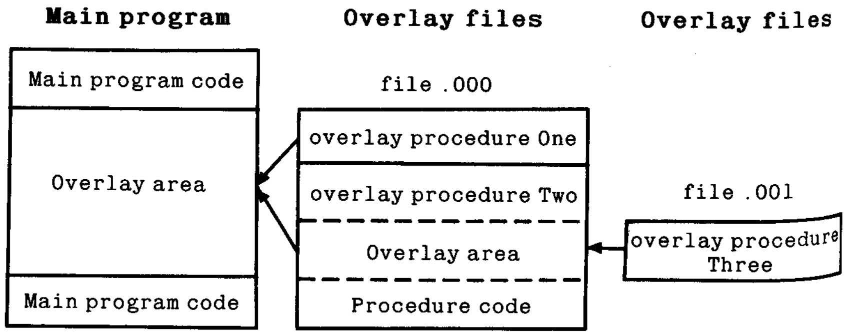

Figure 18-5: Nested Overlay Files

Automatic Overlay Management

An overlay subprogram is loaded into memory only when called. On each call to an overlay subprogram, a check is first made to see if that subprogram is already present in the overlay area. If not, it will automatically be read in from the appropriate overlay file.

Placing Overlay Files

During compilation, overlay files will be placed on the logged drive, i.e. on the same drive as the main program file (.COM or .CMD file).

During execution, the system normally expects to find its overlay files on the logged drive. This may be changed as described on pages 196 (PC/MS-DOS), 233 (CP/M-86), and 265 (CP/M-80).

Efficient Use of Overlays

The overlay technique, of course, adds overhead to a program by adding some extra code to manage the overlays, and by requiring disk accesses during execution. Overlays, therefore, should be carefully planned.

In order not to slow down execution excessively, an overlay subprogram should not be called too often, or - if one **is** called often - it should at least be called without intervening calls to other subprograms in the same overlay file in order to keep disk accesses at a minimum. The added time will of course vary greatly, depending on the actual disk configuration. A 5 1/4" floppy will add much to the run-time, a hard disk much less, and a RAM-disk, as used by many, very little.

To save as much space as possible in the main program, one group of overlays should contain as many individual subprograms as possible. From a pure space-saving point of view, the more subprograms you can put into a single overlay file, the better. The overlay space used in the main program needs only accommodate the largest of these subprograms - the rest of the subprograms have a free ride in the same area of memory. This must be weighed against the time considerations discussed above.

ClibPDF - www.fastio.com

Restrictions Imposed on Overlays

Data Area

Overlay subprograms in the same group share the same area in memory and thus cannot be present simultaneously. They must therefore **not** call each other. Consequently, they may share the same data area which further adds to the space saved when using overlays (CP/M-80 version only).

In example 1 on page 152, none of the procedures may therefore call each other. In example 2, however, overlay procedures *One* and *Two* may call overlay procedure *Three*, and overlay procedure *Three* may call each of the other two, because they are in separate files and consequently in separate overlay areas in the main program.

Forward Declarations

Overlay subprograms may not be **forward** declared. This restriction is easily circumvented, however, by **forward** declaring an ordinary subprogram which then in turn calls the overlay subprogram.

Recursion

Overlay subprograms cannot be recursive. Also this restriction may be circumvented by declaring an ordinary recursive subprogram which then in turn calls the overlay subprogram.

Run-Time Errors

Run-time errors occurring in overlays are found as usual, and an address is issued by the error handling system. This address, however, is an address within the overlay area, and there is no way of knowing which overlay subprogram was actually active when the error occurred.

Run-time errors in overlays can therefore not always be readily found with the Options menu's 'Find run-time error' facility. What 'Find run-time error' will point out is the first occurrence of code at the specified address. This, of course, **may** be the place of the error, but the error may as well occur in a subsequent subprogram within the same overlay group.

This it not a serious limitation, however, as the type of error and the way it occurs will most often indicates in which subprogram the error happened. The way to locate the error precisely is then to place the suspected subprogram as the first subprogram of the overlay group. 'Find run-time error' will then work.

The best thing to do is not to place subprograms in overlays until they have been fully debugged!

ClibPDF - www.fastio.com

Notes:

Chapter 19
IBM PC GOODIES

This chapter applies to the IBM PC-versions only, and the functions described can be expected to work on IBM PC and compatibles only! If you have problems on a compatible, it's not as compatible as you thought.

Screen Mode Control
TURBO provides a number of procedures to control the PC's various screen modes.

Windows
The window routines let you declare a smaller part of the screen to be your actual work area, protecting the rest of the screen from being overwritten.

Basic graphics
These built-in graphics routines let you plot points and draw lines in different colors.

Extended graphics
A set of external graphics routines allow for more advanced graphics. One simple statement includes these routines in your programs.

Turtlegraphics
The same external machine language file also provides you with turtle-graphics routines.

Sound
Standard procedures are provided which let you use the PC's sound capabilities in an easy way.

Keyboard
A number of the special keys of the IBM keyboard are installed as primary commands for the editor. These commands are listed on page 186, and you may add more if you wish. The secondary *WordStar* commands are still available.

ClibPDF - www.fastio.com

Screen Mode Control

The IBM PC gives you a choice of screen modes, each with its own characteristics. Some display characters, some display graphics, and they all have different capabilities of showing colors. TURBO Pascal supports all these screen formats and provides an easy way of using them.

The following screen modes are available:

TextMode	25 lines of 40 or 80 characters
GraphColorMode	320x200 dots color graphics
GraphMode	320x200 dots black & white graphics (color on an RGB monitor)
HiRes	640x200 dots black + one color graphics

Text Modes

In text mode, the PC will display 25 lines of either 40 or 80 characters. The procedure to invoke this mode is named *TextMode* and is called as follows:

```
TextMode;
TextMode(BW40);      BW40 is an integer constant with the value 0
TextMode(C40);       C40 is an integer constant with the value 1
TextMode(BW80);      BW80 is an integer constant with the value 2
TextMode(C80);       C80 is an integer constant with the value 3
```

The first example with no parameters invokes the text mode which was active last, or the one that is currently active. The next two examples activate black and white text modes with 40 and 80 characters on each line. The final two examples activate color text modes with 40 and 80 characters on each line. Calling *TextMode* will clear the screen.

TextMode should be called before exiting a graphics program in order to return the system to text mode.

Color Modes

In the color text modes, each character may be chosen to be one of 16 colors, and the background may be one of 8 colors. The colors are referred to by the numbers 0 through 15. To make things easier, TURBO Pascal includes 16 pre-defined integer constants which may be used to identify colors by names:

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

Table 19-1: Text Mode Color Scale

Characters may be any of these colors, whereas the background may be any of the dark colors. Notice that some monitors do not recognize the intensity signal used to create the eight light colors. On such monitors, the light colors will be displayed as their dark equivalents.

TextColor

Syntax: TextColor(*Color*);

This procedure selects color of the **characters**. *Color* is an integer expression in the range 0 through 15, selecting character colors from the table given above.

Examples:

```
TextColor(1);           selects blue characters
TextColor(Yellow);      selects yellow characters
```

The characters may be made to blink by adding 16 to the color number. There is a pre-defined constant *Blink* for this purpose:

```
TextColor(Red + Blink);     selects red, blinking characters
```


ClibPDF - www.fastio.com

TextBackground

Syntax: TextBackground(*Color*);

This procedure selects color of the **background**, that is, the cell immediately surrounding each character; the entire screen consists of 40 or 80 by 25 such cells. *Color* is an integer expression in the range 0 through 7, selecting character colors from the table given above.

Examples:

`TextBackground(4);`	selects red background
`TextBackground(Magenta);`	selects magenta background

Cursor Position

In text mode, two functions will tell you where the cursor is positioned on the screen:

WhereX

Syntax: WhereX;

This integer function returns the X-coordinate of the current cursor position.

WhereY

Syntax: WhereY;

This integer function returns the Y-coordinate of the current cursor position.

Graphics Modes

With a standard IBM graphics video board, or one that is compatible, TURBO will do graphics. Three modes are supported:

GraphColorMode	320x200 dots color graphics
GraphMode	320x200 dots black & white graphics
HiRes	640x200 dots black + one color graphics

The upper, left corner of the screen is coordinate 0,0. X coordinates stretch to the right, Y coordinates downward. All drawing is 'clip-ped', that is, anything displayed outside the screen will be ignored (except when the turtlegraphics' *Wrap* is in effect).

Activating one of the graphics modes will clear the screen. The standard procedure *ClrScr* works only in text mode, so the way to clear a graphics screen is to activate a graphics mode, possibly the one that's already active. With extended graphics and turtlegraphics, however, there is a *ClearScreen* procedure which clears the active window.

Graphics can be mixed with text. In 320 x 200 modes, the screen can display 40 x 25 characters and in 640 x 200 mode, it can display 80 x 25 characters.

The *TextMode* procedure should be called before exiting a graphics program in order to return the system to text mode, see page 160).

GraphColorMode

Syntax: GraphColorMode;

This standard procedure activates the 320x200 dots color graphics screen giving you X-coordinates between 0 and 319 and Y-coordinates between 0 and 199. Drawings may use colors selected from the palette described on page 165.

ClibPDF - www.fastio.com

GraphMode

Syntax: GraphMode;

This standard procedure activates the 320x200 dots black and white graphics screen giving you X-coordinates between 0 and 319 and Y-coordinates between 0 and 199. On a RGB monitor like the IBM Color/Graphics Display, however, even this mode displays colors from a limited palette as shown on page 166.

HiRes

Syntax: HiRes;

This standard procedure activates the 640x200 dots high resolution graphics screen giving you X-coordinates between 0 and 639 and Y-coordinates between 0 and 199. In high resolutions graphics, the background (screen) is always black, and you draw in one color set by the *HiResColor* standard procedure.

HiResColor

Syntax: HiResColor(*Color*);

This standard procedure selects the color used for drawing in high resolution graphics. *Color* is an integer expression in the range 0 through 15. The background (screen) is always black. Changing *HiResColor* causes anything already on the screen to change to the new color.

