DOS Reference

VERSION 4.5

Borland® C++

Borland C++ DOS Reference Borland

DOS Reference

VERSION 4.5

Borland® C++

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R1094
9495969798-9 8 7 6 5 4 3 2
H1

Contents

Introduction 1

What's in this book . . . 1

Chapter 1
DOS memory management 3

Running out of memory . . . 3
Memory models . . . 3
The 8086 registers . . . 4
General-purpose registers . . . 4
Segment registers . . . 5
Special-purpose registers . . . 5
The flags register . . . 5
Memory segmentation . . . 6
Address calculation . . . 7
Pointers . . . 7
Near pointers . . . 8
Far pointers . . . 8
Huge pointers . . . 9
The six memory models . . . 9
Mixed-model programming: Addressing modifiers . . . 14
Segment pointers . . . 14
Declaring far objects . . . 15
Declaring functions to be near or far . . . 16
Declaring pointers to be near, far, or huge . . . 16
Pointing to a given segment:offset address . 17
Using library files . . . 18
Linking mixed modules . . . 18
Overlays (VROOMM) for DOS . . . 19
How overlays work . . . 19
Guidelines for using Borland C++ overlays effectively . . . 21
Requirements . . . 21
Exception handling and overlays . . . 21
Using overlays . . . 22
Overlay example . . . 22
Overlaid programs . . . 23
The far call requirement . . . 23
Buffer size . . . 23
What not to overlay . . . 23
Debugging overlays . . . 23
External routines in overlays . . . 24
Swapping . . . 24

Chapter 2
Math 27

Floating-point I/O . . . 27
Floating-point options . . . 27
Emulating the 80x87 chip . . . 28
Using the 80x87 code . . . 28
No floating-point code . . . 28
Fast floating-point option . . . 28
The 87 environment variable . . . 29
Registers and the 80x87 . . . 29
Disabling floating-point exceptions . . . 30
Using complex types . . . 30
Using bcd types . . . 31
Converting bcd numbers . . . 32
Number of decimal digits . . . 32

Chapter 3
Video functions 35

Video modes . . . 35
Windows and viewports . . . 36
Programming in graphics mode . . . 36
The graphics library functions . . . 37
Graphics system control . . . 37
A more detailed discussion . . . 38
Drawing and filling . . . 39
Manipulating the screen and viewport . . . 40
Text output in graphics mode . . . 41
Color control . . . 43
Pixels and palettes . . . 43
Background and drawing color . . . 44
Color control on a CGA . . . 44
CGA low resolution . . . 44
CGA high resolution . . . 45
CGA palette routines . . . 45
Color control on the EGA and VGA . . . 45
Error handling in graphics mode . . . 46
State query . . . 47

Chapter 4
Borland graphics interface 49

arc . . . 49
bar . . . 50
bar3d . . . 50
circle . . . 51
cleardevice . . . 51
clearviewport . . . 52
closegraph . . . 52
detectgraph . . . 53
drawpoly . . . 55
ellipse . . . 55
fillellipse . . . 56
fillpoly . . . 56
floodfill . . . 57

getarccoords . . . 57
getaspectratio . . . 58
getbkcolor . . . 58
getcolor . . . 59
getdefaultpalette . . . 59
getdrivername . . . 60
getfillpattern . . . 60
getfillsettings . . . 61
getgraphmode . . . 62
getimage . . . 62
getlinesettings . . . 63
getmaxcolor . . . 64
getmaxmode . . . 65
getmaxx . . . 65
getmaxy . . . 66
getmodename . . . 66
getmoderange . . . 67
getpalette . . . 67
getpalettesize . . . 68
getpixel . . . 68
gettextsettings . . . 69
getviewsettings . . . 69
getx . . . 70
gety . . . 70
graphdefaults . . . 71
grapherrormsg . . . 71
_graphfreemem . . . 72
_graphgetmem . . . 72
graphresult . . . 73
imagesize . . . 74
initgraph . . . 74
installuserdriver . . . 77
installuserfont . . . 78
line . . . 79
linerel . . . 79
lineto . . . 79
moverel . . . 80
moveto . . . 80
outtext . . . 81
outtextxy . . . 81
pieslice . . . 82
putimage . . . 83
putpixel . . . 83
rectangle . . . 84
registerbgifont . . . 84
registerbgidriver . . . 85
restorecrtmode . . . 86
sector . . . 86
setactivepage . . . 87
setallpalette . . . 87
setaspectratio . . . 89
setbkcolor . . . 89
setcolor . . . 90
setfillpattern . . . 91
setfillstyle . . . 92
setgraphmode . . . 93
setgraphbufsize . . . 93
setlinestyle . . . 94
setpalette . . . 95
setrgbpalette . . . 96
settextjustify . . . 97
settextstyle . . . 98
setusercharsize . . . 99
setviewport . . . 100
setvisualpage . . . 101
setwritemode . . . 101
textheight . . . 102
textwidth . . . 102

Chapter 5
DOS-only functions **105**
absread . . . 105
abswrite . . . 106
allocmem, _dos_allocmem . . . 106
bioscom . . . 107
biosdisk . . . 109
_bios_disk . . . 111
bioskey . . . 112
_bios_keybrd . . . 113
biosprint . . . 114
_bios_printer . . . 115
_bios_serialcom . . . 116
brk . . . 118
coreleft . . . 118
delay . . . 119
farcoreleft . . . 119
farheapcheck . . . 120
farheapcheckfree . . . 120
farheapchecknode . . . 121
farheapfillfree . . . 121
farheapwalk . . . 122
freemem, _dos_freemem . . . 122
harderr, hardresume, hardretn . . . 123
_harderr . . . 124

_hardresume. 125
_hardretn. 125
keep, _dos_keep. 126
nosound 127
_OvrInitEms. 127
_OvrInitExt 128
randbrd. 128
randbwr 129
sbrk 130
setblock, _dos_setblock 130
sound 131

Appendix A
DOS libraries 133

The run-time libraries 133
The DOS support libraries. 134
Graphics routines. 134
Interface routines 135
Memory routines 135
Miscellaneous routines. 136

Appendix B
DOS global variables 137

_heaplen 137
_ovrbuffer 138
_stklen 139

Index 141

Tables

1.1 Memory models 13
1.2 Pointer results 14
3.1 Graphics mode state query functions 47
4.1 detectgraph constants 53
4.2 Graphics drivers information 54
4.3 Graphics drivers constants 75
4.4 Color palettes 76
4.5 Graphics modes 76
4.6 Actual color table 88
A.1 Summary of DOS run-time libraries 134

Figures

1.1 8086 registers 4
1.2 Flags register of the 80x86 processors 6
1.3 Tiny model memory segmentation 10
1.4 Small model memory segmentation 11
1.5 Medium model memory segmentation 11
1.6 Compact model memory segmentation 12
1.7 Large model memory segmentation 12
1.8 Huge model memory segmentation 13
1.9 Memory maps for overlays 20

Introduction

This manual provides information you might need to develop 16-bit applications that are targeted to run DOS. The following manuals in this documentation set also discuss DOS-related issues:

- The *User's Guide* provides a description of all the programming options that can be used to develop applications on any platform supported by Borland C++ 4.5.
- The *Programmer's Guide* describes the implementation and extensions to the C and C++ programming languages. Much of the information in the *Programmer's Guide* (for example, information regarding exception-handling, RTTI, and other recent additions to the C++ language) is applicable to 16-bit DOS programming.
- The *Library Reference* provides a reference to functions and macros, many of which are marked as being available to DOS programs.
- The *Class Libraries Guide* provides a discussion and reference to classes and macros that are available only for C++ programs.

Typefaces and icons used in these books are described in the *User's Guide*.

What's in this book

Chapter 1, "DOS memory management" describes memory models, overlays, and mixed-model programming. Remember that in DOS-only applications you can use any of the six memory models (the tiny and huge memory models aren't supported in Windows applications). Overlays are supported only in DOS applications.

Chapter 2, "Math" covers floating-point issues and how to use the *bcd* and *complex* math classes. Much of the information regarding math options is specific to DOS applications. The discussion of *bcd* and *complex* isn't specific to DOS and is available to applications on Windows and OS/2 platforms.

Chapter 3, "Video functions" discusses graphics in Borland C++. The topics discussed in this chapter are available only for 16-bit DOS applications.

Chapter 4, "Borland graphics interface" is a reference to the functions declared in the graphics.h header file. The functions discussed in this chapter are available only for

16-bit DOS applications. Sample programs for these functions are available in the online Help.

Chapter 5, "DOS-only functions" is a reference to those functions that are available only in a 16-bit DOS-targeted application. There are many additional functions and C++ classes that can be used in DOS applications (and are also available to other platforms). Those additional functions are documented in the *Library Reference*. The online Help provides many sample programs for the functions that are referenced here and in the *Library Reference*.

Appendix A, "DOS libraries" provides an overview of the libraries and global variables that are available only for 16-bit DOS applications.

Appendix B, "DOS global variables" describes the global variables that are available only for 16-bit DOS applications.

Chapter 1

DOS memory management

This chapter discusses

- What to do when you receive "Out of memory" errors.
- What memory models are: how to choose one, and why you would (or wouldn't) want to use a particular memory model.
- How overlays work, and how to use them.
- How to overlay modules with exception-handling constructs.

Running out of memory

Borland C++ does not generate any intermediate data structures to disk when it is compiling (Borland C++ writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message "Out of memory" if there isn't enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions.

Memory models

Borland C++ gives you six memory models, each suited for different program and code sizes. Each memory model uses memory differently. What do you need to know to use memory models? To answer that question, you need to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; an 80286, 80386, 80486, or Pentium. For now, we'll just refer to it as an 8086.

Note See page 9 for a summary of each memory model.

The 8086 registers

The following figure shows some of the registers found in the 8086 processor. There are other registers—because they can't be accessed directly, they aren't shown here.

Figure 1.1 8086 registers

General-purpose registers

AX	accumulator (math operations)	
	AH	AL
BX	base (indexing)	
	BH	BL
CX	count (indexing)	
	CH	CL
DX	data (holding data)	
	DH	DL

Segment address registers

CS	code segment pointer
DS	data segment pointer
SS	stack segment pointer
ES	extra segment pointer

Special-purpose registers

SP	stack pointer
BP	base pointer
SI	source index
DI	destination index

General-purpose registers

The general-purpose registers are the registers used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.
- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers

The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers

The 8086 also has some special-purpose registers:

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Borland C++ for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

Borland C++ functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP points to the saved previous BP value if there is a stack frame. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off*.

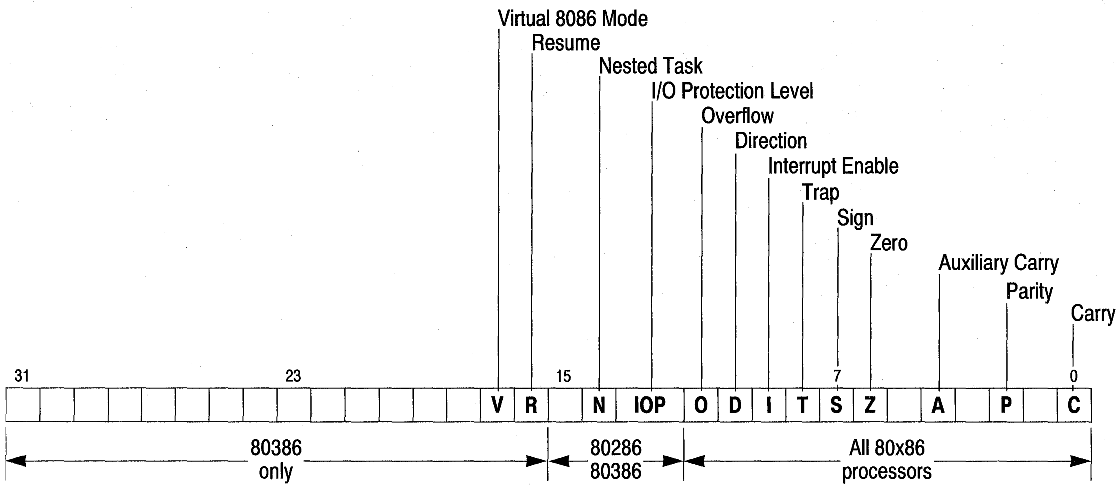
Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to local variables.

The flags register

The 16-bit flags register contains all pertinent information about the state of the 8086 and the results of recent instructions.

For example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the *carry* and *overflow flags*, similarly report the results of arithmetic and logical operations.

Figure 15.2 Flags register of the 80x86 processors



Other flags control the 8086 operation modes. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD**, **STI**, and **CMC**) and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ**, **RCR**, and **MOVSB**. The flags register is not really used as a storage location, but rather holds the status and control data for the 8086.

Memory segmentation

The Intel 8086 microprocessor has a *segmented memory architecture*. It has a total address space of 1 MB, but is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase "segmented memory architecture."

- The 8086 keeps track of four different segments: *code*, *data*, *stack*, and *extra*. The code segment is where the machine instructions are; the data segment is where information is; the stack is, of course, the stack; and the extra segment is also used for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.
- A segment can be located anywhere in memory. In DOS real mode it can be located almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that's evenly divisible by 16 (in decimal).

Address calculation

Note This whole section is applicable only to real mode under DOS. You can safely ignore it for Windows development.

A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment: how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

```
DS register (shifted):  0010 1111 1000 0100 0000  =  2F840
Offset:                      0000 0101 0011 0010  =  00532
______________________  __________________________________

Address:                0010 1111 1101 0111 0010  =  2FD72
```

Note A chunk of 16 bytes is known as a *paragraph*, so you could say that a segment always starts on a paragraph boundary.

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

Segments can overlap (but don't have to). For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that's all the space you'd have for your code, your data, and your stack.

Pointers

Although you can declare a pointer or function to be a specific type regardless of the model used, by default the type of memory model you choose determines the default type of pointers used for code and data. There are four types of pointers: *near* (16 bits), *far* (32 bits), *huge* (also 32 bits), and *segment* (16 bits).

Near pointers

A near pointer (16-bits) relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far pointers

A far pointer (32-bits) contains not only the offset within the segment, but also the segment address (as another 16-bit value), which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; this, in turn, allows you to have programs larger than 64K. You can also address more than 64K of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) . . .
if (b == c) . . .
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the **>**, **>=**, **<**, and **<=** operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) . . .
if (b > c) . . .
if (a > c) . . .
```

The equals (**==**) and not-equal (**!=**) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (**<=**, **>=**, **<**, and **>**) use just the offset.

The **==** and **!=** operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

Note If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge pointers

Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, they are *normalized* to avoid the problems associated with far pointers.

A normalized pointer is a 32-bit pointer that has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

There are three reasons why it is important to always keep huge pointers normalized:

1. For any given memory address there is only one possible huge address (segment:offset) pair. That means that the == and != operators return correct answers for any huge pointers.
2. In addition, the >, >=, <, and <= operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results of these comparisons will also be correct.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **struct**s that's larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

The six memory models

Borland C++ gives you six memory models for 16-bit DOS programs: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

- **Tiny.** As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the **/t** option. Use this model when memory is at an absolute premium.

- **Small.** The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used. This is a good size for average applications.
- **Medium.** Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MB. This model is best for large programs without much data in memory.
- **Compact.** The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1 MB range. This model is best if code is small but needs to address a lot of data.
- **Large.** Far pointers are used for both code and data, giving both a 1 MB range. Large and huge are needed only for very large applications.
- **Huge**. Far pointers are used for both code and data. Borland C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

Figures 1.3 through 1.8 show how memory in the 8086 is apportioned for the Borland C++ memory models. To select these memory models, you can either use menu selections from the IDE or you can type options invoking the command-line compiler version of Borland C++.

Figure 1.3 Tiny model memory segmentation

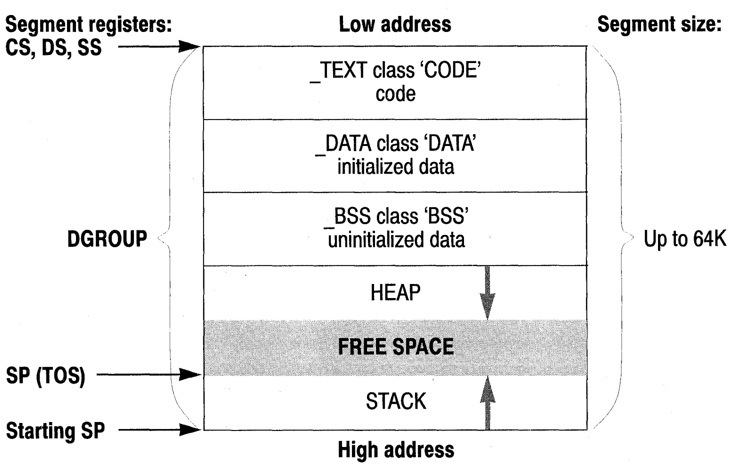


Figure 1.4 Small model memory segmentation

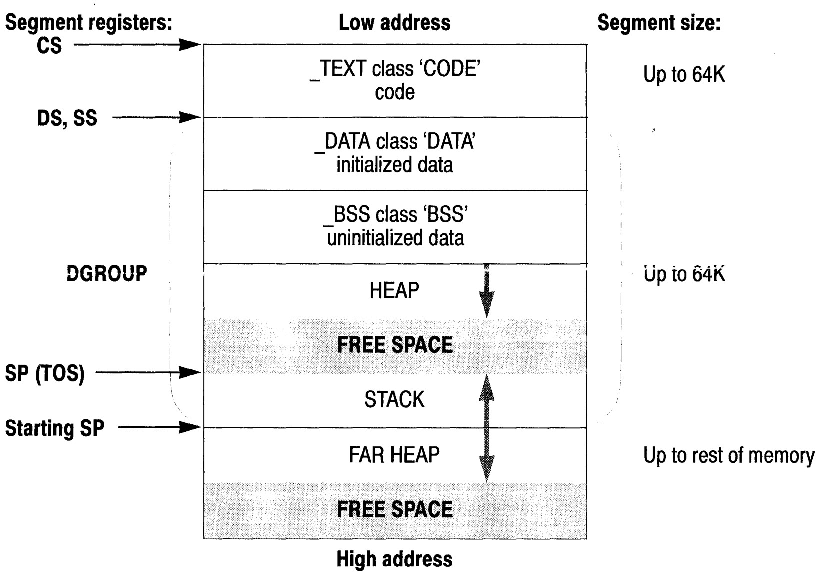


Figure 1.5 Medium model memory segmentation

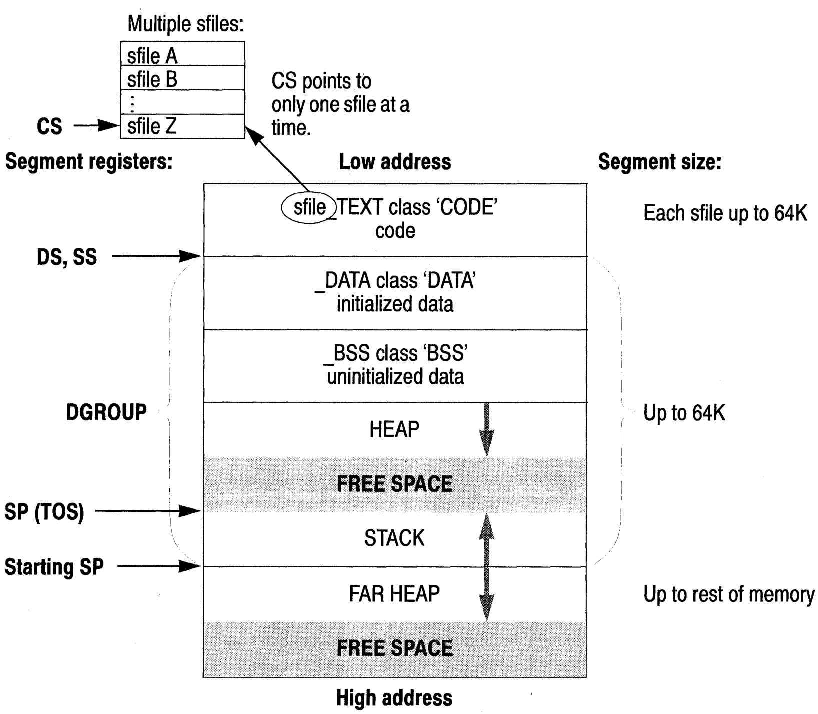


Figure 1.6 Compact model memory segmentation

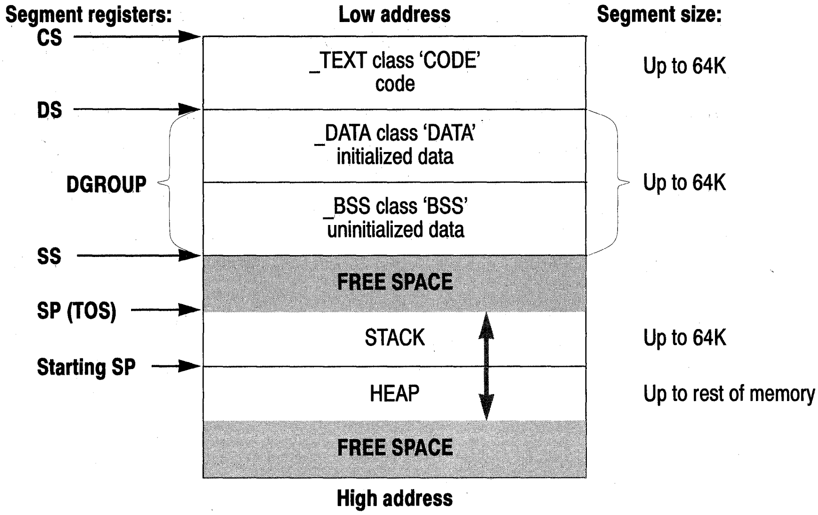


Figure 1.7 Large model memory segmentation

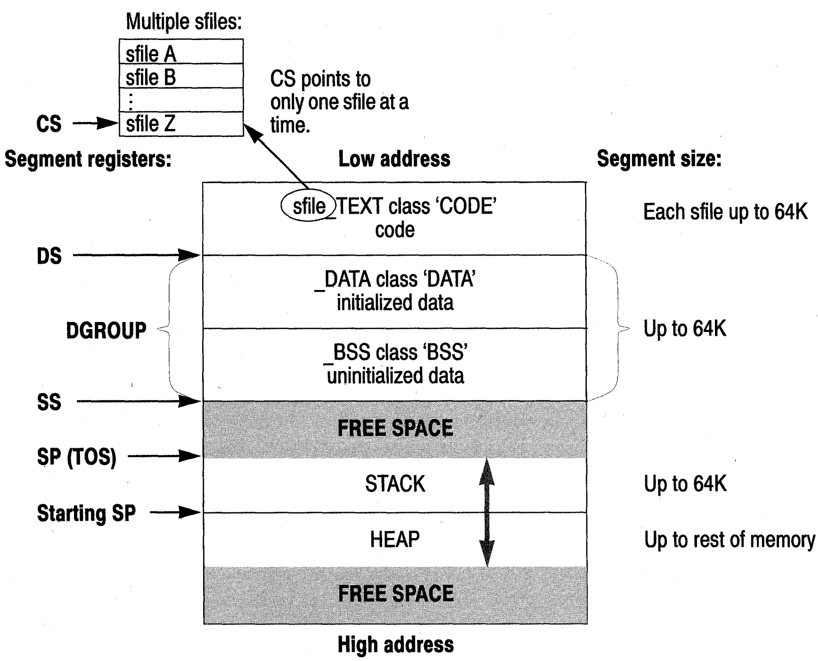


Figure 1.8 Huge model memory segmentation

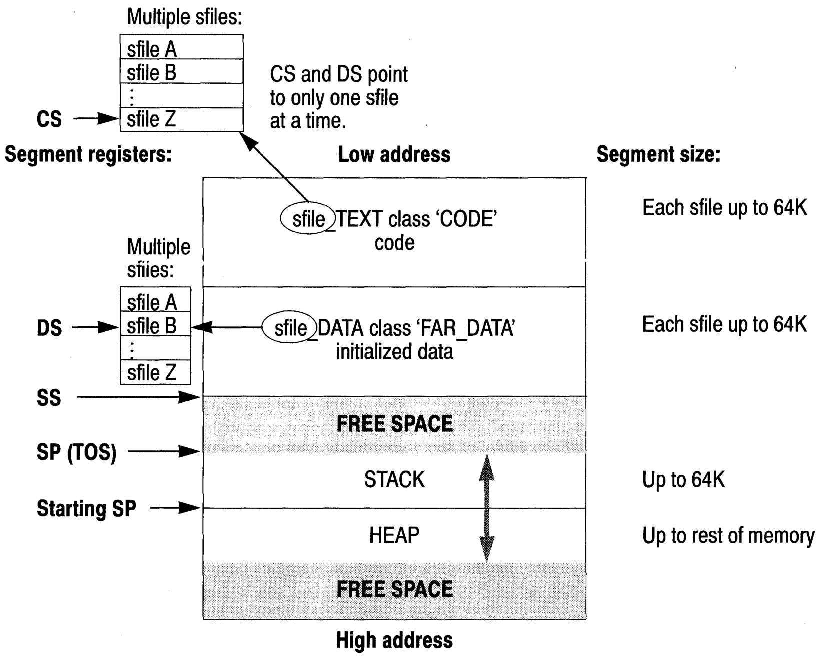


Table 1.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (16 MB); these groups correspond to the rows and columns in Table 1.1.

Table 1.1 Memory models

	Code size	
Data size	**64K**	**16MB**
64K	Tiny (data, code overlap; total size = 64K)	
	Small (no overlap; total size = 128K)	Medium (small data, large code)
16 MB	Compact (large data, small code)	Large (large data, code)
		Huge (same as large but static data > 64K)

The models tiny, small, and compact are small code models because, by default, code pointers are near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code models (medium, large, or huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in *each* module.

Mixed-model programming: Addressing modifiers

Borland C++ introduces eight new keywords not found in standard ANSI C. These keywords are **__near**, **__far**, **__huge**, **__cs**, **__ds**, **__es**, **__ss**, and **__seg**. These keywords can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Borland C++, you can modify the declarations of pointers, objects, and functions with the keywords **__near**, **__far**, or **__huge**. The **__near**, **__far**, and **__huge** data pointers are described earlier in this chapter. You can declare far objects using the **__far** keyword. **__near** functions are invoked with near calls and exit with near returns. Similarly, **__far** functions are called **__far** and return far values. **__huge** functions are like **__far** functions, except that **__huge** functions set DS to a new value, and **__far** functions do not.

There are also four special **__near** data pointers: **__cs**, **__ds**, **__es**, and **__ss**. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

then *p* would contain a 16-bit offset into the stack segment.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The next table shows how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 MB memory space (far, has its own segment address).

Table 1.2 Pointer results

Memory model	Function pointers	Data pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Segment pointers

Use **__seg** in segment pointer type declarators. The resulting pointers are 16-bit segment pointers. The syntax for **__seg** is:

datatype _seg **identifier*;

For example,

```
int _seg *name;
```

Any indirection through *identifier* has an assumed offset of 0. In arithmetic involving segment pointers the following rules hold true:

1. You can't use the **++**, **– –**, **+=**, or **–=** operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. When adding a near pointer to a segment pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer. Therefore, the two pointers must either point to the same type, or one must be a pointer to void. There is no multiplication of the offset regardless of the type pointed to.
4. When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
5. When adding or subtracting an integer operand to or from a segment pointer, the result is a far pointer, with the segment taken from the segment pointer and the offset found by multiplying the size of the object pointed to by the integer operand. The arithmetic is performed as if the integer were added to or subtracted from the far pointer.
6. Segment pointers can be assigned, initialized, passed into and out of functions, compared and so forth. (Segment pointers are compared as if their values were **unsigned** integers.) In other words, other than the above restrictions, they are treated exactly like any other pointer.

Declaring far objects

You can declare far objects in Borland C++. For example,

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

The command-line compiler options **–zE**, **–zF**, and **–zH** (which can also be set using **#pragma option**) affect the far segment name, class, and group, respectively. When you use **#pragma option**, you can make them apply to any ensuing far object declarations. Thus you could use the following sequence to create a far object in a specific segment:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH* -zF*
```

This will put *x* in segment MYSEGMENT 'MYCLASS' in the group 'MYGROUP', then reset all of the far object items to the default values. Note that by using these options, several far objects can be forced into a single segment:

```
#pragma option -zEcombined -zFmyclass
```

```
int far x;
double far y;
#pragma option -zE* -zF*
```

Both *x* and *y* will appear in the segment COMBINED 'MYCLASS' with no group.

Declaring functions to be near or far

On occasion, you'll want (or need) to override the default function type of your memory model.

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp) {
   if (exp <= 0)
      return(1);
   else
      return(x * power(x, exp-1));
   }
```

Every time *power* calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring power as **__near**, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double __near power(double x,int exp)
```

This guarantees that *power* is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you're using a large code model (medium, large, or huge), you can only call *power* from within the module where it is defined. Other modules have their own code segment and thus cannot call **__near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the *power* example at the beginning of this section. It is wise to also declare *power* as static, since it should be called only from within the current module. That way, being a static, its name will not be available to any functions outside the module.

Declaring pointers to be near, far, or huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. For the same reasons given in the preceding section, you might want to modify pointer declarations: either to avoid unnecessary overhead (declaring **__near** when the default would be **__far**) or to reference something outside of the default segment (declaring **__far** or **__huge** when the default would be **__near**).

There are, of course, potential pitfalls in declaring functions and pointers to be of nondefault types. For example, say you have the following small model program:

```
void myputs(s) {
   char *s;
   int  i;
   for (i = 0; s[i] != 0; i++) putc(s[i]);
   }

main() {
   char  near *mystr;

mystr = "Hello, world\n"
   myputs(mystr);
   }
```

This program works fine. In fact, the __**near** declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in *main* is still near (it's still a 16-bit pointer). However, the pointer *s* in *myputs* is now far, because that's the default. This means that *myputs* will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr*.

How do you avoid this problem? If you're going to explicitly declare pointers to be of type __**far** or __**near**, be sure to use function prototypes for any functions that might use them. The solution is to define *myputs* in ANSI C style, like this:

```
void myputs(char *s) {
   /* body of myputs */
   }
```

Now when Borland C++ compiles your program, it knows that *myputs* expects a pointer to **char**; and since you're compiling under the large model, it knows that the pointer must be __**far**. Because of that, Borland C++ will push the data segment (DS) register onto the stack along with the 16-bit value of *mystr*, forming a far pointer.

How about the reverse case: arguments to *myputs* declared as __**far** and compiled with a small data model? Again, without the function prototype, you will have problems, because *main* will push both the offset and the segment address onto the stack, but *myputs* will expect only the offset. With the prototype-style function definitions, though, *main* will only push the offset onto the stack.

Pointing to a given segment:offset address

You can make a far pointer point to a given memory location (a specific segment:offset address). You can do this with the macro *MK_FP*, which takes a segment and an offset and returns a far pointer. For example,

```
MK_FP(segment_value, offset_value)
```

Given a __**far** pointer, *fp*, you can get the segment component with *FP_SEG(fp)* and the offset component with *FP_OFF(fp)*. For more information about these three Borland C++ library routines, refer to the *Library Reference*.

Using library files

Borland C++ offers a version of the standard library routines for each of the six memory models. Borland C++ is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Borland C++ linker, TLINK, directly (as a standalone linker), you need to specify which libraries to use. See Chapter 9 in the *User's Guide* for details on how to do this.

Linking mixed modules

Suppose you compiled one module using the small memory model and another module using the large model, then wanted to link them together. This would present some problems, but they can be solved.

The files would link together fine, but the problems you would encounter would be similar to those described in the earlier section, "Declaring functions to be near or far." If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as described in the earlier section, "Declaring pointers to be near, far, or huge," since a function in the small module would expect to pass and receive **__near** pointers, and a function in the large module would expect **__far** pointers.

The solution, again, is to use function prototypes. Suppose that you put *myputs* into its own module and compile it with the large memory model. Then create a header file called myputs.h (or some other name with a .h extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, put *main* into its own module (called MYMAIN.C), and set things up like this:

```
#include <stdio.h>
#include "myputs.h"

main() {
   char  near *mystr;

   mystr = "Hello, world\n";
   myputs(mystr);
   }
```

When you compile this program, Borland C++ reads in the function prototype from myputs.h and sees that it is a **__far** function that expects a **__far** pointer. Therefore, it generates the proper calling code, even if it's compiled using the small memory model.

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **__far**. To do this, make a copy of each header file you would normally include (such as stdio.h), and rename the copy to something appropriate (such as fstdio.h).

Then edit each function prototype in the copy so that it is explicitly **__far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **__far** calls be made to the routines, but the pointers passed will also be **__far** pointers. Modify your program so that it includes the new header file:

```
#include <fstdio.h>

void main() {
   char  near *mystr;
   mystr = "Hello, world\n";
   printf(mystr);
}
```

Compile your program with the command-line compiler BCC then link it with TLINK, specifying a large model library, such as CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Overlays (VROOMM) for DOS

Overlays are used only in 16-bit DOS programs; you can mark the code segments of a Windows application as discardable to decrease memory consumption. *Overlays* are parts of a program's code that share a common memory area. Only the parts of the program that are required for a given function reside in memory at the same time.

Overlays can significantly reduce a program's total run-time memory requirements. With overlays, you can execute programs that are much larger than the total available memory, since only parts of the program reside in memory at any given time.

How overlays work

Borland C++'s overlay manager (called VROOMM for Virtual Run-time Object-Oriented Memory Manager) is highly sophisticated; it does much of the work for you. In a conventional overlay system, modules are grouped together into a base and a set of overlay units. Routines in a given overlay unit can call other routines in the same unit and routines in the base, but not routines in other units. The overlay units are overlaid against each other; that is, only one overlay unit can be in memory at a time, and each unit occupies the same physical memory. The total amount of memory needed to run the program is the size of the base plus the size of the largest overlay.

This conventional scheme is quite inflexible. It requires complete understanding of the possible calling dependencies in the program, and requires you to have the overlays grouped accordingly. It might be impossible to break your program into overlays if you can't split it into separable calling dependencies.

VROOMM's scheme is quite different. It provides *dynamic segment swapping*. The basic swapping unit is the segment. A segment can be one or more modules. More importantly, any segment can call *any other* segment.

Memory is divided into an area for the base plus a swap area. Whenever a function is called in a segment that is neither in the base nor in the swap area, the segment containing the called function is brought into the swap area, possibly displacing other segments. This is a powerful approach—it is like software virtual memory. You no longer have to break your code into static, distinct, overlay units. You just let it run!

Suppose a segment needs to be brought into the swap area. If there is room for the segment, execution continues. If there is not, then one or more segments in the swap area must be thrown out to make room.

The algorithm for deciding which segment to throw out is quite sophisticated. Here's a simplified version: if there is an inactive segment, choose it for removal. Inactive segments are those without executing functions. Otherwise, pick an active segment and swap it out. Keep swapping out segments until there is enough room available. This technique is called *dynamic swapping*.

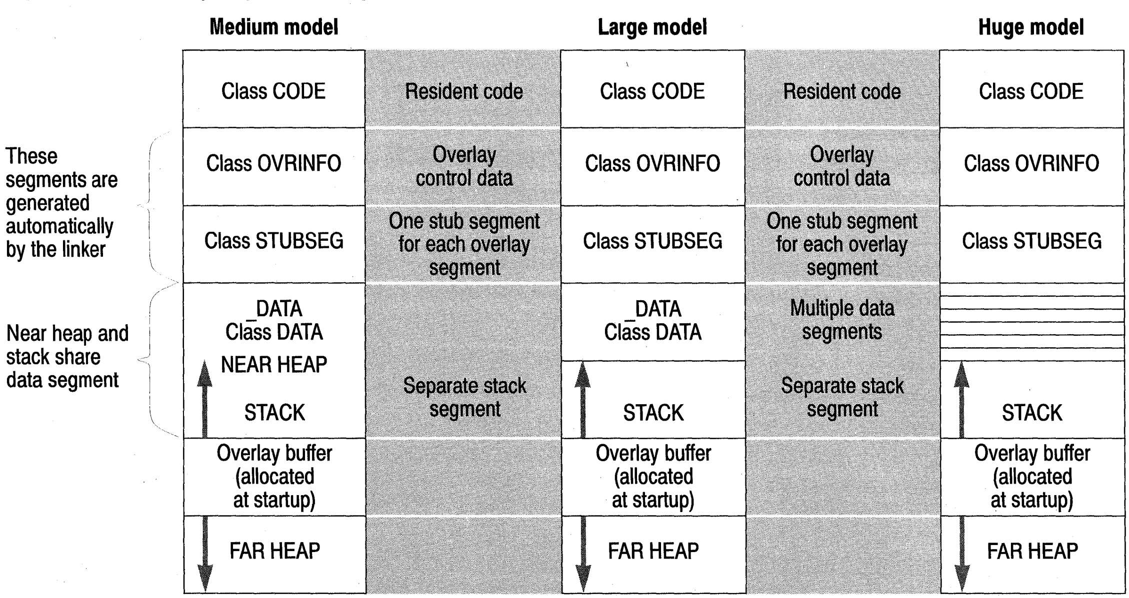
The more memory you provide for the swap area, the better the program performs. The swap area acts like a cache; the bigger the cache, the faster the program runs. The best setting for the size of the swap area is the size of the program's *working set*.

Once an overlay is loaded into memory, it is placed in the overlay buffer, which resides in memory between the stack segment and the far heap. By default, the size of the overlay buffer is estimated and set at startup, but you can change it using the global variable *_ovrbuffer* (see Appendix B). If there isn't enough available memory, an error message is displayed by DOS ("Program too big to fit in memory") or by the C startup code ("Not enough memory to run program").

One important option of the overlay manager is the ability to swap the modules to expanded or extended memory when they are discarded from the overlay buffer. Next time the module is needed, the overlay manager can copy it from where the module was swapped to instead of reading from the file. This makes the overlay manager much faster.

When using overlays, memory is used as shown in the next figure.

Figure 1.9 Memory maps for overlays



Guidelines for using Borland C++ overlays effectively

To get the best out of Borland C++ overlays,

- Minimize resident code (resident run-time library, interrupt handlers, and device drivers are a good starting point).
- Set overlay buffer size to be a comfortable working set (start with 128K and adjust up and down to see the speed/size tradeoff). See page 23 for more information on setting the size of the overlay buffer.
- Think versatility and variety: take advantage of the overlay system to provide support for special cases, interactive help, and other end-user benefits you couldn't consider before.

Requirements

To create overlays, you'll need to remember a few rules:

- The smallest part of a program that can be made into an overlay is a segment.
- Overlaid applications must use the medium, large, or huge programming models; the tiny, small, and compact models are not supported.
- Normal segment merging rules govern overlaid segments. That is, several .OBJ modules can contribute to the same overlaid segment.

The link-time generation of overlays is completely separated from the run-time overlay management; the linker does *not* automatically include code to manage the overlays. In fact, from the linker's point of view, the overlay manager is just another piece of code that gets linked in. The only assumption the linker makes is that the overlay manager takes over an interrupt vector (typically INT 3FH) through which all dynamic loading is controlled. This level of transparency makes it very easy to implement custom-built overlay managers that suit the particular needs of each application.

Exception handling and overlays

If you overlay a C++ program that contains exception-handling constructs, there are a number of situations that you must avoid. The following program elements cannot contain an exception-handling construct:

- Inline functions that are not expanded inline
- Template functions
- Member functions of template classes

Exception-handling constructs include user-written **try/catch** and **__try/__except** blocks. In addition, the compiler can insert exception handlers for blocks with automatic class variables, exception specifications, and some **new/delete** expressions.

If you attempt to overlay any of the above exception-handling constructs, the linker identifies the function and module with the following message:

```
Error: Illegal local public in function_name in module module_name
```

When this error is caused by an inline function, you can rewrite the function so that it is not inline. If the error is caused by a template function, you can do the following:

- Remove all exception-handling constructs from the function
- Remove the function from the overlay module

You need to pay special attention when overlaying a program that uses multiple inheritance. An attempt to overlay a module that defines or uses class constructors or destructors that are required for a multiple inheritance class can cause the linker to generate the following message:

```
Error: Illegal local public in class_name:: in module module_name
```

When such a message is generated, the module identified by the linker message should not be overlaid.

The container classes (in the BIDS?.LIB) have the exception-handling mechanism turned off by default. However, the diagnostic version of BIDS throws exceptions and should not be used with overlays. By default, the *string* class can throw exceptions and should not be used in programs that use overlays. See the *Class Libraries Guide* for a discussion of BIDS and the *string* class.

Using overlays

Overlays can be used only in 16-bit DOS programs. To overlay a program, all of its modules must be compiled with the **–Y** compiler option enabled. To make a particular module into an overlay, it needs to be compiled with the **–Yo** option. (**–Yo** automatically enables **–Y**.)

The **–Yo** option applies to all modules and libraries that follow it on the command line; you can disable it with **–Yo–**. These are the only command line options that are allowed to follow file names. For example, to overlay the module OVL.C but not the library GRAPHICS.LIB, either of the following command lines could be used:

```
BCC -ml -Yo ovl.c -Yo- graphics.lib
```

or

```
BCC -ml graphics.lib -Yo ovl.c
```

If TLINK is invoked explicitly to link the .EXE file, the **/o** linker option must be specified on the linker command line or response file. See the *User's Guide* for details on how to use the **/o** option.

Overlay example

Suppose that you want to overlay a program consisting of three modules: MAIN.C, O1.C, and O2.C. Only the modules O1.C and O2.C should be made into overlays. (MAIN.C contains time-critical routines and interrupt handlers, so it should stay resident.) Let's assume that the program uses the large memory model.

The following command accomplishes the task:

```
BCC -ml -Y main.c -Yo o1.c o2.c
```

The result will be an executable file MAIN.EXE, containing two overlays.

Note See the discussion of TargetExpert in the *User's Guide* for information on programming with overlays.

Overlaid programs

This section discusses issues vital to well-behaved overlaid applications.

The far call requirement

Use a large code model (medium, large, or huge) when you want to compile an overlay module. At any call to an overlaid function in another module, you *must* guarantee that all currently active functions are far.

You *must* compile all overlaid modules with the **–Y** option, which makes the compiler generate code that can be overlaid.

Note Failing to observe the far call requirement in an overlaid program will cause unpredictable and possibly catastrophic results when the program is executed.

Buffer size

The default overlay buffer size is twice the size of the largest overlay. This is adequate for some applications. But imagine that a particular function of a program is implemented through many modules, each of which is overlaid. If the total size of those modules is larger than the overlay buffer, a substantial amount of swapping will occur if the modules make frequent calls to each other.

The solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. You can do this by setting the *_ovrbuffer* global variable (see Appendix B) to the required size in paragraphs. For example, to set the overlay buffer to 128K, include the following statement in your code:

```
unsigned _ovrbuffer = 0x2000;
```

There is no general formula for determining the ideal overlay buffer size.

What not to overlay

Exception-handling constructs in overlays require special attention. See page 21 for a discussion of exception handling.

Don't overlay modules that contain interrupt handlers, or small and time-critical routines. Due to the non-reentrant nature of the DOS operating system, modules that might be called by interrupt functions should not be overlaid.

Borland C++'s overlay manager fully supports passing overlaid functions as arguments, assigning and initializing function pointer variables with addresses of overlaid functions, and calling overlaid routines via function pointers.

Debugging overlays

Most debuggers have very limited overlay debugging capabilities, if any at all. Not so with Borland C++'s Turbo Debugger, the standalone debugger. The debugger fully supports single-stepping and breakpoints in overlays in a manner completely transparent to you. By using overlays, you can easily engineer and debug huge applications—all by using Turbo Debugger.

Note Overlays should not be used with any diagnostic version of the BIDS libraries.

External routines in overlays

Like normal C functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid functions, the assembly language routine *must* be declared FAR, and it *must* set up a stack frame using the BP register. For example, assuming that *OtherFunc* is an overlaid function in another module, and that the assembly language routine *ExternFunc* calls it, then *ExternFunc* must be FAR and set up a stack frame, as shown:

```
ExternFunc      PROC    FAR
   push    bp              ;Save BP
   mov     bp,sp           ;Set up stack frame
   sub     sp,LocalSize    ;Allocate local variables
   :
   call    OtherFunc       ;Call another overlaid module
   :
   mov     sp,bp           ;Dispose local variables
   pop     bp              ;Restore BP
   RET                     ;Return
ExternFunc      ENDP
```

where *LocalSize* is the size of the local variables. If *LocalSize* is zero, you can omit the two lines to allocate and dispose local variables, but you must not omit setting up the BP stack frame even if you have no arguments or variables on the stack.

These requirements are the same if *ExternFunc* makes *indirect* references to overlaid functions. For example, if *OtherFunc* makes calls to overlaid functions, but is not itself overlaid, *ExternFunc* must be FAR and still has to set up a stack frame.

In the case where an assembly language routine doesn't make any direct or indirect references to overlaid functions, there are no special requirements; the assembly language routine can be declared NEAR. It does not have to set up a stack frame.

Overlaid assembly language routines should *not* create variables in the code segment, since any modifications made to an overlaid code segment are lost when the overlay is disposed. Likewise, pointers to objects based in an overlaid code segment cannot be expected to remain valid across calls to other overlays, since the overlay manager freely moves around and disposes overlaid code segments.

Swapping

If you have expanded or extended memory available, you can tell the overlay manager to use it for swapping. If you do so, when the overlay manager has to discard a module from the overlay buffer (because it should load a new module and the buffer is full), it can store the discarded module in this memory. Any later loading of this module is reduced to in-memory transfer, which is significantly faster than reading from a disk file.

In both cases there are two possibilities: the overlay manager can either detect the presence of expanded or extended memory and can take it over by itself, or it can use an already detected and allocated portion of memory. For extended memory, the detection of the memory use is not always successful because of the many different cache and RAM disk programs that can take over extended memory without any mark. To avoid this problem, you can tell the overlay manager the starting address of the extended memory and how much of it is safe to use.

Borland C++ provides two functions that allow you to initialize expanded and extended memory. See Chapter 5 for a description of the *_OvrInitEms* and *_OvrrInitExt* functions.

Chapter

2

Math

This chapter describes the floating-point options and explains how to use *complex* and *bcd* numerical types.

Floating-point I/O

Floating-point output requires linking of conversion routines used by *printf*, *scanf*, and any variants of these functions. To reduce executable size, the floating-point formats are not automatically linked. However, this linkage is done automatically whenever your program uses a mathematical routine or the address is taken of some floating-point number. If neither of these actions occur, the missing floating-point formats can result in a run-time error.

The following program illustrates how to set up your program to properly execute.

```
/* PREPARE TO OUTPUT FLOATING-POINT NUMBERS. */
   #include <stdio.h>

   #pragma extref _floatconvert

   void main() {
      printf("d = %f\n", 1.3);
      }
```

Floating-point options

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor can easily handle integer values, but more time and effort are required to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

Note If you have an 80486 or Pentium processor, the numeric coprocessor is probably already built in.

Emulating the 80x87 chip

The default Borland C++ code-generation option is *emulation* (the **–f** command-line compiler option). This option is for programs that might or might not have floating point, and for machines that might or might not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it uses the 80x87 if it is present; if no coprocessor is present at run time, it uses special software that emulates the 80x87. This software uses 512 bytes of your stack, so make allowance for it when using the emulation option and set your stack size accordingly.

Using the 80x87 code

If your program is going to run only on machines that have an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Choose the 80x87 floating-point code-generation option (the **–f87** command-line compiler option). Borland C++ will then link your programs with FP87.LIB instead of with EMU.LIB.

No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code-generation option (the **–f–** command-line compiler option). Then Borland C++ will not link with EMU.LIB, FP87.LIB, or MATH*x*.LIB.

Fast floating-point option

Borland C++ has a fast floating-point option (the **–ff** command-line compiler option). It can be turned off with **–ff–** on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;
x = (float)(3.5*x);
```

To execute this correctly, *x* is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in *x*. Under the fast floating-point option, the **long**

double product is converted directly to a **double**. Since very few programs depend on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

The 87 environment variable

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

There are some situations in which you might want to override this default autodetection behavior. For example, your own run-time system might have an 80x87, but you might need to verify that your program will work as intended on systems without a coprocessor. Or your program might need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

Borland C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces on either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.

Note Setting the 87 environment variable to *Y* (for Yes) means that the coprocessor is there, and you want the program to use it. *Let the programmer beware:* If you set 87 = *Y* when, in fact, there is no 80x87 available on that system, your system will hang.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press ***Enter*** immediately after typing the equal sign.

Registers and the 80x87

When you use floating point, make note of these points about registers:

- In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
- If you are mixing floating point with inline assembly, you might need to take special care when using 80x87 registers. Unless you are sure that enough free registers exist, you might need to save and pop the 80x87 registers before calling functions that use the coprocessor.

Disabling floating-point exceptions

By default, Borland C++ programs abort if a floating-point overflow or divide-by-zero error occurs. You can mask these floating-point exceptions by a call to *_control87* in *main*, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
   _control87(MCW_EM,MCW_EM);
   ⋮
}
```

You can determine whether a floating-point exception occurred after the fact by calling *_status87* or *_clear87*. See the *Library Reference* entries for these functions for details.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN is likely to cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of *_matherr* into your program:

```
#include <math.h>
int _matherr(struct _exception *e)
{
   return 1;              /* error has been handled */
}
```

Any other use of *_matherr* to intercept math errors is not encouraged; it is considered obsolete and might not be supported in future versions of Borland C++.