Examples:

`HiResColor(7);`	selects light gray
`HiResColor(Blue);`	selects blue

This one color may be chosen from the following 16 colors:

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

Table 19-2: High Resolution Graphics Color Scale

Some monitors do not recognize the intensity signal used to create the eight light colors. On such monitors, the light colors will be dis-played as their dark equivalents.

Palette

Syntax: Palette(*N*);

This procedure activates the color palette indicated by the integer expression *N*. with a parameter specifying the number of the palette. Four color palettes exist, each containing three colors (1-3) and a fourth color (0) which is always equal to the background color (see later):

Color number:	0	1	2	3
Palette 0	Background	Green	Red	Brown
Palette 1	Background	Cyan	Magenta	LightGray
Palette 2	Background	LightGreen	LightRed	Yellow
Palette 3	Background	LightCyan	LightMagenta	White

Table 19-3: Color Palettes in Color Graphics

ClibPDF - www.fastio.com

The graphics routines will use colors from this palette. They are called with a parameter in the range 0 through 3, and the color actually used is selected from the active palette:

`Plot(X,Y,2)`	will plot a red point when palette 0 is active.
`Plot(X,Y,3)`	will plot a yellow point when palette 2 is active.
`Plot(X,Y,0)`	will plot a point in the active background color, in effect erasing that point.

Once a drawing is on the screen, a change of palette will cause all colors on the screen to change to the colors of the new palette. Only three colors plus the color of the background may thus be displayed simultaneously.

The *GraphMode* supposedly displays only black and white graphics, but on on an RGB monitor, like the IBM Color/Graphics Display, even this mode displays the following limited palette:

Color number:	0	1	2	3
Palette 0	Background	Blue	Red	LightGray
Palette 1	Background	LightBlue	LightRed	White

Table 19-4: Color Palettes in B/W Graphics

GraphBackground

Syntax: GraphBackground(*Color*);

This standard procedure sets the the background color, that is the entire screen, to any of 16 colors. *Color* is an integer expression in the range 0 through 1

`GraphBackground(0);`	sets the screen to black
`GraphBackground(11);`	sets the screen to light cyan

The following color numbers and pre-defined constants are available:

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

Table 19-5: Graphics Background Color Scale

Some monitors do not recognize the intensity signal used to create the eight light colors. On such monitors, the light colors will be displayed as their dark equivalents.

ClibPDF - www.fastio.com

Windows

TURBO Pascal lets you declare windows anywhere on the screen. When you write in such a window, the window behaves exactly as if you were using the entire screen, leaving the rest of the screen untouched.

Text Windows

The *Window* procedure allows you to define any area on the screen as the active window in text mode:

```
Window(X1,Y1,X2,Y2);
```

where X1 and Y1 are the absolute coordinates of the upper left corner of the window, X2 and Y2 are the absolute coordinates of the lower right corner. The minimum size of the text window is 2 columns by 2 lines.

The default window is *1,1,80,25* in 80-column modes and *1,1,40,25* in 40-column modes, that is, the entire screen.

All screen coordinates (except the window coordinates themselves) are relative to the active window. This means that after the statement:

```
Window(20,8,60,17);
```

which defines the center portion of the physical screen to be your active window, screen coordinates 1,1 (upper left corner) are now the upper left corner of the *window*, not of the physical screen:

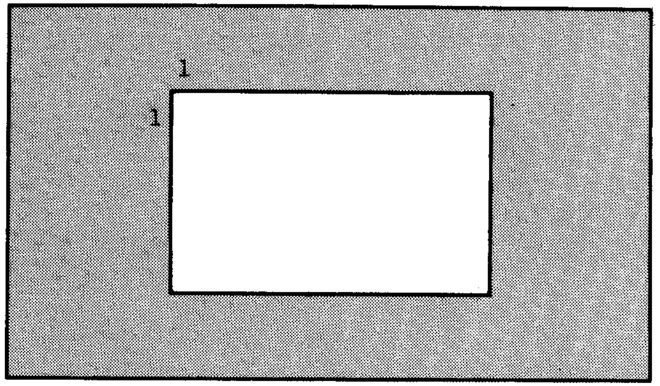

Figure 19-1: Text Windows

The screen outside the window is simply not accessible, and the window behaves as it were the entire screen. You may insert, delete, and scroll lines, and lines will wrap around if too long.

Graphics Windows

The *GraphWindow* procedure allows you to define an area of the screen as the active window in any of the graphics modes:

```
GraphWindow(X1,Y1,X2,Y2);
```

where X1 and Y1 are the absolute coordinates of the upper left corner of the window, X2 and Y2 are the absolute coordinates of the lower right corner.

The default graphics window is *0,0,319,199* in 320x200-dot modes and *0,0,639,199* in 640x200-dot mode, that is, the entire screen.

ALL screen coordinates are relative to the active *window*—not to the physical screen. For example, after:

```
GraphWindow(50,100,200,180);
```

coordinate 0,0 is in the upper left corner of the window.

Windows cause graphics to be 'clipped', that is, if you for example *Draw* between two coordinates outside the window, only the part of the line that falls within the window will be shown:

ClibPDF - www.fastio.com

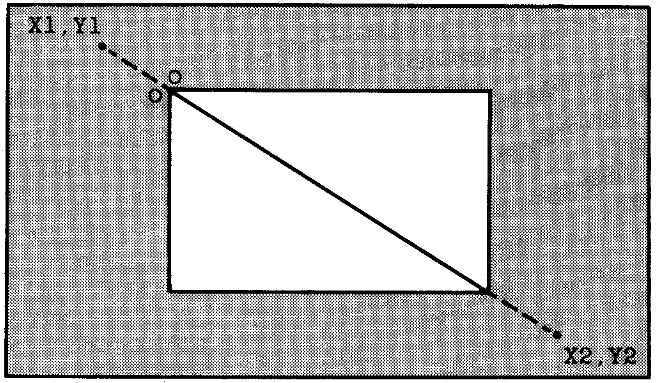

Figure 19-2: Graphics Windows

Basic Graphics

In each of the graphics modes, TURBO Pascal provides standard procedures which will plot points at specified coordinates and draw lines between two coordinates:

Plot

Syntax: Plot(*X, Y, Color*);

Plots a point at the screen coordinates specified by *X* and *Y* in the color specified by *Color*. *X, Y*, and *Color* are integer expressions.

Draw

Syntax: Draw(X1,Y1,X2,Y2,Color);

Draws a line between the screen coordinates specified by *X1,Y1* and *X2,Y2* in the color specified by *Color*. All parameters are integer expressions.

ClibPDF - www.fastio.com

Extended Graphics

TURBO Pascal comes with a set of external machine language routines that can be included in TURBO programs during compilation. They provide extended graphics commands as described in the following.

The external graphics routines are contained in the file GRAPH.BIN. The file GRAPH.P contains the necessary **external** declarations, and the extended graphics routines are included in a TURBO program simply by using this statement to include the GRAPH.P file in the program:

```
{$I GRAPH.P }
```

ColorTable

Syntax: ColorTable(*C1,C2,C3,C4*);

ColorTable supplements *Palette* by defining a color 'translation table' which lets the current color of any given point determine the new color of that point when it is written again. The default color table value is (0,1,2,3), which means that when a point is written on the screen, it does not change the color that's already there:

color 0 becomes color 0
color 1 becomes color 1
color 2 becomes color 2
color 3 becomes color 3

The table (3,2,1,0) would cause

color 0 to become color 3
color 1 to become color 2
color 2 to become color 1
color 3 to become color 0

that is, all colors would be reversed. The *PutPic* procedure always uses the color table; all other draw procedures use the table if a color of −1 is specified, for example:

```
Plot(X,Y,-1);
```


Arc

Syntax: Arc(*X,Y,Angle,Radius,Color*);

Draws an arc of *Angle* degrees, starting at the position given by *X,Y*, with a radius given by *Radius*. If *Angle* is positive, the arc turns clockwise; if it is negative, the arc turns counterclockwise. If *Color* is from 0 through 3, the pen color is selected from the color palette (see page 165); if it is −1, the color is selected from the color translation table defined by the *ColorTable* procedure (page 172).

Circle

Syntax: Circle(*X,Y,Radius,Color*);

Draws a circle in the color given by *Color* with its center at *X,Y* and a radius as specified by *Radius*.

The radius of the circle is the same in the horizontal and vertical axes. In 320 x 200 mode this draws a perfect circle, as the display is almost linear. In 640 x 200 mode, however, circles appear as ellipses.

If *Color* is from 0 through 3, the pen color is selected from the color palette (see page 165); if it is −1, the color is selected from the color translation table defined by the *ColorTable* procedure (page 172).

GetPic

Syntax: GetPic(*Buffer,X1,Y1,X2,Y2*);

Copies the contents of a rectangular area defined by the integer expressions *X1,Y1,X2,Y2* into the variable *Buffer*, which may be of any type. The minimum buffer size in bytes required to store the image is calculated as:

320 x 200 modes:
  Size = ((Width + 3) div 4)*Height*2 + 6

640 x 200 modes:
  Size = ((Width + 7) div 8)*Height + 6

ClibPDF - www.fastio.com

where:

Width = abs(x1-x2) + 1 and Height = abs(y1-y2) + 1

Note that it the responsibility of the programmer to ensure that the buffer is large enough to accommodate the entire transfer.

The first 6 bytes of the buffer constitute a three word header (three integers). After the transfer, the first word contains 2 in 320 x 200 mode or 1 in 640 x 200 mode The second word contains the width of the image and third contains the height. The remaining bytes contain the data. Data is stored with the leftmost pixels in the most significant bits of the bytes. At the end of each row, the remaining bits of the last byte are skipped.

PutPic

Syntax: PutPic(*Buffer,X,Y*);

Copies the contents of the variable *Buffer* onto a rectangular area on the screen. The integer expressions *X* and *Y* define the lower left-hand corner of the picture area. *Buffer* is a variable of any type, in which a picture has previously been stored by *GetPic*. Each bit in the buffer is converted to a color according to the color map before it is written to the screen.