Using complex types

Complex numbers are numbers of the form $x + yi$, where x and y are real numbers, and i is the square root of -1. Borland C++ has always had a type

```
struct complex
{
   double  x, y;
};
```

defined in math.h. This type is convenient for holding complex numbers, because they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

A significant advantage to using the Borland C++ *complex* numerical type is that all of the ANSI C Standard mathematical routines are defined to operate with it. These mathematical routines are not defined for use with the C **struct complex**.

Note See the *Class Libraries Guide* for more information.

To use complex numbers in C++, all you have to do is to include complex.h. In complex.h, all the following have been overloaded to handle complex numbers:

- All of the binary arithmetic operators.
- The input and output operators, >> and <<.
- The ANSI C math functions.

The complex library is invoked only if the argument is of type *complex*. Thus, to get the complex square root of -1, use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

The following functions are defined by class *complex*:

```
double  arg(complex&);     // angle in the plane
complex conj(complex&);    // complex conjugate
double  imag(complex&);    // imaginary part
double  norm(complex&);    // square of the magnitude
double   real(complex&);   // real part
// Use polar coordinates to create a complex.
complex polar(double mag, double angle = 0);
```

Using bcd types

Borland C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This can sometimes be confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.

Note See the *Class Libraries Guide* for more information.

Binary numbers are preferable for most applications, but in some situations the round-off error involved in converting between base 2 and 10 is undesirable. The most common example of this is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```
#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n",x);
```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny round-off error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Borland C++ offers the C++ type *bcd*, which is declared in bcd.h. With *bcd*, the number 0.01 is represented exactly, and the *bcd* variable *x* provides an exact penny count.

```
#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";
```

Here are some facts to keep in mind about *bcd*:

- *bcd* does not eliminate all round-off error: A computation like 1.0/3.0 will still have round-off error.
- *bcd* types can be used with ANSI C math functions.
- *bcd* numbers have about 17 decimal digits precision, and a range of about 1×10^{-125} to 1×10^{125}.

Converting bcd numbers

bcd is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is performed only when at least one operand is of the type *bcd*.

Note The *bcd* member function *real* is available for converting a *bcd* number back to one of the usual formats (**float**, **double**, or **long double**), though the conversion is not done automatically. *real* does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example, a *bcd* can be printed using any of the following four output statements with *cout* and *printf*.

```
/* PRINTING bcd NUMBERS */
/* This must be compiled as a C++ program. */
#include <bcd.h>
#include <iostream.h>
#include <stdio.h>

void main(void) {
   bcd a = 12.1;
   double x = real(a); // This conversion required for printf().

   printf("\na = %g", x);
   printf("\na = %Lg", real(a));
   printf("\na = %g", (double)real(a));
   cout << "\na = " << a; // The preferred method.
   }
```

Note that since *printf* doesn't do argument checking, the format specifier must have the *L* if the **long double** value *real(a)* is passed.

Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a *bcd*. The number of places is an optional second argument to the constructor *bcd*. For example, to convert $1000.00/7 to a *bcd* variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

`1000.00/7`	=	142.85714...
`bcd(1000.00/7, 2)`	=	142.860
`bcd(1000.00/7, 1)`	=	142.900
`bcd(1000.00/7, 0)`	=	143.000
`bcd(1000.00/7, -1)`	=	140.000
`bcd(1000.00/7, -2)`	=	100.000

The number is rounded using banker's rounding (as specified by IEEE), which rounds to the nearest whole number, with ties being rounded to an even digit. For example,

`bcd(12.335, 2)`	=	12.34
`bcd(12.345, 2)`	=	12.34
`bcd(12.355, 2)`	=	12.36

Chapter

3

Video functions

Borland C++ comes with a complete library of graphics functions, so you can produce onscreen charts and diagrams. The graphics functions are available for 16-bit DOS-only applications. This chapter briefly discusses video modes and windows, then explains how to program in graphics mode.

Video modes

Your PC has some type of video adapter. This can be a Monochrome Display Adapter (MDA) for text-only display, or it can be a graphics adapter, such as a Color/Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), a Video Graphics Array adapter (VGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80 or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color or black and white).

The screen's operating mode is defined when your program calls one of the mode-defining functions *textmode*, *initgraph*, or *setgraphmode*.

- In *text mode*, your PC's screen is divided into cells (80- or 40-columns wide by 25, 43, or 50 lines high). Each cell consists of a character and an attribute. The character is the displayed ASCII character; the attribute specifies *how* the character is displayed (its color, intensity, and so on). Borland C++ provides a full range of routines for manipulating the text screen, for writing text directly to the screen, and for controlling cell attributes.
- In *graphics mode*, your PC's screen is divided into pixels; each pixel displays a single dot onscreen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Borland C++'s graphics library to create graphic displays onscreen: You can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper left corner of the screen is position (1,1), with x-coordinates increasing from left to right, and y-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper left corner is position (0,0), with the x- and y-coordinate values increasing in the same manner.

Windows and viewports

Borland C++ provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you aren't familiar with windows and viewports, you should read this brief overview. Borland C++'s window- and viewport-management functions are explained in the "Programming in graphics mode" section.

A *window* is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default window to a text window smaller than the full screen (with a call to the *window* function, which specifies the window's position in terms of screen coordinates).

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a *viewport*. When your graphics program outputs drawings and so on, the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the *setviewport* function.

Except for these window- and viewport-defining functions, all *coordinates* for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

Programming in graphics mode

This section provides a brief summary of the functions used in graphics mode. For more detailed information about these functions, refer to Chapter 4.

Borland C++ provides a separate library of over 70 graphics functions, ranging from high-level calls (like *setviewport*, *bar3d*, and *drawpoly*) to bit-oriented functions (like *getimage* and *putimage*). The graphics library supports numerous fill and line styles, and provides several text fonts that you can size, justify, and orient horizontally or vertically.

These functions are in the library file GRAPHICS.LIB, and they are prototyped in the header file graphics.h. In addition to these two files, the graphics package includes graphics device drivers (*.BGI files) and stroked character fonts (*.CHR files); these files are discussed in following sections.

To use the graphics functions with the BCC.EXE command-line compiler, you have to list GRAPHICS.LIB on the command line. For example, if your program MYPROG.C uses graphics, the BCC command line would be

```
BCC MYPROG GRAPHICS.LIB
```

See the *User's Guide* discussion of TargetExpert for a description of DOS programming with graphics. When you make your program, the linker automatically links in the Borland C++ graphics library.

Note Because graphics functions use **far** pointers, graphics aren't supported in the tiny memory model.

There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, and so on, which are memory-model specific). Each function in GRAPHICS.LIB is a **far** function, and those graphics functions that take pointers take **far** pointers. For these functions to work correctly, it is important that you #include graphics.h in every module that uses graphics.

The graphics library functions

There are seven categories of Borland C++ graphics functions:

- Graphics system control
- Drawing and filling
- Manipulating screens and viewports
- Text output
- Color control
- Error handling
- State query

Graphics system control

Here's a summary of the graphics system control:

Function	Description
closegraph	Shuts down the graphics system.
detectgraph	Checks the hardware and determines which graphics driver to use; recommends a mode.
graphdefaults	Resets all graphics system variables to their default settings.
_graphfreemem	Deallocates graphics memory; hook for defining your own routine.
_graphgetmem	Allocates graphics memory; hook for defining your own routine.
getgraphmode	Returns the current graphics mode.
getmoderange	Returns lowest and highest valid modes for specified driver.
initgraph	Initializes the graphics system and puts the hardware into graphics mode.
installuserdriver	Installs a vendor-added device driver to the BGI device driver table.
installuserfont	Loads a vendor-added stroked font file to the BGI character file table.
registerbgidriver	Registers a linked-in or user-loaded driver file for inclusion at link time.
restorecrtmode	Restores the original (pre-*initgraph*) screen mode.

Function	Description
setgraphbufsize	Specifies size of the internal graphics buffer.
setgraphmode	Selects the specified graphics mode, clears the screen, and restores all defaults.

Borland C++'s graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color/Graphics Adapter (CGA)
- Multi-Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

To start the graphics system, you first call the *initgraph* function. *initgraph* loads the graphics driver and puts the system into graphics mode.

You can tell *initgraph* to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver. If you tell *initgraph* to autodetect, it calls *detectgraph* to select a graphics driver and mode. If you tell *initgraph* to use a particular graphics driver and mode, you must be sure that the hardware is present. If you force *initgraph* to use hardware that is not present, the results will be unpredictable.

Once a graphics driver has been loaded, you can use the *gerdrivername* function to find out the name of the driver and the *getmaxmode* function to find out how many modes a driver supports. *getgraphmode* will tell you which graphics mode you are currently in. Once you have a mode number, you can find out the name of the mode with *getmodename*. You can change graphics modes with *setgraphmode* and return the video mode to its original state (before graphics was initialized) with *restorecrtmode*. *restorecrtmode* returns the screen to text mode, but it does not close the graphics system (the fonts and drivers are still in memory).

graphdefaults resets the graphics state's settings (viewport size, draw color, fill color and pattern, and so on) to their default values.

installuserdriver and *installuserfont* let you add new device drivers and fonts to your BGI.

Finally, when you're through using graphics, call *closegraph* to shut down the graphics system. *closegraph* unloads the driver from memory and restores the original video mode (via *restorecrtmode*).

A more detailed discussion

The previous discussion provided an overview of how *initgraph* operates. In the following paragraphs, we describe the behavior of *initgraph*, *_graphgetmem*, and *_graphfreemem* in some detail.

Normally, the *initgraph* routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGIOBJ utility—see UTILS.TXT, included with your distribution disks), then placing calls to *registerbgidriver* in your source code (before the call to *initgraph*) to *register* the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (via *detectgraph*), *initgraph* checks to see if the desired driver has been registered. If so, *initgraph* uses the registered driver directly from memory. Otherwise, *initgraph* allocates memory for the driver and loads the .BGI file from disk.

Note Using *registerbgidriver* is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Chapter 4.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls *_graphgetmem* to allocate memory and *_graphfreemem* to free memory. By default, these routines call *malloc* and *free*, respectively.

You can override this default behavior by defining your own *_graphgetmem* and *_graphfreemem* functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: they will override the default functions with the same names that are in the standard C libraries.

Note If you provide your own *_graphgetmem* or *_graphfreemem*, you might get a "duplicate symbols" warning message. Just ignore the warning.

Drawing and filling

Here's a quick summary of the drawing and filling functions:

Drawing functions	Description
arc	Draws a circular arc.
circle	Draws a circle.
drawpoly	Draws the outline of a polygon.
ellipse	Draws an elliptical arc.
getarccoords	Returns the coordinates of the last call to *arc* or *ellipse*.
getaspectratio	Returns the aspect ratio of the current graphics mode.
getlinesettings	Returns the current line style, line pattern, and line thickness.
line	Draws a line from (*x0,y0*) to (*x1,y1*).
linerel	Draws a line to a point some relative distance from the current position (CP).
lineto	Draws a line from the current position (CP) to (*x,y*).
moveto	Moves the current position (CP) to (*x,y*).
moverel	Moves the current position (CP) a relative distance.
rectangle	Draws a rectangle.
setaspectratio	Changes the default aspect ratio-correction factor.
setlinestyle	Sets the current line width and style.

Filling functions	Description
bar	Draws and fills a bar.
bar3d	Draws and fills a 3-D bar.
fillellipse	Draws and fills an ellipse.
fillpoly	Draws and fills a polygon.
floodfill	Flood-fills a bounded region.
getfillpattern	Returns the user-defined fill pattern.
getfillsettings	Returns information about the current fillpattern and color.
pieslice	Draws and fills a pie slice.
sector	Draws and fills an elliptical pie slice.
setfillpattern	Selects a user-defined fill pattern.
setfillstyle	Sets the fill pattern and fill color.

With Borland C++'s drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pie slices, two- and three-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape) with one of eleven predefined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the current position (CP).

You draw lines and unfilled shapes with the functions *arc*, *circle*, *drawpoly*, *ellipse*, *line*, *linerel*, *lineto*, and *rectangle*. You can fill these shapes with *floodfill*, or combine drawing and filling into one step with *bar*, *bar3d*, *fillellipse*, *fillpoly*, *pieslice*, and *sector*. You use *setlinestyle* to specify whether the drawing line (and border line for filled shapes) is thick or thin, and whether its style is solid, dotted, and so forth, or some other line pattern you've defined. You can select a predefined fill pattern with *setfillstyle*, and define your own fill pattern with *setfillpattern*. You move the CP to a specified location with *moveto*, and move it a specified displacement with *moverel*.

To find out the current line style and thickness, call *getlinesettings*. For information about the current fill pattern and fill color, call *getfillsettings*; you can get the user-defined fill pattern with *getfillpattern*.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with *getaspectratio*, and the coordinates of the last drawn arc or ellipse with *getarccoords*. If your circles aren't perfectly round, use *setaspectratio* to correct them.

Manipulating the screen and viewport

Here's a quick summary of the screen-, viewport-, image-, and pixel-manipulation functions

Function	Description
Screen manipulation	
cleardevice	Clears the screen (active page).
setactivepage	Sets the active page for graphics output.
setvisualpage	Sets the visual graphics page number.

Function	Description
Viewport manipulation	
clearviewport	Clears the current viewport.
getviewsettings	Returns information about the current viewport.
setviewport	Sets the current output viewport for graphics output.
Image manipulation	
getimage	Saves a bit image of the specified region to memory.
imagesize	Returns the number of bytes required to store a rectangular region of the screen.
putimage	Puts a previously saved bit image onto the screen.
Pixel manipulation	
getpixel	Gets the pixel color at (*x*,*y*).
putpixel	Plots a pixel at (*x*,*y*).

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one step with a call to *cleardevice*; this routine erases the entire screen and homes the CP in the viewport, but leaves all other graphics system settings intact (the line, fill, and text styles; the palette; the viewport settings; and so on).

Depending on your graphics adapter, your system has between one and four screen-page buffer; these are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify the active screen page (where graphics functions place their output) with *setactivepage* and the visual page (the one displayed onscreen) with *setvisualpage*.

Once your screen is in graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to *setviewport*. You define the viewport's position in terms of absolute screen coordinates and specify whether clipping is on (active) or off. You clear the viewport with *clearviewport*. To find out the current viewport's absolute screen coordinates and clipping status, call *getviewsettings*.

You can capture a portion of the onscreen image with *getimage*, call *imagesize* to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with *putimage*.

The coordinates for all output functions (drawing, filling, text, and so on) are viewport-relative.

You can also manipulate the color of individual pixels with the functions *getpixel* (which returns the color of a given pixel) and *putpixel* (which plots a specified pixel in a given color).

Text output in graphics mode

Here's a quick summary of the graphics-mode text output functions:

Function	Description
gettextsettings	Returns the current text font, direction, size, and justification.
outtext	Sends a string to the screen at the current position (CP).

Function	Description
outtextxy	Sends a string to the screen at the specified position.
registerbgifont	Registers a linked-in or user-loaded font.
settextjustify	Sets text justification values used by outtext and outtextxy.
settextstyle	Sets the current text font, style, and character magnification factor.
setusercharsize	Sets width and height ratios for stroked fonts.
textheight	Returns the height of a string in pixels.
textwidth	Returns the width of a string in pixels.

The graphics library includes an 8X8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it retains good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either *outtext* or *outtextxy*, and you control the justification of the output text (with respect to the CP) with *settextjustify*. You choose the character font, direction (horizontal or vertical), and size (scale) with *settextstyle*. You can find out the current text settings by calling *gettextsettings*, which returns the current text font, justification, magnification, and direction in a *textsettings* structure. *setusercharsize* lets you modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by *outtext* and *outtextxy* are clipped at the viewport borders. If clipping is *off*, these functions throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

To determine the onscreen size of a given text string, call *textheight* (which measures the string's height in pixels) and *textwidth* (which measures its width in pixels).

The default 8×8 bit-mapped font is built into the graphics package, so it's always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGIOBJ utility) and linked into your .EXE file.

Normally, the *settextstyle* routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR file to an .OBJ file (using the BGIOBJ utility—you can read about it in UTILS.TXT, the online documentation included with your distribution disks), then placing calls to *registerbgifont* in your source code (before the call to *settextstyle*) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

Note Using *registerbgifont* is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in UTILS.TXT, which is included with your distribution disks.

Color control

Here's a quick summary of the color control functions:

Function	Description
Get color information	
getbkcolor	Returns the current background color.
getcolor	Returns the current drawing color.
getdefaultpalette	Returns the palette definition structure.
getmaxcolor	Returns the maximum color value available in the current graphics mode.
getpalette	Returns the current palette and its size.
getpalettesize	Returns the size of the palette look-up table.
Set one or more colors	
setallpalette	Changes all palette colors as specified.
setbkcolor	Sets the current background color.
setcolor	Sets the current drawing color.
setpalette	Changes one palette color as specified by its arguments.

Before summarizing how these color control functions work, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and palettes

The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot onscreen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette*. The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The number of colors in this subset is equal to the number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* is 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* - 1). *getmaxcolor* returns the highest valid pixel value (*size* - 1) for the current graphics driver and mode.

When we discuss the Borland C++'s graphics functions, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is a pixel's value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, and so on) haven't changed.

Background and drawing color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You choose a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between CGA and EGA, so they're presented separately. Color control on the AT&T driver, and the lower resolutions of the MCGA driver is similar to CGA.

On the CGA, you can choose to display your graphics in low resolution (320×200), which allows you to use four colors, or in high resolution (640×200), in which you can use two colors.

CGA low resolution

In the low-resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can set only the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color; it can be any one of the 16 available colors (see the following table of CGA background colors).

You choose which palette you want by selecting the appropriate mode (CGAC0, CGAC1, CGAC2, CGAC3); these modes use color palette 0 through color palette 3, as detailed in the following table. The CGA drawing colors and the equivalent constants are defined in graphics.h.

Palette number	Constant assigned to color number (pixel value): 1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

To assign one of these colors as the CGA drawing color, call **setcolor** with either the color number or the corresponding constant name as an argument; for example, if you're using palette 3 and you want to use cyan as the drawing color:

```
setcolor(1);
```

or

```
setcolor(CGA_CYAN);
```

The available CGA background and foreground colors, defined in graphics.h, are listed in the following table:

Numeric value	Symbolic name	Numeric value	Symbolic name
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

To assign one of these colors to the CGA background color, use *setbkcolor(color)*, where color is one of the entries in the preceding table. For CGA, this color is not a pixel value (palette index); it directly specifies the actual color to be put in the first palette entry.

CGA high resolution

In high-resolution mode (640x200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its background color; you set it with the *setbkcolor* routine. (Strange but true.)

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

CGA palette routines

Because the CGA palette is predetermined, you shouldn't use the *setallpalette* routine on a CGA. Also, you shouldn't use *setpalette*(*index*, *actual_color*), except for *index* = 0. (This is an alternate way to set the CGA background color to *actual_color*.)

Color control on the EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors; each entry is user-settable. You can retrieve the current palette with *getpalette*, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with *setpalette*, or all at once with *setallpalette*.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in graphics.h that contain the corresponding hardware color values: these are EGA_BLACK, EGA_WHITE, and so on. You can also get these values with *getpalette*.

The *setbkcolor*(*color*) routine behaves differently on an EGA than on a CGA. On an EGA, *setbkcolor* copies the actual color value that's stored in entry #*color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error handling in graphics mode

Here's a quick summary of the graphics-mode error-handling functions:

Function	Description
grapherrormsg	Returns an error message string for the specified error code.
graphresult	Returns an error code for the last graphics operation that encountered a problem.

If an error occurs when a graphics library function is called (such as a font requested with *settextstyle* not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling *graphresult*. A call to *grapherrormsg(graphresult())* returns the error strings listed in the following table.

The error return-code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when *initgraph* executes successfully or when you call *graphresult*. Therefore, if you want to know which graphics function returned which error, you should store the value of *graphresult* into a temporary variable and then test it.

Error code	*graphics_errors* constant	Corresponding error message string
0	grOk	No error
-1	grNoInitGraph	(BGI) graphics not installed (use *initgraph*)
-2	grNotDetected	Graphics hardwaren't detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	Not enough memory to load driver
-6	grNoScanMem	Out of memory in scan fill
-7	grNoFloodMem	Out of memory in flood fill
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

State query

The following table summarizes the graphics mode state query functions:

Table 3.1 Graphics mode state query functions

Function	Returns
getarccoords	Information about the coordinates of the last call to *arc* or *ellipse*.
getaspectratio	Aspect ratio of the graphics screen.
getbkcolor	Current background color.
getcolor	Current drawing color.
getdrivername	Name of current graphics driver.
getfillpattern	User-defined fill pattern.
getfillsettings	Information about the current fill pattern and color.
getgraphmode	Current graphics mode.
getlinesettings	Current line style, line pattern, and line thickness.
getmaxcolor	Current highest valid pixel value.
getmaxmode	Maximum mode number for current driver.
getmaxx	Current *x* resolution.
getmaxy	Current *y* resolution.
getmodename	Name of a given driver mode.
getmoderange	Mode range for a given driver.
getpalette	Current palette and its size.
getpixel	Color of the pixel at *x,y*.
gettextsettings	Current text font, direction, size, and justification.
getviewsettings	Information about the current viewport.
getx	*x* coordinate of the current position (CP).
gety	*y* coordinate of the current position (CP).

Each of Borland C++'s graphics function categories has at least one state query function. These functions are mentioned under their respective categories and also covered here. Each of the Borland C++ graphics state query functions is named get something (except in the error-handling category). Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined in graphics.h, fill that structure with the appropriate information, and return no value.

The state query functions for the graphics system control category are *getgraphmode*, *getmaxmode*, and *getmoderange*: the first returns an integer representing the current graphics driver and mode, the second returns the maximum mode number for a given driver, and the third returns the range of modes supported by a given graphics driver. *getmaxx* and *getmaxy* return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state query functions are *getarccoords*, *getaspectratio*, *getfillpattern*, *getfillsettings*, and *getlinesettings*. *getarccoords* fills a structure with coordinates from the last call to *arc* or *ellipse*; *getaspectratio* tells the current mode's aspect ratio, which the graphics system uses to make circles come out round. *getfillpattern* returns the current user-defined fill pattern. *getfillsettings* fills a structure with the current fill pattern and fill color. *getlinesettings* fills a structure with the current line style (solid, dashed, and so on), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state query functions are *getviewsettings*, *getx*, *gety*, and *getpixel*. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling *getviewsettings*, which fills a structure with the information. *getx* and *gety* return the (viewport-relative) x- and y-coordinates of the CP. *getpixel* returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state query function: *gettextsettings*. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Borland C++'s color-control function category includes four state query functions. *getbkcolor* returns the current background color, and *getcolor* returns the current drawing color. *getpalette* fills a structure with the size of the current drawing palette and the palette's contents. *getmaxcolor* returns the highest valid pixel value for the current graphics driver and mode (palette *size* –1).

Finally, *getmodename* and *getdrivername* return the name of a given driver mode and the name of the current graphics driver, respectively.

Chapter

4

Borland graphics interface

This chapter presents a description, in alphabetical order, of the Borland C++ graphics functions. The graphics functions are available only for 16-bit DOS applications.

arc graphics.h

Function

Draws an arc.

Syntax

```
void far arc(int x, int y, int stangle, int endangle, int radius);
```

Remarks

arc draws a circular arc in the current drawing color centered at (*x*,*y*) with a radius given by *radius*. The *arc* travels from *stangle* to *endangle*. If *stangle* equals 0 and *endangle* equals 360, the call to *arc* draws a complete circle.

The angle for *arc* is reckoned counterclockwise, with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on.

The *linestyle* parameter does not affect arcs, circles, ellipses, or pie slices. Only the *thickness* parameter is used.

If you're using a CGA in high resolution mode or a monochrome graphics adapter, the examples in online Help that show how to use graphics functions might not produce the expected results. If your system runs on a CGA or monochrome adapter, pass the value 1 to those functions that alter the fill or drawing color (*setcolor*, *setfillstyle*, and *setlinestyle*, for example), instead of a symbolic color constant (defined in graphics.h).

Return value

None.

See also
circle, ellipse, fillellipse, getarccoords, getaspectratio, graphresult, pieslice, sector

bar — graphics.h

Function
Draws a two-dimensional bar.

Syntax
```
void far bar(int left, int top, int right, int bottom);
```

Remarks
bar draws a filled-in, rectangular, two-dimensional bar. The bar is filled using the current fill pattern and fill color. *bar* does not outline the bar; to draw an outlined two-dimensional bar, use *bar3d* with *depth* equal to 0.

The upper left and lower right corners of the rectangle are given by (*left*, *top*) and (*right*, *bottom*), respectively. The coordinates refer to pixels.

Return value
None.

See also
bar3d, rectangle, setcolor, setfillstyle, setlinestyle

bar3d — graphics.h

Function
Draws a three-dimensional bar.

Syntax
```
void far bar3d(int left, int top, int right, int bottom, int depth, int topflag);
```

Remarks
bar3d draws a three-dimensional rectangular bar, then fills it using the current fill pattern and fill color. The three-dimensional outline of the bar is drawn in the current line style and color. The bar's depth in pixels is given by *depth*. The *topflag* parameter governs whether a three-dimensional top is put on the bar. If *topflag* is nonzero, a top is put on; otherwise, no top is put on the bar (making it possible to stack several bars on top of one another).

The upper left and lower right corners of the rectangle are given by (*left*, *top*) and (*right*, *bottom*), respectively.

To calculate a typical depth for *bar3d*, take 25% of the width of the bar, like this:

```
bar3d(left,top,right,bottom, (right-left)/4,1);
```

Return value
None.

See also
bar, rectangle, setcolor, setfillstyle, setlinestyle

circle graphics.h

Function
Draws a circle of the given radius with its center at (*x,y*).

Syntax

```
void far circle(int x, int y, int radius);
```

Remarks
circle draws a circle in the current drawing color with its center at (*x,y*) and the radius given by *radius*.

Note The *linestyle* parameter does not affect arcs, circles, ellipses, or pie slices. Only the *thickness* parameter is used.

If your circles are not perfectly round, adjust the aspect ratio.

Return value
None.

See also
arc, ellipse, fillellipse, getaspectratio, sector, setaspectratio

cleardevice graphics.h

Function
Clears the graphics screen.

Syntax

```
void far cleardevice(void);
```

Remarks
cleardevice erases (that is, fills with the current background color) the entire graphics screen and moves the CP (current position) to home (0,0).

Return value
None.

See also
clearviewport

clearviewport — graphics.h

Function
Clears the current viewport.

Syntax

```
void far clearviewport(void);
```

Remarks
clearviewport erases the viewport and moves the CP (current position) to home (0,0), relative to the viewport.

Return value
None.

See also
cleardevice, getviewsettings, setviewport

closegraph — graphics.h

Function
Shuts down the graphics system.

Syntax

```
void far closegraph(void);
```

Remarks
closegraph deallocates all memory allocated by the graphics system, then restores the screen to the mode it was in before you called *initgraph*. (The graphics system deallocates memory, such as the drivers, fonts, and an internal buffer, through a call to *_graphfreemem*.)

Return value
None.

See also
initgraph, setgraphbufsize

detectgraph **graphics.h**

Function
Determines graphics driver and graphics mode to use by checking the hardware.

Syntax

```
void far detectgraph(int far *graphdriver, int far *graphmode);
```

Remarks
detectgraph detects your system's graphics adapter and chooses the mode that provides the highest resolution for that adapter. If no graphics hardware is detected, **graphdriver* is set to grNotDetected (–2), and *graphresult* returns grNotDetected (–2).

**graphdriver* is an integer that specifies the graphics driver to be used. You can give it a value using a constant of the *graphics_drivers* enumeration type, which is defined in graphics.h and listed in the following table.

Table 4.1 detectgraph constants

graphics_drivers constant	Numeric value
CURRENT_DRIVER	–1
DETECT	0 (requests autodetection)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

**graphmode* is an integer that specifies the initial graphics mode (unless **graphdriver* equals DETECT; in which case, **graphmode* is set to the highest resolution available for the detected driver). You can give **graphmode* a value using a constant of the *graphics_modes* enumeration type, which is defined in graphics.h and listed in the following table.

Table 4.2 Graphics drivers information

Graphics driver	*graphics_modes*	Value	Column × row	Palette	Pages
CGA	CGAC0	0	320×200	C0	1
	CGAC1	1	320×200	C1	1
	CGAC2	2	320×200	C2	1
	CGAC3	3	320×200	C3	1
	CGAHI	4	640×200	2 color	1
MCGA	MCGAC0	0	320×200	C0	1
	MCGAC1	1	320×200	C1	1
	MCGAC2	2	320×200	C2	1
	MCGAC3	3	320×200	C3	1
	MCGAMED	4	640×200	2 color	1
	MCGAHI	5	640×480	2 color	1
EGA	EGALO	0	640×200	16 color	4
	EGAHI	1	640×350	16 color	2
EGA64	EGA64LO	0	640×200	16 color	1
	EGA64HI	1	640×350	4 color	1
EGA-MONO	EGAMONOHI	3	640×350	2 color	1*
	EGAMONOHI	3	640×350	2 color	2**
HERC	HERCMONOHI	0	720×348	2 color	2
ATT400	ATT400C0	0	320×200	C0	1
	ATT400C1	1	320×200	C1	1
	ATT400C2	2	320×200	C2	1
	ATT400C3	3	320×200	C3	1
	ATT400MED	4	640×200	2 color	1
	ATT400HI	5	640×400	2 color	1
VGA	VGALO	0	640×200	16 color	2
	VGAMED	1	640×350	16 color	2
	VGAHI	2	640×480	16 color	1
PC3270	PC3270HI	0	720×350	2 color	1
IBM8514	IBM8514HI	0	640×480	256 color	
	IBM8514LO	0	1024×768	256 color	

* 64K on EGAMONO card
** 256K on EGAMONO card

Note The main reason to call *detectgraph* directly is to override the graphics mode that *detectgraph* recommends to initgraph.

Return value

None.

See also
graphresult, initgraph

drawpoly — graphics.h

Function
Draws the outline of a polygon.

Syntax
```
void far drawpoly(int numpoints, int far *polypoints);
```

Remarks
drawpoly draws a polygon with *numpoints* points, using the current line style and color.

**polypoints* points to a sequence of (*numpoints* × 2) integers. Each pair of integers gives the *x* and *y* coordinates of a point on the polygon.

To draw a closed figure with *n* vertices, you must pass *n* + 1 coordinates to *drawpoly* where the *n*th coordinate is equal to the 0th.

Return value
None.

See also
fillpoly, floodfill, graphresult, setwritemode

ellipse — graphics.h

Function
Draws an elliptical arc.

Syntax
```
void far ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
```

Remarks
ellipse draws an elliptical arc in the current drawing color with its center at (*x*,*y*) and the horizontal and vertical axes given by *xradius* and *yradius*, respectively. The ellipse travels from *stangle* to *endangle*. If *stangle* equals 0 and *endangle* equals 360, the call to *ellipse* draws a complete ellipse.

The angle for *ellipse* is reckoned counterclockwise, with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on.

The *linestyle* parameter does not affect arcs, circles, ellipses, or pie slices. Only the *thickness* parameter is used.

Return value
None.

See also
arc, circle, fillellipse, sector

fillellipse graphics.h

Function
Draws and fills an ellipse.

Syntax

```
void far fillellipse(int x, int y, int xradius, int yradius);
```

Remarks
fillellipse draws an ellipse using (*x,y*) as a center point and *xradius* and *yradius* as the horizontal and vertical axes; fills it with the current fill color and pattern.

Return value
None.

See also
arc, circle, ellipse, pieslice

fillpoly graphics.h

Function
Draws and fills a polygon.

Syntax

```
void far fillpoly(int numpoints, int far *polypoints);
```

Remarks
fillpoly draws the outline of a polygon with *numpoints* points in the current line style and color (just as *drawpoly* does), then fills the polygon using the current fill pattern and fill color.

polypoints points to a sequence of (*numpoints* × 2) integers. Each pair of integers gives the *x* and *y* coordinates of a point on the polygon.

Return value
None.

See also

drawpoly, floodfill, graphresult, setfillstyle

floodfill graphics.h

Function

Flood-fills a bounded region.

Syntax

```
void far floodfill(int x, int y, int border);
```

Remarks

floodfill fills an enclosed area on bitmap devices. (*x,y*) is a "seed point" within the enclosed area to be filled. The area bounded by the color *border* is flooded with the current fill pattern and fill color. If the seed point is within an enclosed area, the inside will be filled. If the seed is outside the enclosed area, the exterior will be filled.

Use *fillpoly* instead of *floodfill* whenever possible so that you can maintain code compatibility with future versions.

Note *floodfill* does not work with the IBM-8514 driver.

Return value

If an error occurs while flooding a region, *graphresult* returns a value of –7.

See also

drawpoly, fillpoly, graphresult, setcolor, setfillstyle

getarccoords graphics.h

Function

Gets coordinates of the last call to *arc*.

Syntax

```
void far getarccoords(struct arccoordstype far *arccoords);
```

Remarks

getarccoords fills in the *arccoordstype* structure pointed to by *arccoords* with information about the last call to *arc*. The *arccoordstype* structure is defined in graphics.h as follows:

```
struct arccoordstype {
   int x, y;
   int xstart, ystart, xend, yend;
};
```

The members of this structure are used to specify the center point (*x*,*y*), the starting position (*xstart*, *ystart*), and the ending position (*xend*, *yend*) of the arc. They are useful if you need to make a line meet at the end of an arc.

Return value
None.

See also
arc, *fillellipse*, *sector*

getaspectratio — graphics.h

Function
Retrieves the current graphics mode's aspect ratio.

Syntax
```
void far getaspectratio(int far *xasp, int far *yasp);
```

Remarks
The *y* aspect factor, **yasp*, is normalized to 10,000. On all graphics adapters except the VGA, **xasp* (the *x* aspect factor) is less than **yasp* because the pixels are taller than they are wide. On the VGA, which has "square" pixels, **xasp* equals **yasp*. In general, the relationship between **yasp* and **xasp* can be stated as

$*yasp = 10{,}000$

$*xasp <= 10{,}000$

getaspectratio gets the values in **xasp* and **yasp*.

Return value
None.

See also
arc, *circle*, *ellipse*, *fillellipse*, *pieslice*, *sector*, *setaspectratio*

getbkcolor — graphics.h

Function
Returns the current background color.

Syntax
```
int far getbkcolor(void);
```

Remarks
getbkcolor returns the current background color. (See the table under *setbkcolor* for details.)

Return value
getbkcolor returns the current background color.

See also
getcolor, getmaxcolor, getpalette, setbkcolor

getcolor — graphics.h

Function
Returns the current drawing color.

Syntax
```
int far getcolor(void);
```

Remarks
getcolor returns the current drawing color.

The drawing color is the value to which pixels are set when lines and so on are drawn. For example, in CGAC0 mode, the palette contains four colors: the background color, light green, light red, and yellow. In this mode, if *getcolor* returns 1, the current drawing color is light green.

Return value
getcolor returns the current drawing color.

See also
getbkcolor, getmaxcolor, getpalette, setcolor

getdefaultpalette — graphics.h

Function
Returns the palette definition structure.

Syntax
```
struct palettetype *far getdefaultpalette(void);
```

Remarks
getdefaultpalette finds the *palettetype* structure that contains the palette initialized by the driver during *initgraph*.

Return value

getdefaultpalette returns a pointer to the default palette set up by the current driver when that driver was initialized.

See also

getpalette, initgraph

getdrivername graphics.h

Function

Returns a pointer to a string containing the name of the current graphics driver.

Syntax

```
char *far getdrivername(void);
```

Remarks

After a call to *initgraph*, *getdrivername* returns the name of the driver that is currently loaded.

Return value

getdrivername returns a pointer to a string with the name of the currently loaded graphics driver.

See also

initgraph

getfillpattern graphics.h

Function

Copies a user-defined fill pattern into memory.

Syntax

```
void far getfillpattern(char far *pattern);
```

Remarks

getfillpattern copies the user-defined fill pattern, as set by *setfillpattern*, into the 8-byte area pointed to by *pattern*.

pattern is a pointer to a sequence of 8 bytes, with each byte corresponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is set to 1, the corresponding pixel will be plotted. For example, the following user-defined fill pattern represents a checkerboard:

```
char checkboard[8] = { 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55 };
```

Return value

None.

See also

getfillsettings, setfillpattern

getfillsettings — graphics.h

Function

Gets information about current fill pattern and color.

Syntax

```
void far getfillsettings(struct fillsettingstype far *fillinfo);
```

Remarks

getfillsettings fills in the *fillsettingstype* structure pointed to by *fillinfo* with information about the current fill pattern and fill color. The *fillsettingstype* structure is defined in graphics.h as follows:

```
struct fillsettingstype {
   int pattern;          /* current fill pattern */
   int color;            /* current fill color */
};
```

The functions *bar, bar3d, fillpoly, floodfill,* and *pieslice* all fill an area with the current fill pattern in the current fill color. There are 11 predefined fill pattern styles (such as solid, crosshatch, dotted, and so on). Symbolic names for the predefined patterns are provided by the enumerated type *fill_patterns* in graphics.h (see the following table). In addition, you can define your own fill pattern.

If *pattern* equals 12 (USER_FILL), then a user-defined fill pattern is being used; otherwise, *pattern* gives the number of a predefined pattern.

The enumerated type *fill_patterns*, defined in graphics.h, gives names for the predefined fill patterns, plus an indicator for a user-defined pattern.

Name	Value	Description
EMPTY_FILL	0	Fill with background color
SOLID_FILL	1	Solid fill
LINE_FILL	2	Fill with——
LTSLASH_FILL	3	Fill with ///
SLASH_FILL	4	Fill with ///, thick lines
BKSLASH_FILL	5	Fill with \\\, thick lines
LTBKSLASH_FILL	6	Fill with \\\
HATCH_FILL	7	Light hatch fill
XHATCH_FILL	8	Heavy crosshatch fill
INTERLEAVE_FILL	9	Interleaving line fill

Name	Value	Description
WIDE_DOT_FILL	10	Widely spaced dot fill
CLOSE_DOT_FILL	11	Closely spaced dot fill
USER_FILL	12	User-defined fill pattern

All but EMPTY_FILL fill with the current fill color; EMPTY_FILL uses the current background color.

Return value

None.

See also

getfillpattern, setfillpattern, setfillstyle

getgraphmode — graphics.h

Function

Returns the current graphics mode.

Syntax

```
int far getgraphmode(void);
```

Remarks

Your program must make a successful call to *initgraph* before calling *getgraphmode.*

The enumeration *graphics_mode*, defined in graphics.h, gives names for the predefined graphics modes. For a table listing these enumeration values, refer to the description for *initgraph.*

Return value

getgraphmode returns the graphics mode set by *initgraph* or *setgraphmode.*

See also

getmoderange, restorecrtmode, setgraphmode

getimage — graphics.h

Function

Saves a bit image of the specified region into memory.

Syntax

```
void far getimage(int left, int top, int right, int bottom, void far *bitmap);
```

Remarks

getimage copies an image from the screen to memory.

left, *top*, *right*, and *bottom* define the screen area to which the rectangle is copied. *bitmap* points to the area in memory where the bit image is stored. The first two words of this area are used for the width and height of the rectangle; the remainder holds the image itself.

Return value

None.

See also

imagesize, *putimage*, *putpixel*

getlinesettings — graphics.h

Function

Gets the current line style, pattern, and thickness.

Syntax

```
void far getlinesettings(struct linesettingstype far *lineinfo);
```

Remarks

getlinesettings fills a *linesettingstype* structure pointed to by *lineinfo* with information about the current line style, pattern, and thickness.

The *linesettingstype* structure is defined in graphics.h as follows:

```
struct linesettingstype {
   int linestyle;
   unsigned upattern;
   int thickness;
};
```

linestyle specifies in which style subsequent lines will be drawn (such as solid, dotted, centered, dashed). The enumeration *line_styles*, defined in graphics.h, gives names to these operators:

Name	Value	Description
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User-defined line style

thickness specifies whether the width of subsequent lines drawn will be normal or thick.

Name	Value	Description
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

upattern is a 16-bit pattern that applies only if *linestyle* is USERBIT_LINE (4). In that case, whenever a bit in the pattern word is 1, the corresponding pixel in the line is drawn in the current drawing color. For example, a solid line corresponds to a *upattern* of 0xFFFF (all pixels drawn), while a dashed line can correspond to a *upattern* of 0x3333 or 0x0F0F. If the *linestyle* parameter to *setlinestyle* is not USERBIT_LINE (!=4), the *upattern* parameter must still be supplied but is ignored.

Return value

None.

See also

setlinestyle

getmaxcolor — graphics.h

Function

Returns maximum color value that can be passed to the *setcolor* function.

Syntax

```
int far getmaxcolor(void);
```

Remarks

getmaxcolor returns the highest valid color value for the current graphics driver and mode that can be passed to *setcolor*.

For example, on a 256K EGA, *getmaxcolor* always returns 15, which means that any call to *setcolor* with a value from 0 to 15 is valid. On a CGA in high-resolution mode or on a Hercules monochrome adapter, *getmaxcolor* returns a value of 1.

Return value

getmaxcolor returns the highest available color value.

See also

getbkcolor, getcolor, getpalette, getpalettesize, setcolor

getmaxmode graphics.h

Function
Returns the maximum mode number for the current driver.

Syntax
```
int far getmaxmode(void);
```

Remarks
getmaxmode lets you find out the maximum mode number for the currently loaded driver, directly from the driver. This gives it an advantage over *getmoderange*, which works for Borland drivers only. The minimum mode is 0.

Return value
getmaxmode returns the maximum mode number for the current driver.

See also
getmodename, *getmoderange*

getmaxx graphics.h

Function
Returns maximum *x* screen coordinate.

Syntax
```
int far getmaxx(void);
```

Remarks
getmaxx returns the maximum (screen-relative) *x* value for the current graphics driver and mode.

For example, on a CGA in 320×200 mode, *getmaxx* returns 319. *getmaxx* is invaluable for centering, determining the boundaries of a region onscreen, and so on.

Return value
getmaxx returns the maximum *x* screen coordinate.

See also
getmaxy, *getx*

getmaxy
graphics.h

Function
Returns maximum *y* screen coordinate.

Syntax

```
int far getmaxy(void);
```

Remarks
getmaxy returns the maximum (screen-relative) *y* value for the current graphics driver and mode.

For example, on a CGA in 320×200 mode, *getmaxy* returns 199. *getmaxy* is invaluable for centering, determining the boundaries of a region onscreen, and so on.

Return value
getmaxy returns the maximum *y* screen coordinate.

See also
getmaxx, getx, gety

getmodename
graphics.h

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

Syntax

```
char *far getmodename(int mode_number);
```

Remarks
getmodename accepts a graphics mode number as input and returns a string containing the name of the corresponding graphics mode. The mode names are embedded in each driver. The return values ("320×200 CGA P1," "640×200 CGA," and so on) are useful for building menus or displaying status.

Return value
getmodename returns a pointer to a string with the name of the graphics mode.

See also
getmaxmode, getmoderange

getmoderange

graphics.h

Function
Gets the range of modes for a given graphics driver.

Syntax

```
void far getmoderange(int graphdriver, int far *lomode, int far *himode);
```

Remarks
getmoderange gets the range of valid graphics modes for the given graphics driver, *graphdriver*. The lowest permissible mode value is returned in **lomode,* and the highest permissible value is **himode*. If *graphdriver* specifies an invalid graphics driver, both **lomode* and **himode* are set to –1. If the value of *graphdriver* is –1, the currently loaded driver modes are given.

Return value
None.

See also
getgraphmode, getmaxmode, getmodename, initgraph, setgraphmode

getpalette

graphics.h

Function
Gets information about the current palette.

Syntax

```
void far getpalette(struct palettetype far *palette);
```

Remarks
getpalette fills the *palettetype* structure pointed to by *palette* with information about the current palette's size and colors.

The MAXCOLORS constant and the *palettetype* structure used by *getpalette* are defined in graphics.h as follows:

```
#define MAXCOLORS  15

struct palettetype {
   unsigned char size;
   signed char colors[MAXCOLORS + 1];
};
```

size gives the number of colors in the palette for the current graphics driver in the current mode.

colors is an array of *size* bytes containing the actual raw color numbers for each entry in the palette.

Note *getpalette* cannot be used with the IBM-8514 driver.

Return value
None.

See also
getbkcolor, getcolor, getdefaultpalette, getmaxcolor, setallpalette, setpalette

getpalettesize — graphics.h

Function
Returns size of palette color lookup table.

Syntax
```
int far getpalettesize(void);
```

Remarks
getpalettesize is used to determine how many palette entries can be set for the current graphics mode. For example, the EGA in color mode returns 16.

Return value
getpalettesize returns the number of palette entries in the current palette.

See also
setpalette, setallpalette

getpixel — graphics.h

Function
Gets the color of a specified pixel.

Syntax
```
unsigned far getpixel(int x, int y);
```

Remarks
getpixel gets the color of the pixel located at (*x*,*y*).

Return value
getpixel returns the color of the given pixel.

See also
getimage, putpixel

gettextsettings — graphics.h

Function
Gets information about the current graphics text font.

Syntax

```
void far gettextsettings(struct textsettingstype far *texttypeinfo);
```

Remarks
gettextsettings fills the *textsettingstype* structure pointed to by *textinfo* with information about the current text font, direction, size, and justification.

The *textsettingstype* structure used by *gettextsettings* is defined in graphics.h as follows:

```
struct textsettingstype {
   int font;
   int direction;
   int charsize;
   int horiz;
   int vert;
};
```

See *settextstyle* for a description of these fields.

Return value
None.

See also
outtext, outtextxy, registerbgifont, settextjustify, settextstyle, setusercharsize, textheight, textwidth

getviewsettings — graphics.h

Function
Gets information about the current viewport.

Syntax

```
void far getviewsettings(struct viewporttype far *viewport);
```

Remarks
getviewsettings fills the *viewporttype* structure pointed to by *viewport* with information about the current viewport.

The *viewporttype* structure used by *getviewport* is defined in graphics.h as follows:

```
struct viewporttype {
   int left, top, right, bottom;
   int clip;
};
```

Return value
None.

See also
clearviewport, getx, gety, setviewport

getx — graphics.h

Function
Returns the current graphics position's x-coordinate.

Syntax

```
int far getx(void);
```

Remarks
getx finds the current graphics position's x-coordinate. The value is viewport-relative.

Return value
getx returns the x-coordinate of the current position.

See also
getmaxx, getmaxy, getviewsettings, gety, moveto

gety — graphics.h

Function
Returns the current graphics position's y-coordinate.

Syntax

```
int far gety(void);
```

Remarks
gety returns the current graphics position's y-coordinate. The value is viewport-relative.

Return value
gety returns the y-coordinate of the current position.

See also
getmaxx, getmaxy, getviewsettings, getx, moveto

graphdefaults — graphics.h

Function
Resets all graphics settings to their defaults.

Syntax
```
void far graphdefaults(void);
```

Remarks
graphdefaults resets all graphics settings to their defaults:

- Sets the viewport to the entire screen.
- Moves the current position to (0,0).
- Sets the default palette colors, background color, and drawing color.
- Sets the default fill style and pattern.
- Sets the default text font and justification.

Return value
None.

See also
initgraph, setgraphmode

grapherrormsg — graphics.h

Function
Returns a pointer to an error message string.

Syntax
```
char * far grapherrormsg(int errorcode);
```

Remarks
grapherrormsg returns a pointer to the error message string associated with *errorcode,* the value returned by *graphresult.*

Refer to the entry for *errno* in the *Library Reference* for a list of error messages and mnemonics.

Return value
grapherrormsg returns a pointer to an error message string.

See also
graphresult

_graphfreemem — graphics.h

Function
User hook into graphics memory deallocation.

Syntax
```
void far _graphfreemem(void far *ptr, unsigned size);
```

Remarks
The graphics library calls *_graphfreemem* to release memory previously allocated through *_graphgetmem*. You can choose to control the graphics library memory management by simply defining your own version of *_graphfreemem* (you must declare it exactly as shown in the declaration). The default version of this routine merely calls *free*.

Return value
None.

See also
_graphgetmem, *setgraphbufsize*

_graphgetmem — graphics.h

Function
User hook into graphics memory allocation.

Syntax
```
void far * far _graphgetmem(unsigned size);
```

Remarks
Routines in the graphics library (not the user program) normally call *_graphgetmem* to allocate memory for internal buffers, graphics drivers, and character sets. You can choose to control the memory management of the graphics library by defining your own version of *_graphgetmem* (you must declare it exactly as shown in the declaration). The default version of this routine merely calls *malloc*.

Return value
None.

See also
_graphfreemem, initgraph, setgraphbufsize

graphresult graphics.h

Function
Returns an error code for the last unsuccessful graphics operation.

Syntax
```
int far graphresult(void);
```

Remarks
graphresult returns the error code for the last graphics operation that reported an error and resets the error level to grOk.

The following table lists the error codes returned by *graphresult*. The enumerated type *graph_errors* defines the errors in this table. *graph_errors* is declared in graphics.h.

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

Note that the variable maintained by *graphresult* is reset to 0 after *graphresult* has been called. Therefore, you should store the value of *graphresult* into a temporary variable and then test it.

Return value

graphresult returns the current graphics error number, an integer in the range –15 to 0; *grapherrormsg* returns a pointer to a string associated with the value returned by *graphresult*.

See also

detectgraph, drawpoly, fillpoly, floodfill, grapherrormsg, initgraph, pieslice, registerbgidriver, registerbgifont, setallpalette, setcolor, setfillstyle, setgraphmode, setlinestyle, setpalette, settextjustify, settextstyle, setusercharsize, setviewport, setvisualpage

imagesize — graphics.h

Function

Returns the number of bytes required to store a bit image.

Syntax

```
unsigned far imagesize(int left, int top, int right, int bottom);
```

Remarks

imagesize determines the size of the memory area required to store a bit image. If the size required for the selected image is greater than or equal to 64K – 1 bytes, *imagesize* returns 0xFFFF (–1).

Return value

imagesize returns the size of the required memory area in bytes.

See also

getimage, putimage

initgraph — graphics.h

Function

Initializes the graphics system.

Syntax

```
void far initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);
```

Remarks

initgraph initializes the graphics system by loading a graphics driver from disk (or validating a registered driver), and putting the system into graphics mode.

To start the graphics system, first call the *initgraph* function. *initgraph* loads the graphics driver and puts the system into graphics mode. You can tell *initgraph* to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver.

If you tell *initgraph* to autodetect, it calls *detectgraph* to select a graphics driver and mode. *initgraph* also resets all graphics settings to their defaults (current position, palette, color, viewport, and so on) and resets *graphresult* to 0.

Normally, *initgraph* loads a graphics driver by allocating memory for the driver (through *_graphgetmem*), then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. See UTILS.TXT (included with your distribution disks) for more information on BGIOBJ.

pathtodriver specifies the directory path where *initgraph* looks for graphics drivers. *initgraph* first looks in the path specified in *pathtodriver*, then (if they're not there) in the current directory. Accordingly, if *pathtodriver* is null, the driver files (*.BGI) must be in the current directory. This is also the path *settextstyle* searches for the stroked character font files (*.CHR).

**graphdriver* is an integer that specifies the graphics driver to be used. You can give it a value using a constant of the *graphics_drivers* enumeration type, which is defined in graphics.h and listed in Table 4.3.

Table 4.3 Graphics drivers constants

graphics_drivers constant	Numeric value
DETECT	0 (requests autodetection)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

**graphmode* is an integer that specifies the initial graphics mode (unless **graphdriver* equals DETECT, in which case **graphmode* is set by *initgraph* to the highest resolution available for the detected driver). You can give **graphmode* a value using a constant of the *graphics_modes* enumeration type, which is defined in graphics.h and listed in Table 4.5.

Note *graphdriver* and *graphmode* must be set to valid values from Tables 4.3 and 4.5, or you'll get unpredictable results. The exception is *graphdriver* = DETECT.

In Table 4.5, the **Palette** listings C0, C1, C2, and C3 refer to the four predefined four-color palettes available on CGA (and compatible) systems. You can select the background color (entry #0) in each of these palettes, but the other colors are fixed.

These palettes are described in greater detail in Chapter 3, and summarized in Table 4.4.

Table 4.4 Color palettes

Palette number	Color assigned to pixel value		
	1	2	3
0	LIGHTGREEN	LIGHTRED	YELLOW
1	LIGHTCYAN	LIGHTMAGENTA	WHITE
2	GREEN	RED	BROWN
3	CYAN	MAGENTA	LIGHTGRAY

After a call to *initgraph*, **graphdriver* is set to the current graphics driver, and **graphmode* is set to the current graphics mode.

Table 4.5 Graphics modes

Graphics driver	*graphics_modes*	Value	_Column ×row	Palette	Pages
CGA	CGAC0	0	320×200	C0	1
	CGAC1	1	320×200	C1	1
	CGAC2	2	320×200	C2	1
	CGAC3	3	320×200	C3	1
	CGAHI	4	640×200	2 color	1
MCGA	MCGAC0	0	320×200	C0	1
	MCGAC1	1	320×200	C1	1
	MCGAC2	2	320×200	C2	1
	MCGAC3	3	320×200	C3	1
	MCGAMED	4	640×200	2 color	1
	MCGAHI	5	640×480	2 color	1
EGA	EGALO	0	640×200	16 color	4
	EGAHI	1	640×350	16 color	2
EGA64	EGA64LO	0	640×200	16 color	1
	EGA64HI	1	640×350	4 color	1
EGA-MONO	EGAMONOHI	3	640×350	2 color	1*
	EGAMONOHI	3	640×350	2 color	2**
HERC	HERCMONOHI	0	720×348	2 color	2
ATT400	ATT400C0	0	320×200	C0	1
	ATT400C1	1	320×200	C1	1
	ATT400C2	2	320×200	C2	1
	ATT400C3	3	320×200	C3	1
	ATT400MED	4	640×200	2 color	1
	ATT400HI	5	640×400	2 color	1
VGA	VGALO	0	640×200	16 color	2
	VGAMED	1	640×350	16 color	2
	VGAHI	2	640×480	16 color	1

Table 4.5 Graphics modes (continued)

Graphics driver	*graphics_modes*	Value	_Column ×row	Palette	Pages
PC3270	PC3270HI	0	720×350	2 color	1
IBM8514	IBM8514HI	1	1024×768	256 color	
	IBM8514LO	0	640×480	256 color	

* 64K on EGAMONO card
** 256K on EGAMONO card

Return value

initgraph always sets the internal error code; on success, it sets the code to 0. If an error occurred, **graphdriver* is set to –2, –3, –4, or –5, and *graphresult* returns the same value as listed here:

grNotDetected	–2	Cannot detect a graphics card
grFileNotFound	–3	Cannot find driver file
grInvalidDriver	–4	Invalid driver
grNoLoadMem	–5	Insufficient memory to load driver

See also

closegraph, detectgraph, getdefaultpalette, getdrivername, getgraphmode, getmoderange, graphdefaults, _graphgetmem, graphresult, installuserdriver, registerbgidriver, registerbgifont, restorecrtmode, setgraphbufsize, setgraphmode

installuserdriver — graphics.h

Function

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

Syntax

```
int far installuserdriver(char far *name, int huge (*detect)(void));
```

Remarks

installuserdriver lets you add a vendor-added device driver to the BGI internal table. The *name* parameter is the name of the new device-driver file (.BGI), and the *detect* parameter is a pointer to an optional autodetect function that can accompany the new driver. This autodetect function takes no parameters and returns an integer value.

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

The other, more general way to use this driver is to link in an autodetect function that will be called by *initgraph* as part of its hardware-detection logic (presumably, the manufacturer of the SGA gave you this autodetect function). When you install the driver (by calling *installuserdriver*), you pass the address of this function, along with the device driver's file name.

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

You can install up to ten drivers at one time.

Return value

The value returned by *installuserdriver* is the driver number parameter you would pass to *initgraph* in order to select the newly installed driver manually.

See also

initgraph, registerbgidriver

installuserfont — graphics.h

Function

Loads a font file (.CHR) that is not built into the BGI system.

Syntax

```
int far installuserfont(char far *name);
```

Remarks

name is a filename in the current directory (pathname is not supported) of a font file containing a stroked font. Up to twenty fonts can be installed at one time.

Return value

installuserfont returns a font ID number that can then be passed to *settextstyle* to select the corresponding font. If the internal font table is full, a value of –11 (grError) is returned.

See also

settextstyle

line

graphics.h

Function

Draws a line between two specified points.

Syntax

```
void far line(int x1, int y1, int x2, int y2);
```

Remarks

line draws a line in the current color, using the current line style and thickness between the two points specified, (*x1*,*y1*) and (*x2*,*y2*), without updating the current position (CP).

Return value

None.

See also

getlinesettings, linerel, lineto, setcolor, setlinestyle, setwritemode

linerel

graphics.h

Function

Draws a line a relative distance from the current position (CP).

Syntax

```
void far linerel(int dx, int dy);
```

Remarks

linerel draws a line from the CP to a point that is a relative distance (*dx*,*dy*) from the CP. The CP is advanced by (*dx*,*dy*).

Return value

None.

See also

getlinesettings, line, lineto, setcolor, setlinestyle, setwritemode

lineto

graphics.h

Function

Draws a line from the current position (CP) to (*x*,*y*).

Syntax

```
void far lineto(int x, int y);
```

Remarks

lineto draws a line from the CP to (*x*,*y*), then moves the CP to (*x*,*y*).

Return value

None.

See also

getlinesettings, line, linerel, setcolor, setlinestyle, setvisualpage, setwritemode

moverel graphics.h

Function

Moves the current position (CP) a relative distance.

Syntax

```
void far moverel(int dx, int dy);
```

Remarks

moverel moves the current position (CP) *dx* pixels in the *x* direction and *dy* pixels in the *y* direction.

Return value

None.

See also

moveto

moveto graphics.h

Function

Moves the current position (CP) to (*x*,*y*).

Syntax

```
void far moveto(int x, int y);
```

Remarks

moveto moves the current position (CP) to viewport position (*x*,*y*).

Return value
None.

See also
moverel

outtext graphics.h

Function
Displays a string in the viewport.

Syntax
```
void far outtext(char far *textstring);
```

Remarks
outtext displays a text string in the viewport, using the current font, direction, and size.

outtext outputs *textstring* at the current position (CP). If the horizontal text justification is LEFT_TEXT and the text direction is HORIZ_DIR, the CP's x-coordinate is advanced by *textwidth(textstring)*. Otherwise, the CP remains unchanged.

To maintain code compatibility when using several fonts, use *textwidth* and *textheight* to determine the dimensions of the string.

Note If a string is printed with the default font using *outtext*, any part of the string that extends outside the current viewport is truncated.

outtext is for use in graphics mode; it will not work in text mode.

Return value
None.

See also
gettextsettings, outtextxy, settextjustify, textheight, textwidth

outtextxy graphics.h

Function
Displays a string at a specified location.

Syntax
```
void far outtextxy(int x, int y, char far *textstring);
```

Remarks

outtextxy displays a text string in the viewport at the given position (*x*, *y*), using the current justification settings and the current font, direction, and size.

To maintain code compatibility when using several fonts, use *textwidth* and *textheight* to determine the dimensions of the string.

Note If a string is printed with the default font using *outtext* or *outtextxy*, any part of the string that extends outside the current viewport is truncated.

outtextxy is for use in graphics mode; it will not work in text mode.

Return value

None.

See also

gettextsettings, *outtext*, *textheight*, *textwidth*

pieslice — graphics.h

Function

Draws and fills in pie slice.

Syntax

```
void far pieslice(int x, int y, int stangle, int endangle, int radius);
```

Remarks

pieslice draws and fills a pie slice centered at (*x*,*y*) with a radius given by *radius*. The slice travels from *stangle* to *endangle*. The slice is outlined in the current drawing color and then filled using the current fill pattern and fill color.

The angles for *pieslice* are given in degrees. They are measured counterclockwise, with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on.

If you're using a CGA or monochrome adapter, the examples in online Help that show how to use graphics functions might not produce the expected results. If your system runs on a CGA or monochrome adapter, use the value 1 (one) instead of the symbolic color constant, and consult the second online Help example under *arc* on how to use the *pieslice* function.

Return value

None.

See also

fillellipse, *fill_patterns* (enumerated type), *graphresult*, *sector*, *setfillstyle*

putimage

graphics.h

Function

Outputs a bit image to screen.

Syntax

```
void far putimage(int left, int top, void far *bitmap, int op);
```

Remarks

putimage puts the bit image previously saved with *getimage* back onto the screen, with the upper left corner of the image placed at (*left,top*). *bitmap* points to the area in memory where the source image is stored.

The *op* parameter to *putimage* specifies a combination operator that controls how the color for each destination pixel onscreen is computed, based on the pixel already onscreen and the corresponding source pixel in memory.

The enumeration *putimage_ops*, as defined in graphics.h, gives names to these operators.

Name	Value	Description
COPY_PUT	0	Copy
XOR_PUT	1	Exclusive or
OR_PUT	2	Inclusive or
AND_PUT	3	And
NOT_PUT	4	Copy the inverse of the source

In other words, COPY_PUT copies the source bitmap image onto the screen, XOR_PUT XORs the source image with the image already onscreen, OR_PUT ORs the source image with that onscreen, and so on.

Return value

None.

See also

getimage, imagesize, putpixel, setvisualpage

putpixel

graphics.h

Function

Plots a pixel at a specified point.

Syntax

```
void far putpixel(int x, int y, int color);
```

Remarks
putpixel plots a point in the color defined by *color* at (*x*,*y*).

Return value
None.

See also
getpixel, *putimage*

rectangle — graphics.h

Function
Draws a rectangle.

Syntax
```
void far rectangle(int left, int top, int right, int bottom);
```

Remarks
rectangle draws a rectangle in the current line style, thickness, and drawing color.

(*left*,*top*) is the upper left corner of the rectangle, and (*right*,*bottom*) is its lower right corner.

Return value
None.

See also
bar, *bar3d*, *setcolor*, *setlinestyle*

registerbgifont — graphics.h

Function
Registers linked-in stroked font code.

Syntax
```
int registerbgifont(void (*font)(void));
```

Remarks
Calling *registerbgifont* informs the graphics system that the font pointed to by *font* was included at link time. This routine checks the linked-in code for the specified font; if the code is valid, it registers the code in internal tables. Linked-in fonts are discussed in detail under BGIOBJ in UTILS.TXT included with your distribution disks.

By using the name of a linked-in font in a call to *registerbgifont*, you also tell the compiler (and linker) to link in the object file with that public name.

If you register a user-supplied font, you *must* pass the result of *registerbgifont* to *settextstyle* as the font number to be used.

Return value

registerbgifont returns a negative graphics error code if the specified font is invalid. Otherwise, *registerbgifont* returns the font number of the registered font.

See also

graphresult, initgraph, installuserdriver, registerbgidriver, settextstyle

registerbgidriver — graphics.h

Function

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

Syntax

```
int registerbgidriver(void (*driver)(void));
```

Remarks

registerbgidriver enables a user to load a driver file and "register" the driver. Once its memory location has been passed to *registerbgidriver*, *initgraph* uses the registered driver. A user-registered driver can be loaded from disk onto the heap, or converted to an .OBJ file (using BGIOBJ.EXE) and linked into the .EXE.

Calling *registerbgidriver* informs the graphics system that the driver pointed to by *driver* was included at link time. This routine checks the linked-in code for the specified driver; if the code is valid, it registers the code in internal tables. Linked-in drivers are discussed in detail in UTILS.TXT, included with your distribution disks.

By using the name of a linked-in driver in a call to *registerbgidriver*, you also tell the compiler (and linker) to link in the object file with that public name.

Return value

registerbgidriver returns a negative graphics error code if the specified driver or font is invalid. Otherwise, *registerbgidriver* returns the driver number.

If you register a user-supplied driver, you *must* pass the result of *registerbgidriver* to *initgraph* as the driver number to be used.

See also

graphresult, initgraph, installuserdriver, registerbgifont

restorecrtmode graphics.h

Function

Restores the screen mode to its pre-*initgraph* setting.

Syntax

```
void far restorecrtmode(void);
```

Remarks

restorecrtmode restores the original video mode detected by *initgraph*.

This function can be used in conjunction with *setgraphmode* to switch back and forth between text and graphics modes. *textmode* should not be used for this purpose; use it only when the screen is in text mode, to change to a different text mode.

Return value

None.

See also

getgraphmode, initgraph, setgraphmode

sector graphics.h

Function

Draws and fills an elliptical pie slice.

Syntax

```
void far sector(int x, int y, int stangle, int endangle, int xradius, int yradius);
```

Remarks

Draws and fills an elliptical pie slice using (*x,y*) as the center point, *xradius* and *yradius* as the horizontal and vertical radii, respectively, and drawing from *stangle* to *endangle*. The pie slice is outlined using the current color, and filled using the pattern and color defined by *setfillstyle* or *setfillpattern*.

The angles for *sector* are given in degrees. They are measured counterclockwise with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, and so on.

If an error occurs while the pie slice is filling, *graphresult* returns a value of –6 (grNoScanMem).

Return value

None.

See also
arc, circle, ellipse, getarccoords, getaspectratio, graphresult, pieslice, setfillpattern, setfillstyle, setgraphbufsize

setactivepage **graphics.h**

Function
Sets active page for graphics output.

Syntax
```
void far setactivepage(int page);
```

Remarks
setactivepage makes *page* the active graphics page. All subsequent graphics output will be directed to that graphics page.

The active graphics page might not be the one you see onscreen, depending on how many graphics pages are available on your system. Only the EGA, VGA, and Hercules graphics cards support multiple pages.

Return value
None.

See also
setvisualpage

setallpalette **graphics.h**

Function
Changes all palette colors as specified.

Syntax
```
void far setallpalette(struct palettetype far *palette);
```

Remarks
setallpalette sets the current palette to the values given in the *palettetype* structure pointed to by *palette*.

You can partially (or completely) change the colors in the EGA/VGA palette with *setallpalette*.

The MAXCOLORS constant and the *palettetype* structure used by *setallpalette* are defined in graphics.h as follows:

```
#define MAXCOLORS  15
```

```
struct palettetype {
   unsigned char size;
   signed char colors[MAXCOLORS + 1];
};
```

size gives the number of colors in the palette for the current graphics driver in the current mode.

colors is an array of *size* bytes containing the actual raw color numbers for each entry in the palette. If an element of *colors* is –1, the palette color for that entry is not changed.

The elements in the *colors* array used by *setallpalette* can be represented by symbolic constants which are defined in graphics.h.

Table 4.6 Actual color table

CGA		EGA/VGA	
Name	**Value**	**Name**	**Value**
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_LIGHTGRAY	7
LIGHTGRAY	7	EGA_BROWN	20
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Note that valid colors depend on the current graphics driver and current graphics mode.

Changes made to the palette are seen immediately onscreen. Each time a palette color is changed, all occurrences of that color onscreen will change to the new color value.

Note *setallpalette* cannot be used with the IBM-8514 driver.

Return value

If invalid input is passed to *setallpalette*, *graphresult* returns –11 (grError), and the current palette remains unchanged.

See also
getpalette, getpalettesize, graphresult, setbkcolor, setcolor, setpalette

setaspectratio — graphics.h

Function
Changes the default aspect ratio correction factor.

Syntax
```
void far setaspectratio(int xasp, int yasp);
```

Remarks
setaspectratio changes the default aspect ratio of the graphics system. The graphics system uses the aspect ratio to make sure that circles are round onscreen. If circles appear elliptical, the monitor is not aligned properly. You could correct this in the hardware by realigning the monitor, but it's easier to change in the software by using *setaspectratio* to set the aspect ratio. To obtain the current aspect ratio from the system, call *getaspectratio*.

Return value
None.

See also
circle, getaspectratio

setbkcolor — graphics.h

Function
Sets the current background color using the palette.

Syntax
```
void far setbkcolor(int color);
```

Remarks
setbkcolor sets the background to the color specified by *color*. The argument *color* can be a name or a number, as listed in the following table:

Number	Name	Number	Name
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN

Number	Name	Number	Name
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

Note These symbolic names are which are defined in graphics.h.

For example, if you want to set the background color to blue, you can call

```
ASPROGRAMC setbkcolor(BLUE) /* or */ setbkcolor(1)
```

On CGA and EGA systems, *setbkcolor* changes the background color by changing the first entry in the palette.

Note If you use an EGA or a VGA, and you change the palette colors with *setpalette* or *setallpalette*, the defined symbolic constants might not give you the correct color. This is because the parameter to *setbkcolor* indicates the entry number in the current palette rather than a specific color (unless the parameter passed is 0, which always sets the background color to black).

Return value

None.

See also

getbkcolor, *setallpalette*, *setcolor*, *setpalette*

setcolor — graphics.h

Function

Sets the current drawing color using the palette.

Syntax

```
void far setcolor(int color);
```

Remarks

setcolor sets the current drawing color to *color*, which can range from 0 to *getmaxcolor*.

The current drawing color is the value to which pixels are set when lines, and so on are drawn. The following tables show the drawing colors available for the CGA and EGA, respectively.

Palette number	Constant assigned to color number (pixel value)		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

Number	Name	Number	Name
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

You select a drawing color by passing either the color number itself or the equivalent symbolic name to *setcolor*. For example, in CGAC0 mode, the palette contains four colors: the background color, light green, light red, and yellow. In this mode, either *setcolor*(3) or *setcolor*(CGA_YELLOW) selects a drawing color of yellow.

Return value

None.

See also

getcolor, getmaxcolor, graphresult, setallpalette, setbkcolor, setpalette

setfillpattern — graphics.h

Function

Selects a user-defined fill pattern.

Syntax

```
void far setfillpattern(char far *upattern, int color);
```

Remarks

setfillpattern is like *setfillstyle*, except that you use it to set a user-defined 8×8 pattern rather than a predefined pattern.

upattern is a pointer to a sequence of 8 bytes, with each byte corresponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is set to 1, the corresponding pixel is plotted.

Return value
None.

See also
getfillpattern, *getfillsettings*, *graphresult*, *sector*, *setfillstyle*

setfillstyle — graphics.h

Function
Sets the fill pattern and color.

Syntax
```
void far setfillstyle(int pattern, int color);
```

Remarks
setfillstyle sets the current fill pattern and fill color. To set a user-defined fill pattern, do *not* give a pattern of 12 (USER_FILL) to *setfillstyle*; instead, call *setfillpattern*.

The enumeration *fill_patterns*, which is defined in graphics.h, gives names for the predefined fill patterns and an indicator for a user-defined pattern.

Name	Value	Description
EMPTY_FILL	0	Fill with background color
SOLID_FILL	1	Solid fill
LINE_FILL	2	Fill with ——
LTSLASH_FILL	3	Fill with ///
SLASH_FILL	4	Fill with ///, thick lines
BKSLASH_FILL	5	Fill with \\\, thick lines
LTBKSLASH_FILL	6	Fill with \\\
HATCH_FILL	7	Light hatch fill
XHATCH_FILL	8	Heavy crosshatch fill
INTERLEAVE_FILL	9	Interleaving line fill
WIDE_DOT_FILL	10	Widely spaced dot fill
CLOSE_DOT_FILL	11	Closely spaced dot fill
USER_FILL	12	User-defined fill pattern

All but EMPTY_FILL fill with the current fill color; EMPTY_FILL use the current background color.

If invalid input is passed to *setfillstyle*, *graphresult* returns –11 (grError), and the current fill pattern and fill color remain unchanged.

Return value
None.

See also
bar, bar3d, fillpoly, floodfill, getfillsettings, graphresult, pieslice, sector, setfillpattern

setgraphmode — graphics.h

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

Syntax
```
void far setgraphmode(int mode);
```

Remarks
setgraphmode selects a graphics mode different than the default one set by *initgraph*. *mode* must be a valid mode for the current device driver. *setgraphmode* clears the screen and resets all graphics settings to their defaults (current position, palette, color, viewport, and so on).

You can use *setgraphmode* in conjunction with *restorecrtmode* to switch back and forth between text and graphics modes.

Return value
If you give *setgraphmode* an invalid mode for the current device driver, *graphresult* returns a value of –10 (grInvalidMode).

See also
getgraphmode, getmoderange, graphresult, initgraph, restorecrtmode

setgraphbufsize — graphics.h

Function
Changes the size of the internal graphics buffer.

Syntax
```
unsigned far setgraphbufsize(unsigned bufsize);
```

Remarks
Some of the graphics routines (such as *floodfill*) use a memory buffer that is allocated when *initgraph* is called and released when *closegraph* is called. The default size of this buffer, allocated by *_graphgetmem*, is 4,096 bytes.

You might want to make this buffer smaller (to save memory space) or bigger (if, for example, a call to *floodfill* produces error –7: Out of flood memory).

setgraphbufsize tells *initgraph* how much memory to allocate for this internal graphics buffer when it calls *_graphgetmem*.

Note You *must* call *setgraphbufsize* before calling *initgraph*. Once *initgraph* has been called, all calls to *setgraphbufsize* are ignored until after the next call to *closegraph*.

Return value

setgraphbufsize returns the previous size of the internal buffer.

See also

closegraph, _graphfreemem, _graphgetmem, initgraph, sector

setlinestyle — graphics.h

Function

Sets the current line width and style.

Syntax

```
void far setlinestyle(int linestyle, unsigned upattern, int thickness);
```

Remarks

setlinestyle sets the style for all lines drawn by *line, lineto, rectangle, drawpoly,* and so on.

The *linesettingstype* structure is defined in graphics.h as follows:

```
struct linesettingstype {
   int linestyle;
   unsigned upattern;
   int thickness;
};
```

linestyle specifies in which of several styles subsequent lines will be drawn (such as solid, dotted, centered, dashed). The enumeration *line_styles*, which is defined in graphics.h, gives names to these operators:

Name	Value	Description
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User-defined line style

thickness specifies whether the width of subsequent lines drawn will be normal or thick.

Name	Value	Description
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

upattern is a 16-bit pattern that applies only if *linestyle* is USERBIT_LINE (4). In that case, whenever a bit in the pattern word is 1, the corresponding pixel in the line is drawn in the current drawing color. For example, a solid line corresponds to a *upattern* of 0xFFFF (all pixels drawn), and a dashed line can correspond to a *upattern* of 0x3333 or 0x0F0F. If the *linestyle* parameter to setlinestyle is not USERBIT_LINE (in other words, if it is not equal to 4), you must still provide the *upattern* parameter, but it will be ignored.

Note The *linestyle* parameter does not affect arcs, circles, ellipses, or pie slices. Only the *thickness* parameter is used.

Return value

If invalid input is passed to *setlinestyle*, *graphresult* returns –11, and the current line style remains unchanged.

See also

arc, bar3d, circle, drawpoly, ellipse, getlinesettings, graphresult, line, linerel, lineto, pieslice, rectangle

setpalette — graphics.h

Function

Changes one palette color.

Syntax