GetDotColor

Syntax: GetDotColor(*X,Y*);

This integer function returns the color value of the dot located at coordinate *X,Y*. Values of 0 through 3 may be returned in 320 x 200 dot graphics, and 0 or 1 in 640 x 200 dot graphics. If *X,Y* is outside the window, *GetDotColor* returns −1.

FillScreen

Syntax: FillScreen(*Color*);

Fills the entire active window with the color specified by the integer expression *Color*. If *Color* is in the range 0 through 3, the color will be selected from the color palette, if it is −1, the color table will be used. This allows for dramatic effects; with a color table of *3,2,1,0*, for example, *FillScreen(−1)* will invert the entire image within the active window.

FillShape Procedure

Syntax: FillShape(*X,Y,FillColor,BorderColor*);

Fills an area of any shape with the color specified by the integer expression *FillColor* which must be in the range 0 through 3. The color translation table is not supported. The shape must be entirely enclosed by the color specified by *BorderColor*; if not, *FillShape* will 'spill' onto the area outside the shape. *X* and *Y* are the coordinates of a point within the image to be filled.

FillPattern

Syntax: FillPattern(*X1,Y1,X2,Y2,Color*);

Fills a rectangular area defined by the coordinates *X1,Y1,X2,Y2* with the pattern defined by the *Pattern* procedure. The pattern is replicated both horizontally and vertically to fill the entire area. Bits of value 0 cause no change to the display, whereas bits of value 1 cause a dot to be written using the color selected by *Color*.

ClibPDF - www.fastio.com

Pattern

Syntax: Pattern(*P*);

Defines the pattern used by the *FillPattern* procedure. The pattern is an 8 x 8 matrix defined by the *P* parameter which must be of type **array**[0..7] **of** *Byte*. Each byte corresponds to a horizontal line in the pattern, and each bit corresponds to a pixel. The following shows some sample patterns and the hexadecimal value of each line in the matrix. A hyphen represents a binary 0, and an asterisk represents a binary 1.

```
- * - - - * - -   $44          * - * - * - * -   $AA
* - - - * - - -   $88          - * - * - * - *   $55
- - - * - - - *   $11          * - * - * - * -   $AA
- - * - - - * -   $22          - * - * - * - *   $55
- * - - - * - -   $44          * - * - * - * -   $AA
* - - - * - - -   $88          - * - * - * - *   $55
- - - * - - - *   $11          * - * - * - * -   $AA
- - * - - - * -   $22          - * - * - * - *   $55
```

To use the first pattern, the slanted lines, the following typed constant could be declared and passed as a parameter to *Pattern*:

```
const
  Lines: array[0..7] of Byte =
         ($44,$88,$11,$22,$44,$88,$11,$22);
```

When the pattern is used by the *FillPattern* procedure, low bits cause no change to the display, high bits cause a dot to be written.

Turtlegraphics

The external file GRAPH.BIN that contains the extended graphics routines mentioned in the previous section also contains the TURBO Turtlegraphics routines, so whenever you include the graphics declaration file GRAPH.P:

```
{$I GRAPH.P }
```

you also have access to the turtlegraphics described in the following.

TURBO Turtlegraphics is based on the 'turtle' concept devised by S. Papert and his co-workers at MIT. To make graphics easy for those of us who might have difficulty understanding cartesian coordinates, Papert et al. invented the idea of a 'turtle' that could 'walk' a given distance and turn through a specified angle, drawing a line as it went along. Very simple algorithms in this system can create more interesting images than an algorithm of the same length in cartesian coordinates.

Like the other graphics routines, turtlegraphics operate within a window. This window is set to the entire screen by default but the *Window* or *TurtleWindow* procedures can be used to define only part of the screen as the active graphics area, safeguarding the rest from being overwritten. Turtlegraphics and ordinary graphics can be used simultaneously, and they share a common window.

The TURBO Turtlegraphics routines operate on *turtle coordinates*. The turtle's *home* position (0,0) in this coordinate system is always in the middle of the active window, with positive values stretching to the right (X) and upwards (Y), and negative values stretching to the left (X) and downwards (Y):

ClibPDF - www.fastio.com

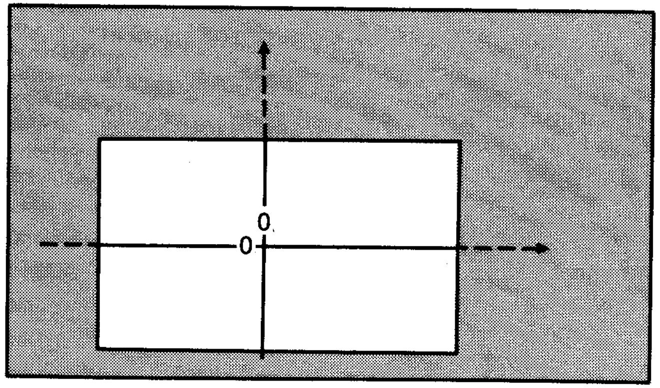

Figure 19-3: Turtle Coordinates

The range of coordinates on a full screen is:

320 x 200 modes:	X = −159..0..160, Y = −99..0..100
640 x 200 mode:	X = −319..0..320, Y = −99..0..100

but the actual range will be limited to the size of the active window. Coordinates outside the active window are legal, but will be ignored. This means that drawings are 'clipped' to the limits of the active window.

Back

Syntax: Back(*Dist*);

Moves the turtle backwards the distance given by the integer expression *Dist* from its current position in the direction opposite to the the turtle's current heading while drawing a line in the current pen color (if *Dist* is is negative, the turtle moves forwards).

ClearScreen

Syntax: ClearScreen;

This procedure clears the active window and homes the turtle.

Forwd

Syntax: Forwd(*Dist*);

Moves the turtle forwards the distance given by the integer expression *Dist* from its current position in the direction the turtle is currently facing, while drawing a line in the current pen color (if *Dist* is is negative, the turtle moves backwards).

Heading

Syntax: Heading;

The *Heading* function returns an integer in the range 0..359 giving the direction in which the turtle is currently pointing. 0 is upwards, and increasing angles represent headings in clockwise direction.

HideTurtle

Syntax: HideTurtle;

Hides the turtle, so that it is not shown on the screen. This is the initial state of the turtle, so to see the turtle, you must first call the *ShowTurtle* procedure.

Home

Syntax: Home;

This procedure puts the turtle to its home position at turtle coordinates 0,0 (the middle of the active window), and points it in heading 0 (upwards).

ClibPDF - www.fastio.com

NoWrap

Syntax: NoWrap;

This procedure disables the turtle from 'wrapping', that is, re-appear ing at the opposite side of the active window if it exceeds the window boundary. *NoWrap* is the system's initial value.

PenDown

Syntax: PenDown;

This procedure 'puts the pen down' so that when the turtle moves, it draws a line. This is the initial status of the pen.

PenUp

Syntax: PenUp;

This procedure 'lifts the pen' so the turtle moves without drawing a line.

SetHeading

Syntax: SetHeading(*Angle*);

Turns the turtle to the angle specified by the integer expression *Angle.* 0 is upwards, and increasing angles represent clockwise rotation. If *Angle* is not in the range 0..359, it is converted into a number in that range.

Four integer constants are pre-defined to easily turn the turtle in the four main directions: *North* = 0 (up), *East* = 90 (right), *South* = 180, and *West* = 270 (left).

SetPenColor

Syntax: SetPenColor(*Color*);

Selects the color of the 'pen', that is, the color that will be used for drawing when the turtle moves. *Color* is an integer expression yielding a value between −1 and 3. If *Color* is from 0 through 3, the pen color is selected from the color palette (see page 165); if it is −1, the color is selected from the color translation table defined by the *ColorTable* procedure (page 172).

SetPosition

Syntax: SetPosition(*X, Y*);

Moves the turtle to the location with coordinates given by the integer expressions *X* and *Y* without drawing a line.

ShowTurtle

Syntax: ShowTurtle;

Displays the turtle as a small triangle. The turtle is initially **hidden**, so to see the turtle, you must first call this procedure.

TurnLeft

Syntax: TurnLeft(*Angle*);

Turns the turtle *Angle* degrees from its current direction. Positive angles turn the turtle to the left, negative angles turn it to the right.

TurnRight

Syntax: TurnRight(*Angle*);

Turns the turtle *Angle* degrees from its current direction. Positive angles turn the turtle to the right, negative angles turn it to the left.

ClibPDF - www.fastio.com

TurtleWindow

Syntax: TurtleWindow(*X,Y,W,H*);

The *TurtleWindow* procedure defines an area of the screen as the active graphics area in any of the graphics modes, exactly as does the *Window* procedure. *TurtleWindow*, however, lets you define the window in terms of *turtle coordinates*, which are more natural to use in turtlegraphics. *X* and *Y* are the screen coordinates of the center of the window; *W* is its width, and *H* is its height.

The default TurtleWindow is *159,99,320,200* in 320x200-dot modes and *319,99,640,200* in 640x200-dot mode, that is, the entire screen. If the turtlewindow is defined to fall partly outside the physical screen, it is clipped the edges of the physical screen.

Turtlegraphics are 'clipped' to the active window, that is, if you move the turtle outside the active window, it will not be shown and it will not draw.

When the window is set (whether by *TurtleWindow* or by *Window*, the turtle is initialized to its *Home* position and heading. Changing screen mode resets the window to the entire screen.

Turtlegraphics operate *in turtle coordinates*. The turtle's *home* position (0,0) in this coordinate system is always in the middle of the active window, with positive values stretching to the right (X) and upwards (Y), and negative values stretching to the left (X) and downwards (Y):

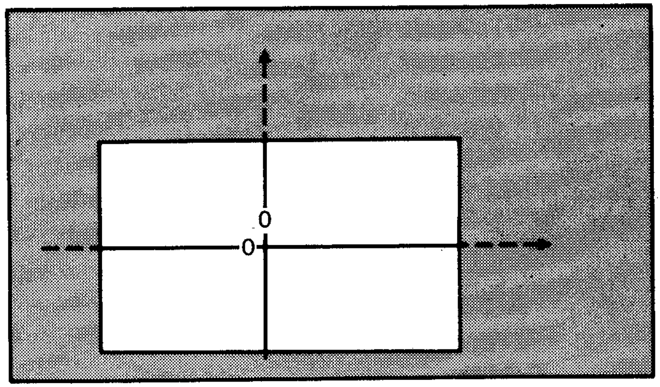

Figure 19-4: Turtle Coordinates

The range of coordinates on a full screen is:

320 x 200 modes:	X = -159..0..160, Y = -99..0..100
640 x 200 mode:	X = -319..0..320, Y = -99..0..100

but the actual range will be limited to the size of the active window.

Coordinates outside the active window are legal, but will be ignored. This means that drawings are 'clipped' to the limits of the active window, and anything drawn outside of the active window is lost.

TurtleThere

Syntax: TurtleThere;

This boolean function returns *True* if the turtle is visible in the active window (after a *ShowTurtle*), otherwise it returns *False*.

TurtleDelay

Syntax: TurtleDelay(Ms);

This procedure sets a delay in milliseconds between each step of the turtle. Normally, there is no delay.

ClibPDF - www.fastio.com

Wrap

Syntax: Wrap;

After a call to this procedure, the turtle will re-appear at the opposite side of the active window when it exceeds the window boundary. Use *NoWrap* to return to normal.

Xcor

Syntax: Xcor;

This function returns the integer value of the turtle's current X-coordinate.

Ycor

Syntax: Ycor;

This function returns the integer value of the turtle's current Y-coordinate.

Sound

The PC's speaker is accessed through the standard procedure *Sound*:

```
Sound(I);
```

where *I* is an integer expression specifying the frequency in Hertz. The specified frequency will be emitted until the speaker is turned off with a call to the *NoSound* standard procedure:

```
NoSound
```

The following example program will emit a 440-Hertz beep for half a second:

```
begin
  Sound(440);
  Delay(500);
  NoSound;
end.
```


ClibPDF - www.fastio.com

Editor Command Keys

In addition to the *WordStar* commands, the editing keys of IBM PC keyboard have been implemented as primary commands. This means that while e.g. Ctrl-E, Ctrl-X, Ctrl-S, and Ctrl-D still move the cursor up, down, left, and right, you may also use the arrows on the numeric keypad. The following table provides an overview of available editing keys, their functions, and their *WordStar*-command equivalents:

ACTION	PC-KEY	COMMAND
Character left	Left arrow	Ctrl-S
Character right	Right arrow	Ctrl-D
Word left	Ctrl-left arrow	Ctrl-A
Word right	Ctrl-right arrow	Ctrl-F
Line up	Up arrow	Ctrl-E
Line down	Down arrow	Ctrl-X
Page up	PgUp	Ctrl-R
Page down	PgDn	Ctrl-C
To left on line	Home	Ctrl-Q-S
To right on line	End	Ctrl-Q-D
To top of page	Ctrl-Home	Ctrl-Q-E
To bottom of page	Ctrl-End	Ctrl-Q-X
To top of file	Ctrl-PgUp	Ctrl-Q-R
To end of file	Ctrl-PgDn	Ctrl-Q-C
Insert mode on/off	Ins	Ctrl-V
Mark block begin	F7	Ctrl-K-B
Mark block end	F8	Ctrl-K-K
Tab	<TAB>	Ctrl-I

Table 19-6: IBM PC Keyboard Editing Keys

Note that while maintaining *WordStar* compatibility in the commands, some function keys have different meanings in *WordStar* and TURBO.

Chapter 20
PC-DOS AND MS-DOS

This chapter describes features of TURBO Pascal specific to the PC-DOS and MS-DOS implementations. It presents two kinds of information:

1) Things you should know to make efficient use of TURBO Pascal. Pages 187 through 209.

2) The rest of the chapter describes things which are of interest only to experienced programmers, such as machine language routines, technical aspects of the compiler, etc.

Tree-Structured Directories

On the Main Menu

The DOS structured directories are supported by TURBO's main menu:

```
Logged drive: A
Active directory: \

Work file:
Main file:

Edit      Compile  Run    Save
Dir       Quit  compiler Options

Text:      0 bytes
Free: 62903 bytes

> ■
```

Figure 20-1: TURBO Main Menu

ClibPDF - www.fastio.com

Notice the addition of the **A** command which lets you change the Active directory using the same path description as with the CHDIR command of DOS. The currently active directory is shown after the colon.

DOS uses a backslash: \ to refer to the ROOT directory, as shown in the example. The rest of directories have names just like files, that is a 1-8 letter name optionally followed by a period and a 1-3 letter type. Each directory can contain ordinary files or other directories.

Files in this system of directories are referenced by a **path** name in addition to the file name. A path name consists of the names of the directories leading to the file, separated by backslashes. The complete reference to a file called INVADERS.PAS in the directory TURBO is thus:

```
\TURBO\INVADERS.PAS
```

The first backslash indicates that the path starts from the root directory. If you were logged on some other directory, and you wanted to move to the TURBO directory, you would press **A** and enter:

```
\TURBO
```

In every sub-directory you will see two special entries in a DIR listing: . and .. The one period serves to identify this directory as a sub-directory. The two periods is a reference to the directory's 'parent' directory. These two periods may be used in a directory path; if, for example, you are logged on a sub-directory of TURBO, you may return to TURBO by pressing **A** and then entering the two periods.

Directory-related procedures

TURBO Pascal provides the following procedures to manipulate the tree-structured directories of MS-DOS.

ChDir

Syntax: ChDir(*St*);

Changes the current directory to the path specified by the string expression *St*. Also changes the logged drive if *St* contains a file name. For example:

```
ChDir('B:\PROG');
```

MkDir

Syntax: MkDir(*St*);

Creates a new sub-directory as specified by the path given by the string expression *St*. The last item in the path must be an non-existing filename.

RmDir

Syntax: RmDir(*St*);

Removes the sub-directory specified by the path given by the string expression *St*.

GetDir

Syntax: GetDir(*Dr*,*St*);

Returns the current directory of the drive indicated by *Dr* in the string variable *St*. *Dr* is an integer expression where 0 = logged drive, 1 = A, etc.

ClibPDF - www.fastio.com

Compiler Options

The **O** command selects the following menu from which you may view and change some default values of the compiler. It also provides a helpful function to find runtime errors in programs compiled into object code files.

```
compile -> Memory
           Com-file
           cHn-file

command line Parameter:

Find run-time error  Quit
```

Figure 20-2: Options Menu

Memory / Com file / cHn-file

The three commands **M**, **C**, and **H** select the compiler mode, i.e. where to put the code which results from the compilation. **M**emory is the default mode. When active, code is produced in memory and resides there ready to be activated by a **R**un command.

Com-file is selected by pressing **C**. The arrow moves to point to this line. The compiler writes code to a file with the same name as the Work file (or Main file, if specified) and the file type .COM. This file contains the program code and Pascal runtime library, and may be activated by typing its name at the console.

c**H**ain-file is selected by pressing **H**. The arrow moves to point to this line. The compiler writes code to a file with the same name as the Work file (or Main file, if specified) and the file type .CHN. This file contains the program code but no Pascal library and must be activated from another TURBO Pascal program with the *Chain* procedure (see page 193).

When the **C**om or c**H**n mode is selected, four additional lines will appear on the screen:


```
minimum cOde segment size:   XXXX paragraphs (max.YYYY)
minimum Data segment size:   XXXX paragraphs (max.YYYY)
mInimum free dynamic memory: XXXX paragraphs
mAximum free dynamic memory: XXXX paragraphs
```

Figure 20-3: Memory Usage Menu

The use of these commands is described in the following sections.

Minimum Code Segment Size

The **O**-command is used to set the minimum size of the code segment for a .COM using *Chain* or *Execute*. As discussed on page 193 , *Chain* and *Execute* do not change the base addresses of the code, data, and stack segments, and a 'root' program using *Chain* or *Execute* must therefore allocate segments of sufficient size to accommodate the largest segments in any *Chained* or *Executed* program.

Consequently, when compiling a 'root' program, you must set the value of the *Minimum Code Segment Size* to at least the same value as the largest code segment size of the programs to be chained/executed from that root. The required values are obtained from the status printout terminating any compilation. The values are in hexadecimal and specify number of paragraphs, a paragraph being 16 bytes.

Minimum Data Segment Size

The **D**-command is used to set the minimum size of the data segment for a .COM using *Chain* or *Execute.* As discussed above, a 'root' program using these commands must allocate segments of sufficient size to accommodate the largest data of any *Chained* or *Executed* program.

Consequently, when compiling a 'root' program, you must set the value of the *Minimum Data Segment Size* to at least the same value as the largest data segment size of the programs to be chained/executed from that root. The required values are obtained from the status printout terminating any compilation. The values are in hexadecimal and specify number of paragraphs, a paragraph being 16 bytes.

ClibPDF - www.fastio.com

Minimum Free Dynamic Memory

This value specifies the minimum memory size required for stack and heap. The value is in hexadecimal and specifies a number of paragraphs, a paragraph being 16 bytes.

Maximum Free Dynamic Memory

This value specifies the maximum memory size allocated for stack and heap. It must be used in programs which operate in a multi-user environment to assure that the program does not allocate the entire free memory. The value is in hexadecimal and specifies a number of paragraphs, a paragraph being 16 bytes.

Command Line Parameters

The **P**-command lets you enter one or more parameters which are passed to your program when running it in **M**emory mode, just as if they had been entered on the DOS command line. These parameters may be accessed through the *ParamCount* and *ParamStr* functions.