```
void far setpalette(int colornum, int color);
```

Remarks

setpalette changes the *colornum* entry in the palette to *color*. For example, *setpalette*(0,5) changes the first color in the current palette (the background color) to actual color number 5. If *size* is the number of entries in the current palette, *colornum* can range between 0 and (*size* –1).

You can partially (or completely) change the colors in the EGA/VGA palette with *setpalette*. On a CGA, you can only change the first entry in the palette (*colornum* equals 0, the background color) with a call to *setpalette*.

The *color* parameter passed to *setpalette* can be represented by symbolic constants which are defined in graphics.h.

CGA		EGA/VGA	
Name	**Value**	**Name**	**Value**
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_LIGHTGRAY	7
LIGHTGRAY	7	EGA_BROWN	20
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Note that valid colors depend on the current graphics driver and current graphics mode.

Changes made to the palette are seen immediately onscreen. Each time a palette color is changed, all occurrences of that color onscreen change to the new color value.

Note *setpalette* cannot be used with the IBM-8514 driver; use *setrgbpalette* instead.

Return value

If invalid input is passed to *setpalette*, *graphresult* returns –11, and the current palette remains unchanged.

See also

getpalette, *graphresult*, *setallpalette*, *setbkcolor*, *setcolor*, *setrgbpalette*

setrgbpalette — graphics.h

Function

Lets user define colors for the IBM 8514.

Syntax

```
void far setrgbpalette(int colornum, int red, int green, int blue);
```

Remarks

setrgbpalette can be used with the IBM 8514 and VGA drivers.

colornum defines the palette entry to be loaded, while *red*, *green*, and *blue* define the component colors of the palette entry.

For the IBM 8514 display (and the VGA in 256K color mode), *colornum* is in the range 0 to 255. For the remaining modes of the VGA, *colornum* is in the range 0 to 15. Only the lower byte of *red*, *green*, or *blue* is used, and out of each byte, only the 6 most significant bits are loaded in the palette.

Note For compatibility with other IBM graphics adapters, the BGI driver defines the first 16 palette entries of the IBM 8514 to the default colors of the EGA/VGA. These values can be used as is, or they can be changed using *setrgbpalette*.

Return value

None.

See also

setpalette

settextjustify — graphics.h

Function

Sets text justification for graphics functions.

Syntax

```
void far settextjustify(int horiz, int vert);
```

Remarks

Text output after a call to *settextjustify* is justified around the current position (CP) horizontally and vertically, as specified. The default justification settings are LEFT_TEXT (for horizontal) and TOP_TEXT (for vertical). The enumeration *text_just* in graphics.h provides names for the *horiz* and *vert* settings passed to *settextjustify*.

Description	Name	Value	Action
horiz	LEFT_TEXT	0	Left-justify text
	CENTER_TEXT	1	Center text
	RIGHT_TEXT	2	Right-justify text
vert	BOTTOM_TEXT	0	Justify from bottom
	CENTER_TEXT	1	Center text
	TOP_TEXT	2	Justify from top

If *horiz* is equal to LEFT_TEXT and direction equals HORIZ_DIR, the CP's x component is advanced after a call to *outtext(string)* by *textwidth(string)*.

settextjustify affects text written with *outtext* and cannot be used with text mode and stream functions.

Return value

If invalid input is passed to *settextjustify*, *graphresult* returns –11, and the current text justification remains unchanged.

See also

gettextsettings, *graphresult*, *outtext*, *settextstyle*

settextstyle — graphics.h

Function

Sets the current text characteristics for graphics output.

Syntax

```
void far settextstyle(int font, int direction, int charsize);
```

Remarks

settextstyle sets the text font, the direction in which text is displayed, and the size of the characters. A call to *settextstyle* affects all text output by *outtext* and *outtextxy*.

The parameters *font*, *direction*, and *charsize* passed to *settextstyle* are described in the following:

font: One 8×8 bit-mapped font and several "stroked" fonts are available. The 8×8 bit-mapped font is the default. The enumeration *font_names*, which is defined in graphics.h, provides names for these different font settings:

Name	Value	Description
DEFAULT_FONT	0	8×8 bit-mapped font
TRIPLEX_FONT	1	Stroked triplex font
SMALL_FONT	2	Stroked small font
SANS_SERIF_FONT	3	Stroked sans-serif font
GOTHIC_FONT	4	Stroked gothic font
SCRIPT_FONT	5	Stroked script font
SIMPLEX_FONT	6	Stroked triplex script font
TRIPLEX_SCR_FONT	7	Stroked triplex script font
COMPLEX_FONT	8	Stroked complex font
EUROPEAN_FONT	9	Stroked European font
BOLD_FONT	10	Stroked bold font

The default bit-mapped font is built into the graphics system. Stroked fonts are stored in *.CHR disk files, and only one at a time is kept in memory. Therefore, when you select a stroked font (different from the last selected stroked font), the corresponding *.CHR file must be loaded from disk.

To avoid this loading when several stroked fonts are used, you can link font files into your program. Do this by converting them into object files with the BGIOBJ utility, then registering them through *registerbgifont*, as described in UTILS.TXT, included with your distributions disks.

direction: Font directions supported are horizontal text (left to right) and vertical text (rotated 90 degrees counterclockwise). The default direction is HORIZ_DIR.

Name	Value	Description
HORIZ_DIR	0	Left to right
VERT_DIR	1	Bottom to top

charsize: The size of each character can be magnified using the *charsize* factor. If *charsize* is nonzero, it can affect bit-mapped or stroked characters. A *charsize* value of 0 can be used only with stroked fonts.

- If *charsize* equals 1, *outtext* and *outtextxy* displays characters from the 8×8 bit-mapped font in an 8×8 pixel rectangle onscreen.
- If *charsize* equals 2, these output functions display characters from the 8×8 bit-mapped font in a 16×16 pixel rectangle, and so on (up to a limit of ten times the normal size).
- When *charsize* equals 0, the output functions *outtext* and *outtextxy* magnify the stroked font text using either the default character magnification factor (4) or the user-defined character size given by *setusercharsize*.

Always use *textheight* and *textwidth* to determine the actual dimensions of the text.

Return value

None.

See also

gettextsettings, *graphresult*, *installuserfont*, *settextjustify*, *setusercharsize*, *textheight*, *textwidth*

setusercharsize — graphics.h

Function

Varies character width and height for stroked fonts.

Syntax

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Remarks

setusercharsize gives you finer control over the size of text from stroked fonts used with graphics functions. The values set by *setusercharsize* are active *only* if *charsize* equals 0, as set by a previous call to *settextstyle*.

With *setusercharsize*, you specify factors by which the width and height are scaled. The default width is scaled by *multx* : *divx*, and the default height is scaled by *multy* : *divy*. For example, to make text twice as wide and 50% taller than the default, set

```
multx = 2;  divx = 1;
multy = 3;  divy = 2;
```

Return value

None.

See also

gettextsettings, *graphresult*, *settextstyle*

setviewport — graphics.h

Function

Sets the current viewport for graphics output.

Syntax

```
void far setviewport(int left, int top, int right, int bottom, int clip);
```

Remarks

setviewport establishes a new viewport for graphics output.

The viewport's corners are given in absolute screen coordinates by (*left*,*top*) and (*right*,*bottom*). The current position (CP) is moved to (0,0) in the new window.

The parameter *clip* determines whether drawings are clipped (truncated) at the current viewport boundaries. If *clip* is nonzero, all drawings will be clipped to the current viewport.

Return value

If invalid input is passed to *setviewport*, *graphresult* returns –11, and the current view settings remain unchanged.

See also

clearviewport, *getviewsettings*, *graphresult*

setvisualpage **graphics.h**

Function
Sets the visual graphics page number.

Syntax
```
void far setvisualpage(int page);
```

Remarks
setvisualpage makes *page* the visual graphics page.

Return value
None.

See also
graphresult, setactivepage

setwritemode **graphics.h**

Function
Sets the writing mode for line drawing in graphics mode.

Syntax
```
void far setwritemode(int mode);
```

Remarks
The following constants are defined:

```
COPY_PUT = 0      /* MOV */
XOR_PUT  = 1      /* XOR */
```

Each constant corresponds to a binary operation between each byte in the line and the corresponding bytes onscreen. COPY_PUT uses the assembly language **MOV** instruction, overwriting with the line whatever is on the screen. XOR_PUT uses the **XOR** command to combine the line with the screen. Two successive **XOR** commands will erase the line and restore the screen to its original appearance.

Note *setwritemode* currently works only with *line, linerel, lineto, rectangle,* and *drawpoly.*

Return value
None.

See also
drawpoly, line, linerel, lineto, putimage

textheight

graphics.h

Function

Returns the height of a string in pixels.

Syntax

```
int far textheight(char far *textstring);
```

Remarks

The graphics function *textheight* takes the current font size and multiplication factor, and determines the height of *textstring* in pixels. This function is useful for adjusting the spacing between lines, computing viewport heights, sizing a title to make it fit on a graph or in a box, and so on.

For example, with the 8×8 bit-mapped font and a multiplication factor of 1 (set by *settextstyle*), the string *TurboC*++ is 8 pixels high.

Note Use *textheight* to compute the height of strings, instead of doing the computations manually. By using this function, no source code modifications have to be made when different fonts are selected.

Return value

textheight returns the text height in pixels.

See also

gettextsettings, *outtext*, *outtextxy*, *settextstyle*, *textwidth*

textwidth

graphics.h

Function

Returns the width of a string in pixels.

Syntax

```
int far textwidth(char far *textstring);
```

Remarks

The graphics function *textwidth* takes the string length, current font size, and multiplication factor, and determines the width of *textstring* in pixels.

This function is useful for computing viewport widths, sizing a title to make it fit on a graph or in a box, and so on.

Note Use *textwidth* to compute the width of strings, instead of doing the computations manually. When you use this function, no source code modifications have to be made when different fonts are selected.

Return value

textwidth returns the text width in pixels.

See also

gettextsettings, outtext, outtextxy, settextstyle, textheight

Chapter

5

DOS-only functions

Except for the functions *brk* and *sbrk* (which are available on DOS and UNIX), the functions described in this chapter are available only for 16-bit DOS applications. The *Library Reference*, Chapter 3, describes additional functions; some of those functions can also be used in 16-bit DOS applications. The descriptions of some of the functions listed in the **See also** entries of this chapter can be found in Chapter 3 of the *Library Reference*.

absread — dos.h

Function

Reads absolute disk sectors.

Syntax

```
int absread(int drive, int nsects, long lsect, void *buffer);
```

Remarks

absread reads specific disk sectors. It ignores the logical structure of a disk and pays no attention to files, FATs, or directories.

absread uses DOS interrupt 0x25 to read specific disk sectors.

drive	=	drive number to read (0 = A, 1 = B, etc.)
nsects	=	number of sectors to read
lsect	=	beginning logical sector number
buffer	=	memory address where the data is to be read

The number of sectors to read is limited to 64K or the size of the buffer, whichever is smaller.

Return value

If it is successful, *absread* returns 0.

On error, the routine returns –1 and sets the global variable *errno* to the value returned by the system call in the AX register.

See also

abswrite, biosdisk

abswrite dos.h

Function

Writes absolute disk sectors.

Syntax

```
int abswrite(int drive, int nsects, long lsect, void *buffer);
```

Remarks

abswrite writes specific disk sectors. It ignores the logical structure of a disk and pays no attention to files, FATs, or directories.

Note If used improperly, *abswrite* can overwrite files, directories, and FATs.

abswrite uses DOS interrupt 0x26 to write specific disk sectors.

drive	=	drive number to write to (0 = A, 1 = B, etc.)
nsects	=	number of sectors to write to
lsect	=	beginning logical sector number
buffer	=	memory address where the data is to be written

The number of sectors to write to is limited to 64K or the size of the buffer, whichever is smaller.

Return value

If it is successful, *abswrite* returns 0.

On error, the routine returns –1 and sets the global variable *errno* to the value of the AX register returned by the system call.

See also

absread, biosdisk

allocmem, _dos_allocmem dos.h

Function

Allocates DOS memory segment.

Syntax

```
int allocmem(unsigned size, unsigned *segp);
```

```
unsigned _dos_allocmem(unsigned size, unsigned *segp);
```

Remarks

allocmem and *_dos_allocmem* use the DOS system call 0x48 to allocate a block of free memory and return the segment address of the allocated block.

size is the desired size in paragraphs (a paragraph is 16 bytes). *segp* is a pointer to a word that will be assigned the segment address of the newly allocated block.

For *allocmem*, if not enough room is available, no assignment is made to the word pointed to by *segp*.

For *_dos_allocmem*, if not enough room is available, the size of the largest available block will be stored in the word pointed to by *segp*.

All allocated blocks are paragraph-aligned.

Note *allocmem* and *_dos_allocmem* cannot coexist with *malloc*.

Return value

allocmem returns –1 on success. In the event of error, *allocmem* returns a number indicating the size in paragraphs of the largest available block.

_dos_allocmem returns 0 on success. In the event of error, *_dos_allocmem* returns the DOS error code and sets the word pointed to by *segp* to the size in paragraphs of the largest available block.

An error return from *allocmem* or *_dos_allocmem* sets the global variable *_doserrno* and sets the global variable *errno* to the following:

ENOMEM	Not enough memory

See also

coreleft, *freemem*, *malloc*, *setblock*

bioscom — bios.h

Function

Performs serial I/O.

Syntax

```
int bioscom(int cmd, char abyte, int port);
```

Remarks

bioscom performs various RS-232 communications over the I/O port given in *port*.

A *port* value of 0 corresponds to COM1, 1 corresponds to COM2, and so forth.

The value of *cmd* can be one of the following:

0	Sets the communications parameters to the value in *abyte*.
1	Sends the character in *abyte* out over the communications line.
2	Receives a character from the communications line.
3	Returns the current status of the communications port.

abyte is a combination of the following bits (one value is selected from each of the groups):

0x02	7 data bits	0x00	110 baud
0x03	8 data bits	0x20	150 baud
		0x40	300 baud
0x00	1 stop bit	0x60	600 baud
0x04	2 stop bits	0x80	1200 baud
0x00	No parity	0xA0	2400 baud
0x08	Odd parity	0xC0	4800 baud
0x18	Even parity	0xE0	9600 baud

For example, a value of 0xEB (0xE0 | 0x08 | 0x00 | 0x03) for *abyte* sets the communications port to 9600 baud, odd parity, 1 stop bit, and 8 data bits. *bioscom* uses the BIOS 0x14 interrupt.

Return value

For all values of *cmd*, *bioscom* returns a 16-bit integer, of which the upper 8 bits are status bits and the lower 8 bits vary, depending on the value of *cmd*. The upper bits of the return value are defined as follows:

Bit 15	Time out
Bit 14	Transmit shift register empty
Bit 13	Transmit holding register empty
Bit 12	Break detect
Bit 11	Framing error
Bit 10	Parity error
Bit 9	Overrun error
Bit 8	Data ready

If the *abyte* value could not be sent, bit 15 is set to 1. Otherwise, the remaining upper and lower bits are appropriately set. For example, if a framing error has occurred, bit 11 is set to 1.

With a *cmd* value of 2, the byte read is in the lower bits of the return value if there is no error. If an error occurs, at least one of the upper bits is set to 1. If no upper bits are set to 1, the byte was received without error.

With a *cmd* value of 0 or 3, the return value has the upper bits set as defined, and the lower bits are defined as follows:

Bit 7	Received line signal detect
Bit 6	Ring indicator
Bit 5	Data set ready
Bit 4	Clear to send
Bit 3	Change in receive line signal detector
Bit 2	Trailing edge ring detector
Bit 1	Change in data set ready
Bit 0	Change in clear to send

biosdisk

bios.h

Function

Issues BIOS disk drive services.

Syntax

```
int biosdisk(int cmd, int drive, int head, int track, int sector, int nsects, void *buffer
);
```

Remarks

biosdisk uses interrupt 0x13 to issue disk operations directly to the BIOS.

drive is a number that specifies which disk drive is to be used: 0 for the first floppy disk drive, 1 for the second floppy disk drive, 2 for the third, and so on. For hard disk drives, a *drive* value of 0x80 specifies the first drive, 0x81 specifies the second, 0x82 the third, and so forth.

For hard disks, the physical drive is specified, not the disk partition. If necessary, the application program must interpret the partition table information itself.

cmd indicates the operation to perform. Depending on the value of *cmd*, the other parameters might or might not be needed.

Here are the possible values for *cmd* for the IBM PC, XT, AT, or PS/2, or any compatible system:

Value	Description
0	Resets disk system, forcing the drive controller to do a hard reset. All other parameters are ignored.
1	Returns the status of the last disk operation. All other parameters are ignored.
2	Reads one or more disk sectors into memory. The starting sector to read is given by *head, track*, and *sector*. The number of sectors is given by *nsects*. The data is read, 512 bytes per sector, into *buffer*.
3	Writes one or more disk sectors from memory. The starting sector to write is given by *head, track*, and *sector*. The number of sectors is given by *nsects*. The data is written, 512 bytes per sector, from *buffer*.
4	Verifies one or more sectors. The starting sector is given by *head, track*, and *sector*. The number of sectors is given by *nsects*.
5	Formats a track. The track is specified by *head* and *track*. *buffer* points to a table of sector headers to be written on the named *track*. See the *Technical Reference Manual* for the IBM PC for a description of this table and the format operation.

The following cmd values are allowed only for the XT, AT, PS/2, and compatibles:

Value	Description
6	Formats a track and sets bad sector flags.
7	Formats the drive beginning at a specific track.
8	Returns the current drive parameters. The drive information is returned in *buffer* in the first 4 bytes.

Value	Description
9	Initializes drive-pair characteristics.
10	Does a long read, which reads 512 plus 4 extra bytes per sector.
11	Does a long write, which writes 512 plus 4 extra bytes per sector.
12	Does a disk seek.
13	Alternates disk reset.
14	Reads sector buffer.
15	Writes sector buffer.
16	Tests whether the named drive is ready.
17	Recalibrates the drive.
18	Controller RAM diagnostic.
19	Drive diagnostic.
20	Controller internal diagnostic.

Note *biosdisk* operates below the level of files on raw sectors. *It can destoy file contents and directories on a hard disk.*

Return value

biosdisk returns a status byte composed of the following bits:

Bits	Description
0x00	Operation successful.
0x01	Bad command.
0x02	Address mark not found.
0x03	Attempt to write to write-protected disk.
0x04	Sector not found.
0x05	Reset failed (hard disk).
0x06	Disk changed since last operation.
0x07	Drive parameter activity failed.
0x08	Direct memory access (DMA) overrun.
0x09	Attempt to perform DMA across 64K boundary.
0x0A	Bad sector detected.
0x0B	Bad track detected.
0x0C	Unsupported track.
0x10	Bad CRC/ECC on disk read.
0x11	CRC/ECC corrected data error.
0x20	Controller has failed.
0x40	Seek operation failed.
0x80	Attachment failed to respond.
0xAA	Drive not ready (hard disk only).
0xBB	Undefined error occurred (hard disk only).
0xCC	Write fault occurred.
0xE0	Status error.
0xFF	Sense operation failed.

0x11 is not an error because the data is correct. The value is returned to give the application an opportunity to decide for itself.

See also

absread, *abswrite*

_bios_disk bios.h

Function

Issues BIOS disk drive services

Syntax

```
unsigned _bios_disk(unsigned cmd, struct diskinfo_t *dinfo);
```

Remarks

_bios_disk uses interrupt 0x13 to issue disk operations directly to the BIOS. The *cmd* argument specifies the operation to perform, and *dinfo* points to a *diskinfo_t* structure that contains the remaining parameters required by the operation.

The *diskinfo_t* structure (defined in bios.h) has the following format:

```
struct diskinfo_t {
   unsigned drive, head, track, sector, nsectors;
   void far *buffer;
};
```

drive is a number that specifies which disk drive is to be used: 0 for the first floppy disk drive, 1 for the second floppy disk drive, 2 for the third, and so on. For hard disk drives, a *drive* value of 0x80 specifies the first drive, 0x81 specifies the second, 0x82 the third, and so forth.

For hard disks, the physical drive is specified, not the disk partition. If necessary, the application program must interpret the partition table information itself.

Depending on the value of *cmd*, the other parameters in the *diskinfo_t* structure might or might not be needed.

The possible values for *cmd* (defined in bios.h) are the following:

Value	Description
_DISK_RESET	Resets disk system, forcing the drive controller to do a hard reset. All *diskinfo_t* parameters are ignored.
_DISK_STATUS	Returns the status of the last disk operation. All *diskinfo_t* parameters are ignored.
_DISK_READ	Reads one or more disk sectors into memory. The starting sector to read is given by *head*, *track*, and *sector*. The number of sectors is given by *nsectors*. The data is read, 512 bytes per sector, into *buffer*. If the operation is successful, the high byte of the return value will be 0 and the low byte will contain the number of sectors. If an error occurred, the high byte of the return value will have one of the following values:
	0x01 Bad command.

Value	Description	
	0x02	Address mark not found.
	0x03	Attempt to write to write-protected disk.
	0x04	Sector not found.
	0x05	Reset failed (hard disk).
	0x06	Disk changed since last operation.
	0x07	Drive parameter activity failed.
	0x08	Direct memory access (DMA) overrun.
	0x09	Attempt to perform DMA across 64K boundary.
	0x0A	Bad sector detected.
	0x0B	Bad track detected.
	0x0C	Unsupported track.
	0x10	Bad CRC/ECC on disk read.
	0x11	CRC/ECC corrected data error.
	0x20	Controller has failed.
	0x40	Seek operation failed.
	0x80	Attachment failed to respond.
	0xAA	Drive not ready (hard disk only).
	0xBB	Undefined error occurred (hard disk only).
	0xCC	Write fault occurred.
	0xE0	Status error.
	0xFF	Sense operation failed.
	0x11 is not an error because the data is correct. The value is returned to give the application an opportunity to decide for itself.	
_DISK_WRITE	Writes one or more disk sectors from memory. The starting sector to write is given by *head*, *track*, and *sector*. The number of sectors is given by *nsectors*. The data is written, 512 bytes per sector, from *buffer*. See _DISK_READ (above) for a description of the return value.	
_DISK_VERIFY	Verifies one or more sectors. The starting sector is given by *head*, *track*, and *sector*. The number of sectors is given by *nsectors*. See _DISK_READ (above) for a description of the return value.	
_DISK_FORMAT	Formats a track. The track is specified by *head* and *track*. *buffer* points to a table of sector headers to be written on the named *track*. See the *Technical Reference Manual* for the IBM PC for a description of this table and the format operation.	

Return value

_bios_disk returns the value of the AX register set by the INT 0x13 BIOS call.

See Also

absread, *abswrite*, *biosdisk*

bioskey — bios.h

Function

Keyboard interface, using BIOS services directly.

Syntax

```
int bioskey(int cmd);
```

Remarks

bioskey performs various keyboard operations using BIOS interrupt 0x16. The parameter *cmd* determines the exact operation.

Return value

The value returned by *bioskey* depends on the task it performs, determined by the value of *cmd*:

Value	Description		
0	If the lower 8 bits are nonzero, *bioskey* returns the ASCII character for the next keystroke waiting in the queue or the next key pressed at the keyboard. If the lower 8 bits are zero, the upper 8 bits are the extended keyboard codes defined in the IBM PC *Technical Reference Manual.*		
1	This tests whether a keystroke is available to be read. A return value of zero means no key is available. The return value is 0xFFFFF (–1) if *Ctrl-Brk* has been pressed. Otherwise, the value of the next keystroke is returned. The keystroke itself is kept to be returned by the next call to *bioskey* that has a *cmd* value of zero.		
2	Requests the current shift key status. The value is obtained by ORing the following values together:		
	Bit 7	0x80	*Insert* on
	Bit 6	0x40	*Caps* on
	Bit 5	0x20	*Num Lock* on
	Bit 4	0x10	*Scroll Lock* on
	Bit 3	0x08	*Alt* pressed
	Bit 2	0x04	*Ctrl* pressed
	Bit 1	0x02	← *Shift* pressed
	Bit 0	0x01	→ *Shift* pressed

_bios_keybrd — bios.h

Function

Keyboard interface, using BIOS services directly.

Syntax

```
unsigned _bios_keybrd(unsigned cmd);
```

Remarks

_bios_keybrd performs various keyboard operations using BIOS interrupt 0x16. The parameter *cmd* determines the exact operation.

Return value

The value returned by *_bios_keybrd* depends on the task it performs, determined by the value of *cmd* (defined in bios.h):

Value	Description		
_KEYBRD_READ	If the lower 8 bits are nonzero, *_bios_keybrd* returns the ASCII character for the next keystroke waiting in the queue or the next key pressed at the keyboard. If the lower 8 bits are zero, the upper 8 bits are the extended keyboard codes defined in the IBM PC *Technical Reference Manual*.		
_NKEYBRD_READ	Use this value instead of _KEYBRD_READY to read the keyboard codes for enhanced keyboards, which have additional cursor and function keys.		
_KEYBRD_READY	This tests whether a keystroke is available to be read. A return value of zero means no key is available. The return value is 0xFFFF (–1) if *Ctrl-Brk* has been pressed. Otherwise, the value of the next keystroke is returned, as described in _KEYBRD_READ (above). The keystroke itself is kept to be returned by the next call to *_bios_keybrd* that has a *cmd* value of _KEYBRD_READ or _NKEYBRD_READ.		
_NKEYBRD_READY	Use this value to check the status of enhanced keyboards, which have additional cursor and function keys.		
_KEYBRD_SHIFTSTATUS	Requests the current shift key status. The value will contain an OR of zero or more of the following values:		
	Bit 7	0x80	*Insert* on
	Bit 6	0x40	*Caps* on
	Bit 5	0x20	*Num Lock* on
	Bit 4	0x10	*Scroll Lock* on
	Bit 3	0x08	*Alt* pressed
	Bit 2	0x04	*Ctrl* pressed
	Bit 1	0x02	Left *Shift* pressed
	Bit 0	0x01	Right *Shift* pressed
_NKEYBRD_SHIFTSTATUS	Use this value instead of _KEYBRD_SHIFTSTATUS to request the full 16-bit shift key status for enhanced keyboards. The return value will contain an OR of zero or more of the bits defined above in _KEYBRD_SHIFTSTATUS, and additionally, any of the following bits:		
	Bit 15	0x8000	*Sys Req* pressed
	Bit 14	0x4000	*Caps Lock* pressed
	Bit 13	0x2000	*Num Lock* pressed
	Bit 12	0x1000	*Scroll Lock* pressed
	Bit 11	0x0800	Right *Alt* pressed
	Bit 10	0x0400	Right *Ctrl* pressed
	Bit 9	0x0200	Left *Alt* pressed
	Bit 8	0x0100	Left *Ctrl* pressed

biosprint — bios.h

Function

Printer I/O using BIOS services directly.

Syntax

```
int biosprint(int cmd, int abyte, int port);
```

Remarks

biosprint performs various printer functions on the printer identified by the parameter *port* using BIOS interrupt 0x17.

A *port* value of 0 corresponds to LPT1; a *port* value of 1 corresponds to LPT2; and so on.

The value of *cmd* can be one of the following:

0	Prints the character in *abyte*.
1	Initializes the printer port.
2	Reads the printer status.

The value of *abyte* can be 0 to 255.

Return value

The value returned from any of these operations is the current printer status, which is obtained by ORing these bit values together:

Bit 0	0x01	Device time out
Bit 3	0x08	I/O error
Bit 4	0x10	Selected
Bit 5	0x20	Out of paper
Bit 6	0x40	Acknowledge
Bit 7	0x80	Not busy

_bios_printer **bios.h**

Function

Printer I/O using BIOS services directly.

Syntax

```
unsigned _bios_printer(int cmd, int port, int abyte);
```

Remarks

_bios_printer performs various printer functions on the printer identified by the parameter *port* using BIOS interrupt 0x17.

A *port* value of 0 corresponds to LPT1; a *port* value of 1 corresponds to LPT2; and so on.

The value of *cmd* can be one of the following values (defined in bios.h):

_PRINTER_WRITE	Prints the character in *abyte*. The value of *abyte* can be 0 to 255.
_PRINTER_INIT	Initializes the printer port. The *abyte* argument is ignored.
_PRINTER_STATUS	Reads the printer status. The *abyte* argument is ignored.

Return value

The value returned from any of these operations is the current printer status, which is obtained by ORing these bit values together:

Bit 0	0x01	Device time out
Bit 3	0x08	I/O error
Bit 4	0x10	Selected
Bit 5	0x20	Out of paper
Bit 6	0x40	Acknowledge
Bit 7	0x80	Not busy

_bios_serialcom bios.h

Function

Performs serial I/O.

Syntax

```
unsigned _bios_serialcom(int cmd, int port, char abyte);
```

Remarks

_bios_serialcom performs various RS-232 communications over the I/O port given in *port*.

A *port* value of 0 corresponds to COM1, 1 corresponds to COM2, and so forth.

The value of *cmd* can be one of the following values (defined in bios.h):

Value	Description
_COM_INIT	Sets the communications parameters to the value in *abyte*.
_COM_SEND	Sends the character in *abyte* out over the communications line.
_COM_RECEIVE	Receives a character from the communications line. The *abyte* argument is ignored.
_COM_STATUS	Returns the current status of the communications port. The *abyte* argument is ignored.

When *cmd* is _COM_INIT, *abyte* is a OR combination of the following bits:

- Select only one of these:

  _COM_CHR7 7 data bits
  _COM_CHR8 8 data bits

- Select only one of these:

  _COM_STOP1 1 stop bit
  _COM_STOP2 2 stop bits

- Select only one of these:

_COM_NOPARITY	No parity
_COM_ODDPARITY	Odd parity
_COM_EVENPARITY	Even parity

- Select only one of these:

_COM_110	110 baud
_COM_150	150 baud
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1200 baud
_COM_2400	2400 baud
_COM_4800	4800 baud
_COM_9600	9600 baud

For example, a value of (_COM_9600 | _COM_ODDPARITY | _COM_STOP1 | _COM_CHR8) for *abyte* sets the communications port to 9600 baud, odd parity, 1 stop bit, and 8 data bits. *_bios_serialcom* uses the BIOS 0x14 interrupt.

Return value

For all values of *cmd*, *_bios_serialcom* returns a 16-bit integer of which the upper 8 bits are status bits and the lower 8 bits vary, depending on the value of *cmd*. The upper bits of the return value are defined as follows:

Bit 15	Time out
Bit 14	Transmit shift register empty
Bit 13	Transmit holding register empty
Bit 12	Break detect
Bit 11	Framing error
Bit 10	Parity error
Bit 9	Overrun error
Bit 8	Data ready

If the *abyte* value could not be sent, bit 15 is set to 1. Otherwise, the remaining upper and lower bits are appropriately set. For example, if a framing error has occurred, bit 11 is set to 1.

With a *cmd* value of _COM_RECEIVE, the byte read is in the lower bits of the return value if there is no error. If an error occurs, at least one of the upper bits is set to 1. If no upper bits are set to 1, the byte was received without error.

With a *cmd* value of _COM_INIT or _COM_STATUS, the return value has the upper bits set as defined, and the lower bits are defined as follows:

Bit 7	Received line signal detect
Bit 6	Ring indicator
Bit 5	Data set ready
Bit 4	Clear to send
Bit 3	Change in receive line signal detector
Bit 2	Trailing edge ring detector
Bit 1	Change in data set ready
Bit 0	Change in clear to send

brk alloc.h

Function
Changes data-segment space allocation.

Syntax

```
int brk(void *addr);
```

Remarks
brk dynamically changes the amount of space allocated to the calling program's heap. The change is made by resetting the program's *break value*, which is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

brk sets the break value to *addr* and changes the allocated space accordingly.

This function will fail without making any change in the allocated space if such a change would allocate more space than is allowable.

Return value
Upon successful completion, *brk* returns a value of 0. On failure, this function returns a value of –1 and the global variable *errno* is set to the following:

ENOMEM	Not enough memory

See also
coreleft, *sbrk*

coreleft alloc.h

Function
Returns a measure of unused RAM memory.

Syntax
In the tiny, small, and medium models:

```
unsigned coreleft(void);
```

In the compact, large, and huge models:

```
unsigned long coreleft(void);
```

Remarks
coreleft returns a measure of RAM memory not in use. It gives a different measurement value, depending on whether the memory model is of the small data group or the large data group.

Return value

In the small data models, *coreleft* returns the amount of unused memory between the top of the heap and the stack. In the large data models, *coreleft* returns the amount of memory between the highest allocated block and the end of available memory.

See also

allocmem, brk, farcoreleft, malloc

delay — dos.h

Function

Suspends execution for an interval (milliseconds).

Syntax

```
void delay(unsigned milliseconds);
```

Remarks

With a call to *delay*, the current program is suspended from execution for the number of milliseconds specified by the argument *milliseconds*. It is no longer necessary to make a calibration call to delay before using it. *delay* is accurate to a millisecond.

Return value

None.

See also

nosound, sleep, sound

farcoreleft — alloc.h

Function

Returns a measure of unused memory in far heap.

Syntax

```
unsigned long farcoreleft(void);
```

Remarks

farcoreleft returns a measure of the amount of unused memory in the far heap beyond the highest allocated block.

A tiny model program cannot make use of *farcoreleft*.

Return value
farcoreleft returns the total amount of space left in the far heap, between the highest allocated block and the end of available memory.

See also
coreleft, farcalloc, farmalloc

farheapcheck alloc.h

Function
Checks and verifies the far heap.

Syntax
```
int farheapcheck(void);
```

Remarks
farheapcheck walks through the far heap and examines each block, checking its pointers, size, and other critical attributes.

Return value
The return value is less than zero for an error and greater than zero for success.

_HEAPEMPTY is returned if there is no heap (value 1).
_HEAPOK is returned if the heap is verified (value 2).
_HEAPCORRUPT is returned if the heap has been corrupted (value –1).

See also
heapcheck

farheapcheckfree alloc.h

Function
Checks the free blocks on the far heap for a constant value.

Syntax
```
int farheapcheckfree(unsigned int fillvalue);
```

Return value
The return value is less than zero for an error and greater than zero for success.

_HEAPEMPTY is returned if there is no heap (value 1).
_HEAPOK is returned if the heap is accurate (value 2).
_HEAPCORRUPT is returned if the heap has been corrupted (value –1).
_BADVALUE is returned if a value other than the fill value was found (value –3).

See also
farheapfillfree, heapcheckfree

farheapchecknode alloc.h

Function
Checks and verifies a single node on the far heap.

Syntax

```
int farheapchecknode(void *node);
```

Remarks
If a node has been freed and *farheapchecknode* is called with a pointer to the freed block, *farheapchecknode* can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists.

Return value
The return value is less than zero for an error and greater than zero for success.

_HEAPEMPTY is returned if there is no heap (value 1).
_HEAPCORRUPT is returned if the heap has been corrupted (value –1).
_BADNODE is returned if the node could not be found (value –2).
_FREEENTRY is returned if the node is a free block (value 3).
_USEDENTRY is returned if the node is a used block (value 4).

See also
heapchecknode

farheapfillfree alloc.h

Function
Fills the free blocks on the far heap with a constant value.

Syntax

```
int farheapfillfree(unsigned int fillvalue);
```

Return value
The return value is less than zero for an error and greater than zero for success.

_HEAPEMPTY is returned if there is no heap (value 1).
_HEAPOK is returned if the heap is accurate (value 2).
_HEAPCORRUPT is returned if the heap has been corrupted (value –1).

See also
farheapcheckfree, heapfillfree

farheapwalk — alloc.h

Function
farheapwalk is used to "walk" through the far heap node by node.

Syntax
```
int farheapwalk(struct farheapinfo *hi);
```

Remarks
farheapwalk assumes the heap is correct. Use *farheapcheck* to verify the heap before using *farheapwalk*. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to *farheapwalk*.

farheapwalk receives a pointer to a structure of type *heapinfo* (defined in alloc.h). For the first call to *farheapwalk*, set the hi.ptr field to null. *farheapwalk* returns with hi.ptr containing the address of the first block. hi.size holds the size of the block in bytes. hi.in_use is a flag that's set if the block is currently in use.

Return value
_HEAPEMPTY is returned if there is no heap (value 1).
_HEAPOK is returned if the heapinfo block contains valid data (value 2).
_HEAPEND is returned if the end of the heap has been reached (value 5).

See also
heapwalk

freemem, _dos_freemem — dos.h

Function
Frees a previously allocated DOS memory block.

Syntax
```
int freemem(unsigned segx);
unsigned _dos_freemem(unsigned segx);
```

Remarks
freemem frees a memory block allocated by a previous call to *allocmem*.

_dos_freemem frees a memory block allocated by a previous call to *_dos_allocmem*. *segx* is the segment address of that block.

Return value

freemem and *_dos_freemem* return 0 on success.

In the event of error, *freemem* returns –1 and sets *errno*.

In the event of error, *_dos_freemem* returns the DOS error code and sets *errno*.

In the event of error, these functions set global variable *errno* to the following:

ENOMEM	Insufficient memory

See also

allocmem, *_dos_allocmem*, *free*

harderr, hardresume, hardretn — dos.h

Function

Establishes and handles hardware errors.

Syntax

```
void harderr(int (*handler)());
void hardresume(int axret);
void hardretn(int retn);
```

Remarks

The error handler established by *harderr* can call *hardresume* to return to DOS. The return value of the *rescode* (result code) of *hardresume* contains an abort (2), retry (1), or ignore (0) indicator. The abort is accomplished by invoking DOS interrupt 0x23, the control-break interrupt.

The error handler established by *harderr* can return directly to the application program by calling *hardretn*. The returned value is whatever value you passed to *hardretn*.

harderr establishes a hardware error handler for the current program. This error handler is invoked whenever an interrupt 0x24 occurs. (See your DOS reference manuals for a discussion of the interrupt.)

The function pointed to by *handler* is called when such an interrupt occurs. The handler function is called with the following arguments:

```
handler(int errval, int ax, int bp, int si);
```

errval is the error code set in the DI register by DOS. *ax*, *bp*, and *si* are the values DOS sets for the AX, BP, and SI registers, respectively.

- *ax* indicates whether a disk error or other device error was encountered. If *ax* is nonnegative, a disk error was encountered; otherwise, the error was a device error. For a disk error, *ax* ANDed with 0x00FF gives the failing drive number (0 equals A, 1 equals B, and so on).

- *bp* and *si* together point to the device driver header of the failing driver. *bp* contains the segment address, and *si* the offset.

The function pointed to by *handler* is not called directly. *harderr* establishes a DOS interrupt handler that calls the function.

The handler can issue DOS calls 1 through 0xC; any other DOS call corrupts DOS. In particular, any of the C standard I/O or UNIX-emulation I/O calls *cannot* be used.

The handler must return 0 for ignore, 1 for retry, and 2 for abort.

Return value

None.

See also

peek, *poke*

_harderr dos.h

Function

Establishes a hardware error handler.

Syntax

```
void _harderr(int (far *handler)());
```

Remarks

_harderr establishes a hardware error handler for the current program. This error handler is invoked whenever an interrupt 0x24 occurs. (See your DOS reference manuals for a discussion of the interrupt.)

The function pointed to by *handler* is called when such an interrupt occurs. The handler function is called with the following arguments:

```
void far handler(unsigned deverr, unsigned errval, unsigned far *devhdr);
```

- *deverr* is the device error code (passed to the handler by DOS in the AX register).
- *errval* is the error code (passed to the handler by DOS in the DI register).
- *devhdr* a far pointer to the driver header of the device that caused the error (passed to the handler by DOS in the BP:SI register pair).

The handler should use these arguments instead of referring directly to the CPU registers.

deverr indicates whether a disk error or other device error was encountered. If bit 15 of *deverr* is 0, a disk error was encountered. Otherwise, the error was a device error. For a disk error, *deverr* ANDed with 0x00FF give the failing drive number (0 equals A, 1 equals B, and so on).

The function pointed to by *handler* is not called directly. *_harderr* establishes a DOS interrupt handler that calls the function.

The handler can issue DOS calls 1 through 0xC; any other DOS call corrupts DOS. In particular, any of the C standard I/O or UNIX-emulation I/O calls *cannot* be used.

The handler does not return a value, and it must exit using *_hardretn* or *_hardresume.*

Return value
None.

See also
_hardresume, _hardretn

_hardresume — dos.h

Function
Hardware error handler.

Syntax

```
void _hardresume(int rescode);
```

Remarks
The error handler established by *_harderr* can call *_hardresume* to return to DOS. The return value of the *rescode* (result code) of *_hardresume* contains one of the following values:

_HARDERR_ABORT	Abort the program by invoking DOS interrupt 0x23, the control-break interrupt.
_HARDERR_IGNORE	Ignore the error.
_HARDERR_RETRY	Retry the operation.
_HARDERR_FAIL	Fail the operation.

Return value
The *_hardresume* function does not return a value, and does not return to the caller.

See also
_harderr, _hardretn

_hardretn — dos.h

Function
Hardware error handler.

Syntax

```
void _hardretn(int retn);
```

Remarks

The error handler established by *_harderr* can return directly to the application program by calling *_hardretn*.

If the DOS function that caused the error is less than 0x38, and it is a function that can indicate an error condition, then *_hardretn* will return to the application program with the AL register set to 0xFF. The *retn* argument is ignored for all DOS functions less than 0x38.

If the DOS function is greater than or equal to 0x38, the *retn* argument should be a DOS error code; it is returned to the application program in the AX register. The carry flag is also set to indicate to the application that the operation resulted in an error.

Return value

The *_hardresume* function does not return a value, and does not return to the caller.

See also

_harderr, *_hardresume*

keep, _dos_keep — dos.h

Function

Exits and remains resident.

Syntax

```
void keep(unsigned char status, unsigned size);
void _dos_keep(unsigned char status, unsigned size);
```

Remarks

keep and *_dos_keep* return to DOS with the exit status in *status*. The current program remains resident, however. The program is set to *size* paragraphs in length, and the remainder of the memory of the program is freed.

keep and *_dos_keep* can be used when installing TSR programs. *keep* and *_dos_keep* use DOS function 0x31.

Before *_dos_keep* exits, it calls any registered "exit functions" (posted with *atexit*), flushes file buffers, and restores interrupt vectors modified by the startup code.

Return value

None.

See also
abort, exit

nosound **dos.h**

Function
Turns PC speaker off.

Syntax

```
void nosound(void);
```

Remarks
Turns the speaker off after it has been turned on by a call to *sound*.

Return value
None.

See also
delay, sound

_OvrInitEms **dos.h**

Function
Initializes expanded memory swapping for the overlay manager.

Syntax

```
int _ _cdecl _ _far _OvrInitEms(unsigned emsHandle, unsigned firstPage, unsigned pages);
```

Remarks
_OvrInitEms checks for the presence of expanded memory by looking for an EMS driver and allocating memory from it. If *emsHandle* is zero, the overlay manager allocates EMS pages and uses them for swapping. If *emsHandle* is not zero, then it should be a valid EMS handle; the overlay manager will use it for swapping. In that case, you can specify *firstPage*, where the swapping can start inside that area.

In both cases, a nonzero *pages* parameter gives the limit of the usable pages by the overlay manager.

Return value
_OvrInitEms returns 0 if the overlay manager is able to use expanded memory for swapping.

See also

_OvrInitExt, *_ovrbuffer* (global variable)

_OvrInitExt dos.h

Function

Initializes extended memory swapping for the overlay manager.

Syntax

```
int _ _cdecl _ _far _OvrInitExt(unsigned long startAddress, unsigned long length);
```

Remarks

_OvrInitExt checks for the presence of extended memory, using the known methods to detect the presence of other programs using extended memory, and allocates memory from it. If *startAddress* is zero, the overlay manager determines the start address and uses, at most, the size of the overlays. If *startAddress* is not zero, then the overlay manager uses the extended memory above that address.

In both cases, a nonzero *length* parameter gives the limit of the usable extended memory by the overlay manager.

Return value

_OvrInitExt returns 0 if the overlay manager is able to use extended memory for swapping.

See also

_OvrInitEms, *_ovrbuffer* (global variable)

randbrd dos.h

Function

Reads random block.

Syntax

```
int randbrd(struct fcb *fcb, int rcnt);
```

Remarks

randbrd reads *rcnt* number of records using the open file control block (FCB) pointed to by *fcb*. The records are read into memory at the current disk transfer address (DTA). They are read from the disk record indicated in the random record field of the FCB. This is accomplished by calling DOS system call 0x27.

The actual number of records read can be determined by examining the random record field of the FCB. The random record field is advanced by the number of records actually read.

Return value

The following values are returned, depending on the result of the *randbrd* operation:

0	All records are read.
1	End-of-file is reached and the last record read is complete.
2	Reading records would have wrapped around address 0xFFFF (as many records as possible are read).
3	End-of-file is reached with the last record incomplete.

See also

getdta, randbwr, setdta

randbwr — dos.h

Function

Writes random block.

Syntax

```
int randbwr(struct fcb *fcb, int rcnt);
```

Remarks

randbwr writes *rcnt* number of records to disk using the open file control block (FCB) pointed to by *fcb*. This is accomplished using DOS system call 0x28. If *rcnt* is 0, the file is truncated to the length indicated by the random record field.

The actual number of records written can be determined by examining the random record field of the FCB. The random record field is advanced by the number of records actually written.

Return value

The following values are returned, depending on the result of the *randbwr* operation:

0	All records are written.
1	There is not enough disk space to write the records (no records are written).
2	Writing records would have wrapped around address 0xFFFF (as many records as possible are written).

See also

randbrd

sbrk

alloc.h

Function

Changes data segment space allocation.

Syntax

```
void *sbrk(int incr);
```

Remarks

sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated space is decreased.

sbrk will fail without making any change in the allocated space if such a change would result in more space being allocated than is allowable.

Return value

Upon successful completion, *sbrk* returns the old break value. On failure, *sbrk* returns a value of –1, and the global variable *errno* is set to the following:

ENOMEM	Not enough core

See also

brk

setblock, _dos_setblock

dos.h

Function

Modifies the size of a previously allocated block.

Syntax

```
int setblock(unsigned segx, unsigned newsize);
unsigned _dos_setblock(unsigned newsize, unsigned segx, unsigned *maxp);
```

Remarks

setblock and *_dos_setblock* modify the size of a memory segment. *segx* is the segment address returned by a previous call to *allocmem* or *_dos_allocmem*. *newsize* is the new, requested size in paragraphs. If the segment cannot be changed to the new size, *_dos_setblock* stores the size of the largest possible segment at the location pointed to by *maxp*.

Return value

setblock returns –1 on success. In the event of error, it returns the size of the largest possible block (in paragraphs), and the global variable *_doserrno* is set.

_dos_setblock returns 0 on success. In the event of error, it returns the DOS error code, and the global variable *errno* is set to the following:

ENOMEM	Not enough memory, or bad segment address

See also
allocmem, freemem

sound — dos.h

Function
Turns PC speaker on at specified frequency.

Syntax