Find Run-time Error

When you run a program compiled in memory, and a run-time error occurs, the editor is invoked, and the error is automatically pointed out. This, of course, is not possible if the program is in a .COM file or an .CHN file. Run time errors then print out the error code and the value of the program counter at the time of the error:

```
Run-time error 01, PC=1B56
Program aborted
```

Figure 20-4: Run-time Error Message

To find the place in the source text where the error occurred, enter the **F** command. When prompted for the address, enter the address given by the error message:

```
   Enter PC: 1B56
```

Figure 20-5: Find Run-time Error

The place in the source text is now found and pointed out exactly as if the error had occurred while running the program in memory.

Notice that locating errors in programs using overlays can be a bit more tricky, as explained on page 196.

Standard Identifiers

The following standard identifiers are unique to the DOS implementations:

CSeg	*LongFilePos*	*MemW*	*PortW*
DSeg	*LongFileSize*	*MsDos*	*SSeg*
Intr	*LongSeek*	*Ofs*	*Seg*

Chain and Execute

TURBO Pascal provides two procedures *Chain* and *Execute* which allow TURBO programs to activate other TURBO programs. The syntax of the procedure calls are:

Chain(*FilVar*)
Execute(*FilVar*)

where *FilVar* is a file variable of any type, previously assigned to a disk file with the standard procedure *Assign*. If the file exists, it is loaded into memory and executed.

The *Chain* procedure is used only to activate special TURBO Pascal .CHN files, i.e. files compiled with the c**H**n-file option selected on the **O**ptions menu (see page 190). Such a file contains only program code; no Pascal library, it uses the Pascal library already present in memory.

The *Execute* procedure is used to activate any TURBO Pascal .COM file.

If the disk file does not exist, an I/O error occurs. This error is treated as described on page 116. When the **I** compiler directive is passive ({$I-}), program execution continues with the statement following the failed *Chain* or *Execute* statement, and the *IOresult* function must be called prior to further I/O.

ClibPDF - www.fastio.com

Data can be transferred from the current program to the chained program either by *shared global variables* or by *absolute address variables*.

To ensure overlapping, shared global variables should be declared as the very first variables in both programs, and they must be listed in the same order in both declarations. Furthermore, both programs must be compiled to the same size of code and data segments (see page 191). When these conditions are satisfied, the variables will be placed at the same address in memory by both programs, and as TURBO Pascal does not automatically initialize its variables, they may be shared.

Example:
Program *MAIN.COM:*

```
program Main;
var
  Txt:      string[80];
  CntPrg:   file;

begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```


Program *CHRCOUNT.CHN:*

```
program ChrCount;
var
  Txt:      string[80];
  NoOfChar,
  NoOfUpc,
  I:        Integer;

begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc := Succ(NoOfUpc);
  Write('No of characters in entry: ',NoOfChar);
  Writeln('. No of upper case characters: ', NoOfUpc,'.');
end.
```

If you want a TURBO program to determine whether it was invoked by eXecute or directly from the DOS command line, you should use an **absolute** variable at address *Cseg:$80.* This is the command line length byte, and when a program is called from DOS, it contains a value between 0 and 127. When eXecuting a program, therefore, the calling program should set this variable to something higher than 127. When you then check the variable in the called program, a value between 0 and 127 indicates that the program was called from DOS, a higher value that it was called from another TURBO program.

Chaining and eXecuting TURBO programs does not alter the memory allocation state. The base addresses and sizes of the code, data and stack segments are not changed; *Chain* and *Execute* only replace the program code in the code segment. 'Alien' programs, therefore, cannot be initiated from a TURBO program.

It is important that the first program which executes a *Chain* statement allocates enough memory for the code, data, and stack segments to accommodate largest .CHN program. This is done by using the **O**ptions menu to change the minimum code, data and free memory sizes (see page 190).

Note that neither *Chain* nor *Execute* can be used in direct mode, that is, from a program run with the compiler options switch in position **M**emory (page 190).

ClibPDF - www.fastio.com

Overlays

During execution, the system normally expects to find its overlay files on the logged drive and current directory. The *OvrPath* procedure may be used to change this default value.

OvrPath Procedure

Syntax: OvrPath(*Path*);

where *Path* is a string expression specifying a subdirectory path (see page 188 for an explanation of DOS directory paths). On subsequent calls to overlay files, the files will be expected in the specified directory. Once an overlay file has been opened in one directory, future calls to the same file will look in the same directory. The path may optionally specify a drive (*A:*, *B:*, etc.).

The current directory is identified by a single period. *OvrPath('.')* thus causes overlay files to be sought on the current directory.

Example :

```
program OvrTest;

overlay procedure ProcA;
begin
  Writeln('Overlay A');
end;

overlay procedure ProcB;
begin
  Writeln('Overlay B');
end;

procedure Dummy;
begin
  {Dummy procedure to separate the overlays
   into two groups}
end;

overlay procedure ProcC;
begin
  Writeln('Overlay C');
end;
```


ClibPDF - www.fastio.com


```
begin
  OvrPath('\subl');
  ProcA;
  OvrPath('.');
  ProcC;
  OvrPath('\subl');
  ProcB;
end.
```

The first call to *OvrPath* specifies overlays to be sought on the subdirectory *\sub1*. The call to *ProcA* therefore causes the first overlay file (containing the two overlay procedures *ProcA* and *ProcB* to be opened on this directory.

Next, the *OvrPath('.')* statement specifies that following overlays are to be found on the current directory. The call to *ProcC* opens the second overlay file here.

The following *ProcB* statement calls an overlay procedure in the first overlay file; and to ensure that it is sought on the *\sub1* directory, the *OvrPath('\sub1')* statement must be executed before the call.

Files

File Names

A file name in DOS consists of a path of directory names, separated by backslashes, leading up to the desired directory, followed by the actual file name:

Drive:*Dirname*\\...*Dirname**Filename*

If the path begins with a backslash, it starts in the root directory; otherwise, it starts in the logged drive.

The *Drive* and path specification is optional. If omitted, the file is assumed to reside on the logged drive.

The *FileName* consists of a name of one through eight letters or digits, optionally followed by a period and a file type of one through three letters or digits.

Number of Open Files

The number of files that may be open at the same time is controlled through the **F** compiler directive. The default setting is {$F16}, which means that up to 16 files may be open at any one time. If, for instance, a {$F24} directive is placed at the beginning of a program (**before** the declaration part), up to 24 files may be open concurrently. The F compiler directive does not limit the number of files that may be declared in a program; it only sets a limit to the number of files that may be open at the same time.

Note that even though the F compiler directive has been used to allocate sufficient file space, you may still experience a 'too many open files' error condition, if the operating system runs out of file buffers. If that happens, you should supply a higher value for the 'files = xx' parameter in the CONFIG.SYS file. The default value is usually 8. For further detail, please refer to your MS-DOS documentation.

Extended File Size

The following three additional file routines exist to accommodate the extended range of records in DOS. These are:

LongFileSize function,
LongFilePosition function, and
LongSeek procedure

They correspond to their *Integer* equivalents *FileSize, FilePosition,* and *Position* but operate with *Reals*. The functions thus return results of type *Real*, and the second parameter of the *LongSeek* procedure must be an expression of type *Real*.

File of Byte

In the CP/M implementations, access to non-TURBO files (except text files) must be done through untyped files because the two first bytes of typed TURBO files always contain the number of components in the file. This is not the case in the DOS versions, however, and a non-turbo file may therefore be declared as a **file of byte** and accessed randomly with *Seek, Read*, and *Write*.

Flush Procedure

The *Flush* procedure has no effect with typed files in DOS, as DOS typed file variables do not employ a sector buffer.

Truncate Procedure

Syntax: Truncate(*FilVar*);

This procedure truncates the file identified by *FilVar* at the current position of the file pointer, that is, records beyond the file pointer are cut away. *Truncate* also prepares the file for subsequent output.

ClibPDF - www.fastio.com

Text Files

Buffer Size

The text file buffer size is 128 bytes by default. This is adequate for most applications, but heavily I/O-bound programs, as for example a copy program, will benefit from a larger buffer, as it will reduce disk head movement.

You are therefore given the option to specify the buffer size when declaring a text file:

```
VAR  TextFile: Text[$800];
```

declares a text file variable with a buffer size of 2K bytes.

Append Procedure

Syntax: Append(*FilVar*);

The disk file assigned to the file variable *FilVar* is opened, and the file pointer is moved to the end of the file. The only operation allowed after *Append* is appending of new components.

Flush Procedure

The *Flush* procedure causes the file buffer to be flushed when used with text files.

Logical Devices

The following additional logical devices are provided:

INP: Refers to the MS-DOS standard input file (standard handle number 0).

OUT: Refers to the MS-DOS standard output file (standard handle number 1).

ERR: Refers to the MS-DOS standard error output file (standard handle number 2).

These devices may also be used with typed and untyped files.

The MS-DOS operating system itself also provides a number of logical devices, for instance 'CON', 'LST' and 'AUX'. TURBO Pascal will treat these devices as if they were disk files, with one exception: when a text file is opened, using *Reset*, *Rewrite* or *Append*, TURBO Pascal asks MS-DOS for the status of the file. If MS-DOS reports that the file is a device, TURBO Pascal disables the buffering that normally occurs on textfiles, and all I/O operations on the file are done on a character by character basis.

The D compiler option may be used to disable this check. The default state of the D option is {$D+}, and in this state, device checks are made. In the {$D-} state, no checks are made and all device I/O operations are buffered. In this case, a call to the flush standard procedure will ensure that the characters you have written to a file have actually been sent to it.

I/O redirection

PC/MS-DOS TURBO Pascal supports the I/O redirection feature provided by the MS-DOS operating system. In short, I/O redirection allows you to use disk files as the standard input source and/or standard output destination. Furthermore, a program supporting I/O redirection can be used as a *filter* in a *pipe*. Details on I/O redirection, filters, and pipes, are found in the MS-DOS documentation.

I/O redirection is enabled through the **G** (get) and **P** (put) compiler directives. The G directive controls the input file and the P directive controls the output file. The G and P directives both require an integer argument, which defines the size of the input or output buffer. The default buffer sizes are zero, and with these, *Input* and *Output* will refer to the *CON:* or the *TRM:* device.

If a non-zero input buffer is defined, for instance {$G256}, the standard *Input* file will refer to the MS-DOS standard input handle. Likewise, if a non-zero output buffer is defined, for instance {$P1024}, the standard *Output* file will refer to the MS-DOS standard output handle. The **D** compiler directive (see page 201) applies to such non-zero-buffer *Input* and *Output* files. The P and G compiler directives must be placed at the beginning of a program to have any effect, i.e. before the declaration part.

ClibPDF - www.fastio.com

The following program demonstrates re-directed I/O. It will read characters from the standard input file, keep a count of each alphabetical character (A through Z), and output a frequency distribution graph to the standard output file:

```
{$G512,P512,D-}
program CharacterFrequencyCounter;
const
  Bar     = #223;
var
  Count:    array[65..90] of Real;
  Ch:       Char;
  I,Graph:  Integer;
  Max,
  Total:    Real;
begin
  Max := 0; Total := 0;
  for I := 65 to 90 do Count[I] := 0;
  while not EOF do
  begin
    Read(Ch);
    if Ord(Ch) > 127 then Ch := Chr(Ord(Ch)-128);
    Ch := UpCase(Ch);
    if Ch in ['A'..'Z'] then
    begin
      Count[Ord(Ch)] := Count[Ord(Ch)] +1;
      if Count[Ord(Ch)] > Max then Max := Count[Ord(Ch)];
      Total := Total +1;
    end;
  end;
  Writeln('     Count    %');
  for I := 65 to 90 do
  begin
    Write(Chr(I),':   ',Count[I]:5:0,
          Count[I]*100/Total:5:0,'  ');
    for Graph := 1 to Round(Count[I]*63/Max) do
      Write(Bar);
    Writeln;
  end;
  Writeln('Total', Total:5:0);
end.
```


If the program is compiled into a file called COUNT.COM, then the MS-DOS command:

COUNT <TEXT.DOC >CHAR.CNT

will read the file TEXT.DOC and output the graph to the file CHAR.CNT.

Absolute Variables

Variables may be declared to reside at specific memory addresses, and are then called **absolute**. This is done by adding to the variable declaration the reserved word **absolute** followed by two *Integer* constants specifying a segment and an offset at which the variable is to be located:

```
var
  Abc: Integer absolute $0000:$00EE;
  Def: Integer absolute $0000:$00F0;
```

The first constant specifies the segment base address, and the second constant specifies the offset within that segment. The standard identifiers *CSeg* and *DSeg* may be used to place variables at absolute addresses within the code segment (CSeg) or the data segment (DSeg):

```
Special: array[1..CodeSize] absolute CSeg:$05F3;
```

Absolute may also be used to declare a variable "on top" of another variable, i.e. that a variable should start at the same address as another variable. When **absolute** is followed by the identifier of a variable or parameter, the new variable will start at the address of that variable parameter.

Example:

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and as the first byte of a string variable contains the length of the string, *StrLen* will contain the length of *Str*. Notice that an **absolute** variable declaration may only specify one identifier.

ClibPDF - www.fastio.com

Further details on space allocation for variables are found on page 216.

Absolute Address Functions

The following functions are provided for obtaining information about program variable addresses and system pointers.

Addr

Syntax: Addr(*Name*);

Returns the address in memory of the first byte of the variable with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is a 32 bit pointer consisting of a segment address and an offset.

Ofs

Syntax: Ofs(*Name*);

Returns the offset in the segment of memory occupied by the first byte of the variable, procedure or function with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is an *Integer*.

Seg

Syntax: Seg(*Name*);

Returns the address of the segment containing the first byte of the variable with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is an *Integer*.

Cseg

Syntax: Cseg;

Returns the base address of the **Code** segment. The value returned is an *Integer*.

Dseg

Syntax: Dseg;

Returns the base address of the **Data** segment. The value returned is an *Integer*.

Sseg

Syntax: Sseg;

Returns the base address of the **Stack** segment. The value returned is an *Integer*.

Predefined Arrays

TURBO Pascal offers four predefined arrays of type *Byte*, called *Mem*, *MemW*, *Port* and *PortW* which are used to access CPU memory and data ports.

Mem Array

The predefined arrays *Mem* and *MemW* are used to access memory. Each component of the array *Mem* is a byte, and each component of the array

Wmem is a word (two bytes, LSB first). The index must be an address specified as the segment base address and an offset separated by a colon and both of type *Integer*.

The following statement assigns the value of the byte located in segment 0000 at offset $0081 to the variable *Value*

```
Value:=Mem[0000:$0081];
```


ClibPDF - www.fastio.com

While the following statement:

```
MemW[Seg(Var):Ofs(Var)]:=Value;
```

places the value of the *Integer* variable *Value* in the memory location occupied by the two first bytes of the variable *Var*.

Port Array

The *Port* and *PortW* array are used to access the data ports of the 8086/88 CPU. Each element of the array represents a data port, with the index corresponding to port numbers. As data ports are selected by 16-bit addresses the index type is *Integer*. When a value is assigned to a component of *Port* or *PortW* it is output to the port specified. When a component of port is referenced in an expression, its value is input from the port specified. The components of the *Port* array are of type *Byte* and the components of *PortW* are of type *Integer*.

Example:
```
Port[56]:=10;
```

The use of the port array is restricted to assignment and reference in expressions only, i.e. components of *Port* and *PortW* cannot be used as variable parameters to procedures and functions. Furthermore, operations referring to the entire port array (reference without index) are not allowed.

With Statements

With statements may be nested to a maximum of 9 levels.

Pointer Related Items

MemAvail

The standard function *MemAvail* is available to determine the available space on the heap at any given time. The result is an *Integer* specifying the number of available *paragraphs* on the heap (a *paragraph* is 16 bytes).

Pointer Values

In very special circumstances it can be of interest to assign a specific value to a pointer variable *without using another pointer variable* or it can be of interest to obtain the actual value of a pointer variable.

Assigning a Value to a Pointer

The standard function *Ptr* can be used to assign specific values to a pointer variable. The function returns a 32 bit pointer consisting of a segment address and an offset.

Example:
```
Pointer:=Ptr(Cseg,$80);
```

Obtaining The Value of a Pointer

A pointer value is represented as a 32 bit entity and the standard function *Ord* can therefore **not** be used to obtain its value. Instead the functions *Ofs* and *Seg* must be used.

The following statement obtains the value of the pointer *P* (which is a segment address and an offset):

```
SegmentPart:=Seg(P^);
OffsetPart:=Ofs(P^);
```


ClibPDF - www.fastio.com

DOS Function Calls

For the purpose of making DOS system calls, TURBO Pascal introduces a procedure *MsDos*, which has a record as parameter:

```
MsDos(Record);
```

Details on DOS system calls and BIOS routines are found in the IBM DOS Technical Reference Manual.

The parameter to *MsDos* must be of the type:

```
record
  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
end;
```

or, alternatively:

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

Before TURBO makes the DOS system call, the registers AX, BX, CX, DX, BP, SI, DI, DS, and ES are loaded with the values specified in the record parameter. When DOS has finished operation the *MsDos* procedure will restore the registers to the record thus making any results from DOS available.

The following example shows how to use an MsDos function call to get the time from DOS:

```
procedure Timer(var Hour,Min,Sec,Frac:Integer);
type
  RegPack  =  record
                AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
              end;
var
  Regs:       Regpack;
```



```
begin
  with Regs do
  begin
    AX := $2C00;
    MsDos(Regs);
    Hour := hi(CX);
    Min  := lo(CX);
    Sec  := hi(DX);
    Frac := lo(DX);
  end;
  :
  :
  :
end;   { procedure Timer }
```

User Written I/O Drivers

For some applications it is practical for a programmer to define his own input and output drivers, i.e. routines which perform input and output of characters to and from an external device. The following drivers are part of the TURBO environment, and used by the standard I/O drivers (although they are not available as standard procedures or functions):

function *ConSt*: boolean; { 11 }
function *ConIn*: Char; { 8 }
procedure *ConOut* (Ch: Char); { 2 }
procedure *LstOut* (Ch: Char); { 5 }
procedure *AuxOut* (Ch: Char); { 4 }
function *AuxIn*: Char; { 3 }
procedure *UsrOut* (Ch: Char); { 2 }
function *UsrIn*: Char; { 8 }

The *ConSt* routine is called by the function *KeyPressed*, the *ConIn* and *ConOut* routines are used by the CON:, TRM:, and KBD: devices, the *LstOut* routine is used by the LST: device, the *AuxOut* and *AuxIn* routines are used by the AUX: device, and the *UsrOut* and *UsrIn* routines are used by the USR: device.

By default, these drivers are assigned to the DOS system calls as showed in curly brackets in the above listing of drivers.

This, however, may be changed by the programmer by assigning the

ClibPDF - www.fastio.com

address of a self-defined driver procedure or a driver function to one of the following standard variables:

Variable	Contains the address of the
ConStPtr	*ConSt* function
ConInPtr	*ConIn* function
ConOutPtr	*ConOut* procedure
LstOutPtr	*LstOut* procedure
AuxOutPtr	*AuxOut* procedure
AuxInPtr	*AuxIn* function
UsrOutPtr	*UsrOut* procedure
UsrInPtr	*UsrIn* function

A user defined driver procedure or driver function must match the definitions given above, i.e. a *ConSt* driver must be a boolean function, a *ConIn* driver must be a char function, etc.

External Subprograms

The reserved word **external** is used to declare external procedures and functions, typically procedures and functions written in machine code.

The reserved word **external** must be followed by a string constant specifying the name of a file in which executable machine code for the external procedure or function must reside. The default file type is *.COM*.

During compilation of a program containing external functions or procedures, the associated files are loaded and placed in the object code. As it is impossible to know in advance exactly *where* in the object code the external code will be placed this code **must** be relocatable, and no references must be made to the data segment. Furthermore the external code must save the registers BP, CS, DS and SS and restore these before executing the RET instruction.

An external subprogram has no *block*, i.e. no declaration part and no statement part. Only the subprogram heading is specified, immediately followed by the reserved word **external** and a filename specifying where to find the executable code for the subprogram.

Example:

```
procedure DiskReset; external 'DSKRESET';
function IOstatus: boolean; external 'IOSTAT';
```

An external file may contain code for more than one subprogram. The first subprogram is declared as described above, and the following are declared by specifying the identifier of the first subprogram followed by an an integer constant specifying an offset, enclosed in square brackets. The entry point of each subprogram is the address of the first subprogram plus the offset.

Example:

```
procedure Com1; external 'SERIAL.BIN';
function Com1Stat: Boolean; external Com1[3];
procedure Com1In: Char; external Com1[6];
procedure Com1Out: Char; external Com1[9];
```

The above example loads the file SERIAL.BIN into the program code, and defines four procedures called *Com1*, *Com1Stat*, *Com1In*, and *Com1Out* with entry points at the base address of the external code plus 0, 3, 6 and 9, respectively. When an external file contains several subprograms, the first part of the code is typically a jump table, as assumed in the example. In that way, the entry points of the subprograms remain unchanged if the external file is modified.

Parameters may be passed to external subprograms, and the syntax is exactly the same as that of calls to ordinary procedures and functions:

```
procedure Plot(X,Y: Integer); external 'PLOT';
procedure QuickSort(var List: PartNo); external 'QS';
```

External subprograms and parameter passing is discussed further on page 221.

In-line Machine Code

TURBO Pascal features the **inline** statements as a very convenient way of inserting machine code instructions directly into the program text. An inline statement consists of the reserved word **inline** followed by one or more *code elements* separated by slashes and enclosed in parentheses.

ClibPDF - www.fastio.com

A code element is built from one or more data elements, separated by plus (+) or minus (−) signs. A data element is either an integer constant, a variable identifier, a procedure identifier, a function identifier, or a location counter reference. A location counter reference is written as an asterisk (*).

Example:

```
inline (10/$2345/count+1/sort-*+2);
```

Each code element generates one byte or one word (two bytes) of code. The value of the byte or the word is calculated by adding or subtracting the values of the data elements according to the signs that separate them. The value of a variable identifier is the address (or offset) of the variable. The value of a procedure or function identifier is the address (or offset) of the procedure or function. The value of a location counter reference is the address (or offset) of the location counter, i.e. the address at which to generate the next byte of code.

A code element will generate one byte of code if it consists of integer constants only, and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range, or if the code element refers to variable, procedure, or function identifiers, or if the code element contains a location counter reference, one word of code is generated (least significant byte first).

The '<' and '>' characters may be used to override the automatic size selection described above. If a code element starts with a '<' character, only the least significant byte of the value is coded, even if it is a 16-bit value. If a code element starts with a '>' character, a word is always coded, even though the most significant byte is zero.

Example:

```
inline (<$1234/>$44);
```

This **inline** statement generates three bytes of code: $34, $44, $00.

The value of a variable identifier use in a **inline** statement is the offset address of the variable within its base segment. The base segment of global variables (i.e. variables declared in the main program block) is the data segment, which is accessible through the DS register. The base segment of local variables (i.e. variables declared within the current subprogram) is the stack segment, and in this case the variable offset is relative to the BP (base page) register, the

use of which automatically causes the stack segment to be selected. The base segment of typed constants is the code segment, which is accessible through the CS register. **inline** statements should not attempt to access variables that are not declared in the main program nor in the current subprogram.

The following example of an inline statement generates machine code that will convert all characters in its string argument to upper case.

```
procedure UpperCase(var Strg: Str);
{Str is type String[255]}
begin
  inline
    ($C4/$BE/Strg/        {      LES  DI,Strg[BP]             }
     $26/$8A/$0D/         {      MOV  CL,ES:[DI]              }
     $FE/$C1/             {      INC  CL                      }
     $FE/$C9/             { L1:  DEC  CL                      }
     $74/$13/             {      JZ   L2                      }
     $47/                 {      INC  DI                      }
     $26/$80/$3D/$61/     {      CMP  ES:BYTE PTR [DI],'a'}
     $72/$F5/             {      JB   L1                      }
     $26/$80/$3D/$7A/     {      CMP  ES:BYTE PTR [DI],'z'}
     $77/$EF/             {      JA   L1                      }
     $26/$80/$2D/$20/     {      SUB  ES:BYTE PTR [DI],20H}
     $EB/$E9);            {      JMP  SHORT L1                }
                          { L2:                               }
end;
```

Inline statements may be freely mixed with other statements throughout the statement part of a block, and **inline** statements may use all CPU registers. **Note**, however, that the contents of the registers BP, SP, DS, and SS must be the same on exit as on entry.

ClibPDF - www.fastio.com

Interrupt Handling

A TURBO Pascal interrupt routine must manually preserve registers AX, BX, CX, DX, SI, DI, DS and ES. This is done by placing the following in-line statement as the first statement of the procedure:

```
inline ($50/$53/$51/$52/$56/$57/$1E/$06/$FB);
```

The last byte ($FB) is an STI instruction which enables further interrupts - it may or may not be required. The following inline statement must be the last statement in the procedure:

```
inline ($07/$1F/$5F/$5E/$5A/$59/$5B/$58/$8B/$E5/$5D/$CF);
```

This restores the registers and reloads the stack pointer (SP) and the base page register (BP). The last byte ($CF) is an IRET instruction which overrides the RET instruction generated by the compiler.

An interrupt service procedure must not employ any I/O operations using the standard procedures and functions of TURBO Pascal, as the BDOS is not re-entrant. The programmer must initialize the interrupt vector used to activate the interrupt service routine.

Intr procedure

Syntax: Intr(*InterruptNo, Result*)

This procedure initializes the registers and flags as specified in the parameter *Result* which must be of type:

```
Result = record
           AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
         end;
```

It then makes the software interrupt given by the parameter *interruptNo* which must be an *Integer* constant. When the interrupt service routine returns control to your program, *Result* will contain any values returned from the service routine.

Note that the data segment register DS, used to access global variables, will not have the correct value when the interrupt service routine is entered. Therefore, global variables cannot be directly accessed. *Typed constants*, however, are available, as they are stored in the code segment. The way to access global variables in the interrupt service routine is therefore to store the value of *Dseg* in a typed constant in the main program. This typed constant can then be accessed by the interrupt handler and used to set its DS register.

ClibPDF - www.fastio.com

Internal Data Formats

In the following descriptions, the symbol @ denotes the offset of the first byte occupied by a variable of the given type within its segment. The segment base address can be determined by using the standard function *Seg*.

Global and *local variables*, and *typed constants* occupy different segments as follows:

Global variables reside in the data segment and the offset is relative to the DS register.

Local variables reside in the stack segment and the offset is relative to the BP register.

Typed constants reside in the code segment and the offset is relative to the CS register.

All variables are contained within their base segment.

Basic Data Types

The basic data types may be grouped into structures (arrays, records, and disk files), but this structuring will not affect their internal formats.

Scalars

The following scalars are all stored in a single byte: *Integer* subranges with both bounds in the range 0..255, booleans, chars, and declared scalars with less than 256 possible values. This byte contains the ordinal value of the variable.

The following scalars are all stored in two bytes: *Integers*, *Integer* subranges with one or both bounds not within the range 0..255, and declared scalars with more than 256 possible values. These bytes contain a 2's complement 16-bit value with the least significant byte stored first.

Reals

Reals occupy 6 bytes, giving a floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next five bytes with the least significant byte first:

@	Exponent
@ +1	LSB of mantissa
:	
@ +5	MSB of mantissa

The exponent uses binary format with an offset of $80. Hence, an exponent of $84 indicates that the value of the mantissa is to be multiplied by 2 ^($84-$80) = 2 ^4 = 16. If the exponent is zero, the floating point value is considered to be zero.

The value of the mantissa is obtained by dividing the 40-bit unsigned integer by 2^40. The mantissa is always normalized, i.e. the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. The sign of the mantissa is stored in this bit, however, a 1 indicating that the number is negative, and a 0 indicating that the number is positive.

Strings

A string occupies as many bytes as its maximum length plus one. The first byte contains the current length of the string. The following bytes contains the string with the first character stored at the lowest address. In the table shown below, **L** denotes the current length of the string, and *Max* denotes the maximum length:

@	Current length (*L*)
@ +1	First character
@ +2	Second character
:	
@ +*L*	Last character
@ +*L*+1	Unused
:	
@ + *Max*	Unused

ClibPDF - www.fastio.com

Sets

An element in a *Set* occupies one bit, and as the maximum number of elements in a set is 256, a set variable will never occupy more than 32 bytes (256/8).

If a set contains less than 256 elements, some of the bits are bound to be zero at all times and need therefore not be stored. In terms of memory efficiency, the best way to store a set variable of a given type would then be to "cut off" all insignificant bits, and rotate the remaining bits so that the first element of the set would occupy the first bit of the first byte. Such rotate operations, however, are quite slow, and TURBO therefore employs a compromise: Only bytes which are statically zero (i.e. bytes of which no bits are used) are not stored. This method of compression is very fast and in most cases as memory efficient as the rotation method.

The number of bytes occupied by a set variable is calculated as (*Max* **div** 8) − (*Min* **div** 8) + 1, where *Max* and *Min* are the upper and lower bounds of the base type of that set. The memory address of a specific element *E* is:

MemAddress = @ + (*E* **div** 8) − (*Min* **div** 8)

and the bit address within the byte at *MemAddress* is:

BitAddress = *E* **mod** 8

where *E* denotes the ordinal value of the element.

Pointers

A pointer consists of four bytes containing a segment base address and an offset. The two least significant bytes contains the offset and the two most significant bytes the base address. Both are stored in memory using byte reversed format, i.e. the least significant byte is stored first. The value **nil** corresponds to two zero words.

Data Structures

Data structures are built from the basic data types using various structuring methods. Three different structuring methods exist: Arrays, records, and disk files. The structuring of data does not in any way affect the internal formats of the basic data types.

Arrays

The components with the lowest index values are stored at the lowest memory address. A multi-dimensional array is stored with the rightmost dimension increasing first, e.g. given the array

```
Board: array[1..8,1..8] of Square
```

you have the following memory layout of its components:

```
lowest address:  Board[1,1]
                 Board[1,2]
                 :
                 Board[1,8]
                 Board[2,1]
                 Board[2,2]
                 :
                 :
Highest address: Board[8,8]
```

Records

The first field of a record is stored at the lowest memory address. If the record contains no variant parts, the length is given by the sum of the lengths of the individual fields. If a record contains a variant, the total number of bytes occupied by the record is given by the length of the fixed part plus the length of largest of its variant parts. Each variant starts at the same memory address.

ClibPDF - www.fastio.com

Disk Files

Disk files are different from other data structures in that data is not stored in internal memory but in a file on an external device. A disk file is controlled through a file interface block (FIB).

File Interface Blocks

The following table shows the format of a FIB:

@+0	File handle (LSB).
@+1	File handle (MSB).
@+2	Record length (LSB) or flags byte.
@+3	Record length (MSB) or character buffer.
@+4	Buffer offset (LSB).
@+5	Buffer offset (MSB).
@+6	Buffer size (LSB).
@+7	Buffer size (MSB).
@+8	Buffer pointer (LSB).
@+9	Buffer pointer (MSB).
@+10	Buffer end (LSB).
@+11	Buffer end (MSB).
@+12	First byte of file path.
:	
@+75	Last byte of file path.

The word at @+0 and @+1 contains the 16-bit file handle returned by MS-DOS when the file was opened (or 0FFFFH when the file is closed). For typed and untyped files, the word at @+2 and @+3 contains the record length in bytes (zero if the file is closed), and bytes @+4 to @+11 are unused.

For text files, the format of the flags byte at @+2 is:

Bit 0..3	File type.
Bit 5	Pre-read character flag.
Bit 6	Output flag.
Bit 7	Input flag.

File type 0 denotes a disk file, and 1 through 5 denote the TURBO Pascal logical I/O devices (CON:, KBD:, LST:, AUX:, and USR:). Bit 5 is set if the character buffer contains a pre-read character, bit 6 is set if output is allowed, and bit 7 is set if input is allowed.

The four words from @+4 to @+11 store the offset address of the buffer, its size, the offset of the next character to read or write, and the offset of the first byte after the buffer. The buffer always resides in the same segment as the FIB, usually starting at @+76. When a textfile is assigned to a logical device, only the flags byte and the character buffer are used.

The file path is an ASCII string (a string terminated by a zero byte) of up to 63 characters.

Random Access Files

A random access file consists of a sequence of records, all of the same length and same internal format. To optimize file storage capacity, the records of a file are totally contiguous.

TURBO saves no information about the record length. The programmer must therefore see to it that a random access file is accessed with the correct record length.

The size returned by the standard function *Filesize* is obtained from the DOS directory.

Text Files

The basic components of a text file are characters, but a text file is furthermore divided into *lines*. Each line consists of any number of characters ended by a CR/LF sequence (ASCII $0D/ $0A). The file is terminated by a Ctrl-Z (ASCII $1B).

Parameters

Parameters are transferred to procedures and functions via the stack which is addressed through SS:SP.

On entry to an **external** subroutine, the top of the stack always contains the return address within the code segment (a word). The parameters, if any, are located below the return address, i.e. at higher addresses on the stack.

ClibPDF - www.fastio.com

If an external function has the following subprogram header:

```
function Magic(var R: Real; S: string5): Integer;
```

then the stack upon entry to *Magic* would have the following contents:

```
< Function result              >
< Segment base address of R    >
< Offset address of R          >
< First character of S         >
:
< Last character of S          >
< Length of S                  >
< Return address               >  SP
```

An external subroutine should save the Base Page register (BP) and then copy the Stack Pointer SP into the Base Page register in order to be able to refer to parameters. Furthermore the subroutine should reserve space on the stack for local workarea. This can be obtained by the following instructions:

```
PUSH  BP
MOV   BP,SP
SUB   SP,WORKAREA
```

The last instruction will have the effect of adding the following to the stack:

```
< Return address               >  BP
< The saved BP register        >
< First byte of local workarea >
:
< Last byte of local work area >  SP
```

Parameters are accessed via the BP register.

The following instruction will load length of the string into the AL register:

```
MOV  AL,[BP-1]
```


Before executing a RET instruction the subprogram must reset the Stack Pointer and Base Page register to their original values. When executing the RET the parameters may be removed by giving RET a parameter specifying how many bytes to remove. The following instructions should therefore be used when exiting from a subprogram:

```
MOV  SP,BP
POP  BP
RET  NoOfBytesToRemove
```

Variable Parameters

With a variable (**var**) parameter, two words are transferred on the stack giving the base address and offset of the first byte occupied by the actual parameter.

Value Parameters

With value parameters, the data transferred on the stack depends upon the type of the parameter as described in the following sections.

Scalars

Integers, Booleans, Chars and declared scalars (i.e. all scalars except *Reals*) are transferred on the stack as a word. If the variable occupies only one byte when it is stored, the most significant byte of the parameter is zero.

Reals

A real is transferred on the stack using six bytes.

Strings

When a string is at the top of the stack, the topmost byte contains the length of the string followed by the characters of the string.

ClibPDF - www.fastio.com

Sets

A set always occupies 32 bytes on the stack (set compression only applies to the loading and storing of sets).

Pointers

A pointer value is transferred on the stack as two words containing the base address and offset of a dynamic variable. The value NIL corresponds to two zero words.

Arrays and Records

Even when used as value parameters, *Array* and *Record* parameters are not actually transferred on the stack. Instead, two words containing the base address and offset of the first byte of the parameter are transferred. It is then the responsibility of the subroutine to use this information to make a local copy of the variable.

Function Results

User written **external** functions must remove all parameters and the function result from the stack when they return.

User written **external** functions must return their results exactly as specified in the following:

Values of scalar types, except *Reals*, must be returned in the AX register. If the result is only one byte then AH should be set to zero. Boolean functions must return the function value by setting the Z flag (Z = *False*, NZ = *True*).

Reals must be returned on the stack with the exponent at the lowest address. This is done by not removing the function result variable when returning.

Sets must be returned on the top of the stack according to the format described on page 223. On exit SP must point at the byte containing the string length.

Pointer values must be returned in DX:AX.

Heap and The Stacks

During execution of TURBO Pascal program the following segments are allocated for the program:

a Code Segment,
a Data Segment, and
a Stack Segment

Two stack-like structures are maintained during execution of a program: the *heap* and the *stack*.

The heap is used to store dynamic variables, and is controlled with the standard procedures *New, Mark,* and *Release*. At the beginning of a program, the heap pointer *HeapPtr* is set to low memory in the stack segment and the heap grows upwards towards the stack. The predefined variable *HeapPtr* contains the value of the heap pointer and allows the programmer to control the position of the heap.

The stack is used to store local variables, intermediate results during evaluation of expressions and to transfer parameters to procedures and functions. At the beginning of a program, the stack pointer is set to the address of the top of the stack segment.

On each call to the procedure *New* and on entering a procedure or function, the system checks for collision between the heap and the recursion stack. If a collision has occurred, an execution error results, unless the **K** compiler directive is passive ({ $K-}).

ClibPDF - www.fastio.com

Memory Management

When a TURBO program is executed, three segments are allocated for the program: A code segment, a data segment, and a stack segment.

Code segment (CS is the code segment register):

```
CS:0000 - CS:00FF    MS-DOS base page.
CS:0100 - CS:EOFR    Run-time library code.
CS:EOFR - CS:EOFP    Program code.
CS:EOFP - CS:EOFC    Unused.
```

Data segment (DS is the data segment register):

```
DS:0000 - DS:EOFW    Run-time library workspace.
DS:EOFW - DS:EOFM    Main program block variables.
DS:EOFM - DS:EOFD    Unused.
```

The unused areas between (CS:EOFP-CS:EOFC and DS:EOFM-DS:EOFD) are allocated only if a minimum c**O**de segment size larger than the required size is specified at compilation. The sizes of the code and data segments never exceed 64K bytes each.

The stack segment is slightly more complicated, as it may be larger than 64K bytes. On entry to the program the stack segment register (SS) and the stack pointer (SP) is loaded so that SS:SP points at the very last byte available in the entire segment. During execution of the program SS is never changed but SP may move downwards until it reaches the bottom of the segment, or 0 (corresponding to 64K bytes of stack) if the stack segment is larger than 64K bytes.

The heap grows from low memory in the stack segment towards the actual stack residing in high memory. Each time a variable is allocated on the heap, the heap pointer (which is a double word variable maintained by the TURBO run-time system) is moved upwards, and then normalized, so that the offset address is always between $0000 and $000F. Therefore, the maximum size of a single variable that can be allocated on the heap is 65521 bytes (corresponding to $10000 less $000F). The total size of all variables allocated on the heap is however only limited by the amount of memory available. The heap pointer is available to the programmer through the *HeapPtr* standard identifier. *HeapPtr* is a typeless pointer which is compatible with all pointer types. Assignments to *HeapPtr* should be exercised only with extreme care.

...apter 21
...M-86

This chapter describes features of TURBO Pascal specific to the CP/M-86 implementation. It presents two kinds of information:

Things you should know to make efficient use of TURBO Pascal. Pages 227 through 240.

...) The rest of the chapter describes things which are of interest only to experienced programmers, such as machine language routines, technical aspects of the compiler, etc.

Compiler Options

The **O** command selects the following menu from which you may view and change some default values of the compiler. It also provides a helpful function to find runtime errors in programs compiled into object code files.

```
compile -> Memory
           Cmd-file
           cHn-file

command line Parameter:

Find run-time error  Quit
```

Figure 21-1: Options Menu

Memory / Cmd file / cHn-file

The three commands **M**, **C**, and **H** select the compiler mode, i.e. where to put the code which results from the compilation. **M**emory is the default mode. When active, code is produced in memory and resides there ready to be activated by a **R**un command.

ClibPDF - www.fastio.com