```
void sound(unsigned frequency);
```

Remarks
sound turns on the PC's speaker at a given frequency. *frequency* specifies the frequency of the sound in hertz (cycles per second). To turn the speaker off after a call to *sound*, call the function *nosound.*

See also
delay, nosound

Appendix

A

DOS libraries

This appendix provides an overview of the Borland C++ library routines available to 16-bit DOS-only applications. Library routines are composed of functions and macros that you can call from within your C and C++ programs to perform a wide variety of tasks. These tasks include low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more.

This appendix provides the following information:

- Names the libraries and files found in the LIB subdirectory, and describes their uses.
- Categorizes the library routines according to the type of tasks they perform.

The run-time libraries

The DOS-specific applications use static run-time libraries (OBJ and LIB). The libraries summarized in this appendix are available only to the 16-bit development tools. See the *Borland C++ Library Reference*, Chapter 1, "Library cross-reference", for a description of additional libraries.

Several versions of the run-time library are available. For example, there are memory-model specific versions and diagnostic versions. There are also optional libraries that provide containers, graphics, and mathematics.

Keep these guidelines in mind when selecting which run-time libraries to use:

- The libraries listed below are for use in 16-bit DOS applications only.
- Information on additional DOS routines can be found in the *Borland C++ Library Reference*.
- Exception-handling should not be used with overlays. See the discussion of exceptions on page 21.

The DOS support libraries

The static (OBJ and LIB) 16-bit Borland C++ run-time libraries are contained in the LIB subdirectory of your installation. For each of the library file names, the '?' character represents one of the six (tiny, compact, small, medium, large, and huge) distinct memory models supported by Borland. Each model has its own library file and math file containing versions of the routines written for that particular model. See Chapter 1, "DOS memory management" for details on memory models.

The following table lists the Borland C++ libraries names and uses that are available for 16-bit DOS-only applications. See the *Borland C++ User's Guide* for information on linkers, linker options, requirements, and selection of libraries. See also the *Borland C++ Library Reference* for more information on other libraries and header files that can provide additional DOS support.

Table A.1 Summary of DOS run-time libraries

File name	Use
BIDSH.LIB	Huge memory model of Borland classlibs
BIDSDBH.LIB	Diagnostic version of the above library
C?.LIB	DOS-only libraries
C0F?.OBJ	MS compatible startup
C0?.OBJ	BC startup
EMU.LIB	Floating-point emulation
FP87.LIB	For programs that run on machines with 80x87 coprocessor
GRAPHICS.LIB	Borland graphics interface
MATH?.LIB	Math routines
OVERLAY.LIB	Overlay development

Graphics routines

These routines let you create onscreen graphics with text.

arc	(graphics.h)	*getdefaultpalette*	(graphics.h)
bar	(graphics.h)	*getdrivername*	(graphics.h)
bar3d	(graphics.h)	*getfillpattern*	(graphics.h)
circle	(graphics.h)	*getfillsettings*	(graphics.h)
cleardevice	(graphics.h)	*getgraphmode*	(graphics.h)
clearviewport	(graphics.h)	*getimage*	(graphics.h)
closegraph	(graphics.h)	*getlinesettings*	(graphics.h)
detectgraph	(graphics.h)	*getmaxcolor*	(graphics.h)
drawpoly	(graphics.h)	*getmaxmode*	(graphics.h)
ellipse	(graphics.h)	*getmaxx*	(graphics.h)
fillellipse	(graphics.h)	*getmaxy*	(graphics.h)
fillpoly	(graphics.h)	*getmodename*	(graphics.h)
floodfill	(graphics.h)	*getmoderange*	(graphics.h)
getarccoords	(graphics.h)	*getpalette*	(graphics.h)
getaspectratio	(graphics.h)	*getpalettesize*	(graphics.h)
getbkcolor	(graphics.h)	*getpixel*	(graphics.h)
getcolor	(graphics.h)	*gettextsettings*	(graphics.h)

getviewsettings	(graphics.h)	*registerbgifont*	(graphics.h)
getx	(graphics.h)	*restorecrtmode*	(graphics.h)
gety	(graphics.h)	*sector*	(graphics.h)
graphdefaults	(graphics.h)	*setactivepage*	(graphics.h)
grapherrormsg	(graphics.h)	*setallpalette*	(graphics.h)
_graphfreemem	(graphics.h)	*setaspectratio*	(graphics.h)
_graphgetmem	(graphics.h)	*setbkcolor*	(graphics.h)
graphresult	(graphics.h)	*setcolor*	(graphics.h)
imagesize	(graphics.h)	*_setcursortype*	(conio.h)
initgraph	(graphics.h)	*setfillpattern*	(graphics.h)
installuserdriver	(graphics.h)	*setfillstyle*	(graphics.h)
installuserfont	(graphics.h)	*setgraphbufsize*	(graphics.h)
line	(graphics.h)	*setgraphmode*	(graphics.h)
linerel	(graphics.h)	*setlinestyle*	(graphics.h)
lineto	(graphics.h)	*setpalette*	(graphics.h)
moverel	(graphics.h)	*setrgbpalette*	(graphics.h)
moveto	(graphics.h)	*settextjustify*	(graphics.h)
outtext	(graphics.h)	*settextstyle*	(graphics.h)
outtextxy	(graphics.h)	*setusercharsize*	(graphics.h)
pieslice	(graphics.h)	*setviewport*	(graphics.h)
putimage	(graphics.h)	*setvisualpage*	(graphics.h)
putpixel	(graphics.h)	*setwritemode*	(graphics.h)
rectangle	(graphics.h)	*textheight*	(graphics.h)
registerbgidriver	(graphics.h)	*textwidth*	(graphics.h)

Interface routines

These routines provide operating-system BIOS and machine-specific capabilities.

absread	(dos.h)	*_dos_freemem*	(dos.h)
abswrite	(dos.h)	*freemem*	(dos.h)
bioscom	(bios.h)	*_harderr*	(dos.h)
_bios_disk	(bios.h)	*harderr*	(dos.h)
biosdisk	(bios.h)	*_hardresume*	(dos.h)
_bios_keybrd	(bios.h)	*hardresume*	(dos.h)
bioskey	(bios.h)	*_hardretn*	(dos.h)
biosprint	(bios.h)	*hardretn*	(dos.h)
_bios_printer	(bios.h)	*keep*	(dos.h)
_bios_serialcom	(bios.h)	*randbrd*	(dos.h)
_dos_keep	(dos.h)	*randbwr*	(dos.h)

Memory routines

These routines provide dynamic memory allocation in the small-data and large-data models.

allocmem	(dos.h)	*farheapcheck*	(alloc.h)
brk	(alloc.h)	*farheapcheckfree*	(alloc.h)
coreleft	(alloc.h, stdlib.h)	*farheapchecknode*	(alloc.h)
_dos_allocmem	(dos.h)	*farheapfillfree*	(alloc.h)
_dos_freemem	(dos.h)	*farheapwalk*	(alloc.h)
_dos_setblock	(dos.h)	*farrealloc*	(alloc.h)
farcoreleft	(alloc.h)	*sbrk*	(alloc.h)

Miscellaneous routines

These routines provide sound effects and time delay.

delay	(dos.h)	*sound*	(dos.h)
nosound	(dos.h)		

Appendix

B

DOS global variables

This appendix describes the Borland C++ global variables that are available for 16-bit DOS-only applications. Additional global variables are described in the *Library Reference*.

_heaplen **dos.h**

Function

Holds the length of the near heap.

Syntax

```
extern unsigned _heaplen;
```

Remarks

_heaplen specifies the size (in bytes) of the near heap in the small data models (tiny, small, and medium). *_heaplen* does not exist in the large data models (compact, large, and huge) because they do not have a near heap.

In the small and medium models, the data segment size is computed as follows:

```
data segment [small,medium] = global data + heap + stack
```

where the size of the stack can be adjusted with *_stklen*.

If *_heaplen* is set to 0, the program allocates 64K bytes for the data segment, and the effective heap size is

```
64K - (global data + stack) bytes
```

By default, *_heaplen* equals 0, so you'll get a 64K data segment unless you specify a particular *_heaplen* value.

In the tiny model, everything (including code) is in the same segment, so the data segment computations are adjusted to include the code plus 256 bytes for the program segment prefix (PSP).

```
data segment [tiny] = 256 + code + global data + heap + stack
```

If *_heaplen* equals 0 in the tiny model, the effective heap size is obtained by subtracting the PSP, code, global data, and stack from 64K.

In the compact and large models, there is no near heap, and the stack is in its own segment, so the data segment is

```
data segment [compact,large] = global data
```

In the huge model, the stack is a separate segment, and each module has its own data segment.

See also

_stklen

_ovrbuffer **dos.h**

Function

Changes the size of the overlay buffer.

Syntax

```
unsigned _ovrbuffer = size;
```

Remarks

The default overlay buffer size is twice the size of the largest overlay. This is adequate for some applications. But imagine that a particular function of a program is implemented through many modules, each of which is overlaid. If the total size of those modules is larger than the overlay buffer, a substantial amount of swapping will occur if the modules make frequent calls to each other.

The solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. You can do this by setting the *_ovrbuffer* global variable to the required size in paragraphs. For example, to set the overlay buffer to 128K, include the following statement in your code:

```
unsigned _ovrbuffer = 0x2000;
```

There is no general formula for determining the ideal overlay buffer size.

See also

_OvrInitEms, *_OvrInitExt*

_stklen

dos.h

Function

Holds size of the stack.

Syntax

```
extern unsigned _stklen;
```

Remarks

_stklen specifies the size of the stack for all six memory models. The minimum stack size allowed is 128 words; if you give a smaller value, *_stklen* is automatically adjusted to the minimum. The default stack size is 4K.

In the small and medium models, the data segment size is computed as follows:

```
data segment [small, medium] = global data + heap + stack
```

where the size of the heap can be adjusted with *_heaplen*.

In the tiny model, everything (including code) is in the same segment, so the data segment computations are adjusted to include the code plus 256 bytes for the program segment prefix (PSP).

```
data segment [tiny] = 256 + code + global data + heap + stack
```

In the compact and large models, there is no near heap, and the stack is in its own segment, so the data segment is simply

```
data segment [compact,large] = global data
```

In the huge model, the stack is a separate segment, and each module has its own data segment.

See also

_heaplen

Index

Symbols

!= operator
  huge pointer comparison and 9
<< operator
  complex numbers and 31
== operator
  huge pointer comparison and 9
>> operator
  complex numbers and 31

Numerics

0x13 BIOS interrupt 109
0x16 BIOS interrupt 113
0x17 BIOS interrupt 115
0x23 DOS interrupt 125
0x24 DOS interrupt 123, 124
0x25 DOS interrupt 105
0x26 DOS interrupt 106
0x27 DOS system call 128
0x28 DOS system call 129
0x31 DOS function 126
0x48 DOS system call 107
80x86 processors
  address segment
    offset notation 7
  functions (list) 135
  registers 4-6
80x87 coprocessors 28, 29
  See also numeric coprocessors
87 environment variable 29

A

absolute disk sectors 105, 106
absread function 105
abswrite function 106
access, memory (DMA) 110
accounting applications 31
active page 101
  defined 41
  setting 40, 87
adapters
  graphics 53
  monochrome 49, 82
allocmem function 106
arc function 49
  coordinates 57
arcs, elliptical 55
aspect ratio
  correcting 89
  determining current 47
  getting 58
  setting 40
assembly language
  inline, floating point in 29
  routines, overlays and 24
AT&T 6300 PC
  detecting presence of 53
attributes, screen cells 35
autodetection (graphics drivers) 53, 60, 74, 77
AX register 4
  hardware error handlers and 123

B

banker's rounding 33
bar function 50
bar3d function 50
bars
  three-dimensional 50
  two-dimensional 50
base address register 5
baud rate 108, 116
bcd class 31
  converting 32
  number of decimal digits 32
  output 32
  range 32
  rounding errors and 32
beeps 127, 131
BGI *See* Borland Graphics Interface
BGIOBJ (graphics converter) 75
  initgraph function and 39
  stroked fonts and 99
BIOS functions (list) 135
BIOS interrupts
  0x13 109
  0x16 113
  0x17 115
_bios_disk function 111
_bios_keybrd function 113
_bios_printer function 115
_bios_serialcom function 116
bioscom function 107
biosdisk function 109
bioskey function 112
biosprint function 114
bit images, functions for 40
  saving 62
  storage requirements 74
  writing to screen 83
bits
  status 108, 117
  stop 108, 116
Borland Graphics Interface (BGI)
  device driver table 77
  fonts 84
    new 78
  graphics drivers and 73, 74, 85
BP register 5
  hardware error handlers and 123
  overlays and 24
break value 118, 130
brk function 118
buffers
  graphics, internal 93
  overlays, default size 23, 138
BX register 4
bytes, status (disk drives) 110

C

CGA *See* Color/Graphics Adapter
characters
  in screen cells 35
  magnification, user-defined 99
  size 102
.CHR files 78
circle function 51
circles
  drawing 51
  roundness of 40
_clear87 function, floating point exceptions and 30
cleardevice function 51
clearing screens 40, 51, 93
clearviewport function 52
clipping, defined 42
closegraph function 52
code segment 6
Color/Graphics Adapter (CGA)
  background and foreground colors 83
  color palettes 44, 45
  detecting presence of 53

problems 49, 82
resolution 44
high 45
colors and palettes 71, 87
background color 58, 89
setting 71, 89
changing 87, 95
color table 88, 89, 96
default 59
definition structure 59
drawing 59, 71, 82, 90
fill colors 56, 57
information on 61
pie slices 82, 86
setting 92
fill patterns 56, 57, 71
defining 60, 61
by user 91, 92
information on 61
pie slices 82, 86
predefined 61
setting to default 71
filling graphics 57
IBM 8514 96
information on 67
returning 59
maximum value 64
pixels 68, 83
problems with 49, 82
rectangle 84
setting 90, 96
background 89
drawing 71
size of 68
VGA 96
.COM files, memory models and 10
communications
parity 108, 116
ports 107, 116
protocol settings 108, 116
RS-232 107, 116
stop bits 108, 116
compiler options
code segment 15
data segment 15
far objects 15
floating point 28
overlays 22, 23
complex numbers 30
header file 30
overloading operators and 30
complex.h 30
_control87 function, floating point exceptions and 30
control-break interrupt 125
conversions, bcd 32
coordinates,
arc, returning 57
current position 70, 100
origin 36
starting positions 36
x-coordinate 65, 70
maximum 65
y-coordinate 66, 70
maximum 66
coprocessors *See* numeric coprocessors
coreleft function 118
correction factor of aspect ratio 89
_cs keyword 14
CS register 6, 8
current position (graphics) 71
coordinates 70, 100
justified text and 97
lines and 79
moving 80
CX register 4

D

data bits 108, 116
data segments 6, 137
allocation 118
changing 130
naming and renaming 15
debugging, overlays 23
delay function 119
detectgraph function 53
detection
graphics adapter 53, 60
graphics drivers 74
device drivers 77
device errors 123, 124
DI register 5
hardware error handlers and 123
direct memory access (DMA)
checking for presence of 110
disk drives
functions 109
I/O operations 109
status byte 110
disk sectors
reading 105, 109, 111
writing 106, 109, 112
disk transfer address (DTA)
DOS 128
random blocks and 128
disks
errors 123, 124
operations 109
DOS environment
87 variable 29
DOS functions 126
list 135
DOS interrupts
0x23 123, 125
0x24 123, 124
0x25 105
0x26 106
handlers 124, 125
DOS system calls 124, 125
0x27 128
0x28 129
0x48 107
_dos_freemem function 122
_dos_keep function 126
DMA *See* direct memory access
drawing functions 39
drawpoly function 55
_ds keyword 14
DS register 6, 8
DTA *See* disk transfer address
DX register 4

E

EGA *See* Enhanced Graphics Adapter
ellipse function 55
ellipses, drawing and filling 56
elliptical arcs 55
elliptical pie slices 86
Enhanced Graphics Adapter (EGA) 45
color control on 45
detecting presence of 53
environment, 87 variable (DOS) 29
error handlers, hardware 123, 124, 125
error messages
graphics 46, 73
returning 71
pointer to, returning 71
errors
floating-point, disabling 30
graphics, functions for handling 46
math, masking 30
out of memory 3
_es keyword 14
ES register 6
even parity 108, 116
exception handling 21
execution, suspending 119
exit status 126

extended and expanded memory
    overlays and 24
extra segment 6

F

-f compiler option 28
-f87 compiler option 28
far calls 23
far heap
    checking 120
        nodes 121
    free blocks 120, 121
    unused memory 119
    walking through 122
far keyword 7, 14, 18
farcoreleft function 119
farheapcheck function 120
farheapcheckfree function 120
farheapchecknode function 121
farheapfillfree function 121
farheapwalk function 122
FCB *See* file control block
-ff compiler option 28
fields, random record 128, 129
file control block (FCB) 128, 129
files
    font 78
    graphics driver 75
        linking 38
    project
        graphics library listed
            in 36
fill style (graphics) 71
fillellipse function 56
filling functions 39
fillpoly function 56
financial applications 31
flags, register 4, 5
floating point 27
    emulating 28
    exceptions, disabling 30
    fast 28
    formats 27
    I/O 27
    libraries 27
    registers and 29
floodfill function 57
fonts 98
    bit-mapped 42
        stroked vs. 42
    character size 102
    characteristics 98
    clipping 42
    files, loading and
        registering 42
    graphics text 71, 98
        information on 69
    height 42, 102
    ID numbers 78
    information on current
        settings 48
    linked-in 85
    multiple 81, 99
    new 78
    registering 43
        sans-serif 85
    setting size 42, 85
        small 85
    stroked
        advantages of 42
        fine tuning 99
        linked-in code 84
        maximum number 78
        multiple 99
    text 81
    triplex 85
    width 42, 102
FP_OFF 17
FP_SEG 17
freemem function 122
functions
    See also specific function
    8086 135
    BIOS 135
    color control 44
    declaring as near or far 16
    drawing 39
    error-handling, graphics 46
    far 16
    filling 39
    goto 136
    graphics 37-48, 134
        drawing operations 39
        fill operations 39
        system control 37
    image manipulation 40
    international information 136
    list 120
    locale 136
    memory, allocating and
        checking 135
    near 16
    operating system 135
    pixel manipulation 41
    pointers, overlaid routines 23
    recursive, memory models
        and 16
    screen manipulation 40
    sound 136
    state queries 47
    text
        output, graphics mode 41
        viewport manipulation 40

G

getarccoords function 57
getaspectratio function 58
getbkcolor function 58
getcolor function 59
getdefaultpalette function 59
getdrivername function 60
getfillpattern function 60
getfillsettings function 61
getgraphmode function 62
getimage function 62
getlinesettings function 63
getmaxcolor function 64
getmaxmode function 65
getmaxx function 65
getmaxy function 66
getmodename function 66
getmoderange function 67
getpalette function 67
getpalettesize function 68
getpixel function 68
gettextsettings function 69
getviewsettings function 69
getx function 70
gety function 70
global variables
    heap size 137
    overlay buffer, changing 20,
        23, 138
    stack size 139
gothic fonts 98
goto statements
    functions (list) 136
graphdefaults function 71
grapherrormsg function 71
_graphgetmem function 72
graphics
    active page 101
        setting 87
    arcs 49
        coordinates of 57
        elliptical 55
    aspect ratio
        correcting 89
        getting 58
    bars 50
    circles
        aspect ratio 40
        drawing 51
    colors
        background 45
            defined 44
        CGA 44, 45
        drawing 44
        EGA/VGA 45
        foreground 45

functions 43
information on current
settings 48
default settings 59, 71
restoring 38
displaying 44
drawing functions 39
ellipses 56
error messages 71
error-handling 46
fill operations 39
fill patterns 40, 47
functions 36-58
justifying text for 97
list 134
header file 36
I/O 100
library 36
memory and 72
line style 40
memory 39
allocation of memory
from 72
freeing 72
pages 41, 101
setting 40, 87
palettes
defined 43
functions 43
information on current 48
pie slices 82, 86
pixels
colors 68, 83
current 48
functions for 41
setting color of 43
polygons 55, 56
rectangle 84
screens, clearing 41, 51
settings 41
default 59, 71
state queries 47
system
control functions 37
initialization 74
shutting down 38, 52
state queries 47
text and 41
viewports 71
clearing 52
defined 36
displaying strings in 81
functions 40
information 69
on current 48
setting for output 100
visual page 101
graphics adapters 53
problems with 49, 82
graphics buffers 41
internal 93
graphics drivers
BGI and 73, 74, 85
device driver table 77
current 38, 47, 60
returning information
on 48
detecting 53, 60, 74, 77
error messages 73
file 75
initialization 74
linking 38
loading 38, 74, 85
modes 86
maximum number for
current driver 65
names 66
overriding 53
range 67
returning 62
setting 93
switching 93
new 77
adding 38
registering 39, 85
returning information on 47,
48
selecting 38
supported 38
vendor added 77
graphics.h 36
graphresult function 73

H

handlers, interrupt (DOS) 124,
125
_harderr function 124
harderr function 123
_hardresume function 125
hardresume function 123
_hardretn function 125
hardretn function 123
hardware
error handlers 123, 124, 125
printer ports 114, 115
header files
complex numbers 30
graphics 36
heap 119
length 137
near, size of 137
_heaplen global variable 137
Hercules card
detecting presence of 53
huge keyword 7, 14

I

I/O
disk 109
floating-point numbers 27
graphics 100
serial 107, 116
IBM 3270 PC
detecting presence of 53
IBM 8514
colors, setting 96
detecting presence of 53
IDs (fonts) 78
IEEE, rounding 33
imagesize function 74
initgraph function 74
initialization, graphics system 74
installuserdriver function 77
installuserfont function 78
internal graphics buffer 93
international information
functions (list) 136
interrupts
control-break 123, 125
handlers
DOS 124, 125
modules and 23
IP (instruction pointer) register 4

J

justifying text for graphics
functions 97

K

keep function 126
keyboard operations 113

L

libraries
files (list) 133
floating-point 27
graphics 36, 72
selecting 133
summary 134
line function 79
linerel function 79
lines
drawing
between points 79
from current position 79
mode 101
relative to current
position 79
pattern of 63

rectangles and 84
style of 63, 94
thickness of 63, 94
lineto function 79
linked-in fonts 85
linked-in graphics drivers code 85
linker 18
locale functions (list) 136

M

macros, far pointers 17
math errors, masking 30
_matherr function 30
maximum color value 64
MCGA
detecting presence of 53
memory 3
access (DMA) 110
allocation 106
data segment 118
changing 130
freeing 122
functions (list) 135
graphics 39, 72
unused 118, 119
bit images 74
saving to 62
checking 135
direct access (DMA) 110
expanded and extended 127, 128
freeing
in DOS memory 122
in graphics memory 72
memory models and 10
overlays and 20
paragraphs 7
boundary 7
segments in 6
memory addresses
calculating 5, 7
far pointers and 8
near pointers and 8
pointing to 17
segment, offset notation 7
standard notation for 7
memory blocks
adjusting size of 130
file control 128, 129
free 120
filling 121
random
reading 128
writing 129
memory models 3-19
changing 17
compact 10
comparison 13
defined 9
functions list 135
graphics library 37
huge 10
large 10
libraries 133
math files for 133
medium 10
memory apportionment and 10
mixing 18
overlays and 21, 23
pointers and 7, 14
program segment prefix and 138, 139
size of near heap 137
small 10
stack size 139
supported 9
tiny 9
restrictions 119
mixed modules, linking 18
MK_FP macro 17
modifiers (pointers) 14
modules
linking mixed 18
size limit 13
monochrome adapters 53
graphics problems 49, 82
moverel function 80
moveto function 80

N

near heap 137
near keyword 7, 14
negative offsets 5
no parity 108, 116
nosound function 127
numeric coprocessors
autodetecting 29
built-in 28
floating-point emulation 28
registers and 29

O

.OBJ files, converting .BGI files to 39
objects, far
class names 15
combining into one segment 15
declaring 15
option pragma and 15
odd parity 108, 116
offsets 8
component of a pointer 17
option pragma, far objects and 15
out of memory errors 3
outtext function 81
outtextxy function 81
_OvrInitEms function 127
overlays 19-25
assembly language routines and 24
BP register and 24
buffers, default size 23, 138
cautions 23
command-line option 22
debugging 23
designing programs for 23
expanded and extended memory and 24, 127, 128
large programs 19
linking 21
errors 21, 22
memory map 20
memory models and 21, 23
routines, calling via function pointers 23
overloaded operators
complex numbers and 30
_ovrbuffer global variable 20, 23, 138
_OvrInitExt function 128

P

pages
active 87
defined 41
setting 40
buffers 41
visual 41
numbers, setting 101
setting 40
palettes *See* colors and palettes
parity 108, 116
pause (suspended execution) 119
PC speaker 127, 131
pie slices 82
elliptical 86
pieslice function 82
pixels, graphics color 68, 83
pointers
arithmetic 8

changing memory models
and 17
comparing 8
default data 13
error messages 71
far 8
comparing 8
declaring 8-17
function prototypes
and 17
far memory model and 7
huge 9
comparing 9
declaring 8-17
overhead 9
huge memory model and 7
manipulating 8
memory addresses 17
memory models and 7, 14
modifiers 14
near 8
declaring 8-17
function prototypes
and 17
near memory model and 7
normalized 9
overlays and 23
segment 14
stack 5
polygons
drawing 55, 56
filling 56
ports
communications 107, 116
printer 114, 115
positive offsets 5
#pragma directives
far objects and 15
printers
ports 114, 115
printing directly 114, 115
profilers 23, 138
program segment prefix (PSP)
memory models and 138, 139
programs
large, overlaying 19
RAM resident 126
stopping
exit status 126
suspended execution 119
TSR 126
project files
graphics library listed in 36
protocol settings
(communications) 108, 116
prototypes
far and near pointers and 17
mixing modules and 18
PSP *See* program segment prefix
putimage function 83
putpixel function 83

R

RAM 3
unused 118
RAM resident programs 126
randbrd function 128
randbwr function 129
random block read 128
random block write 129
records, random fields 128, 129
rectangle function 84
recursive functions
memory models and 16
registerbgidriver function 85
registerbgifont function 84
registers 123
8086 4-6
AX 4
base point 5
BP 5
overlays and 24
BX 4
CS 6, 8
CX 4
DI 5
DS 6, 8
DX 4
ES 6
flags 4, 5
index 4, 5
IP (instruction pointer) 4
LOOP and string
instruction 4
math operations 4
numeric coprocessors and 29
segment 5, 6
SI 5
SP 5
special-purpose 5
SS 6
restorecrtmode function 86
restoring screen 86
rounding
banker's 33
errors 31
RS-232 communications 107, 116

S

sbrk function 130
scaling factor (graphics) 40
screens
aspect ratio 40
correcting 89
getting 58
cells, characters in 35
clearing 40, 51, 93
colors 43
coordinates 36
starting positions 36
modes
defining 35
graphics 35, 36, 38
restoring 86
selecting 38
text 35, 38
resolution 35
x-coordinate
maximum 65
y-coordinate
maximum 66
sector function 86
_seg keyword 14
segment prefix (program) 138,
139
segmented memory
architecture 6
segments 7, 10
component of a pointer 17
memory 6
offset address notation 7
making far pointers
from 17
pointers 14
registers 5, 6
serial communications 107, 116
setactivepage function 87
setallpalette function 87
setaspectratio function 89
setbkcolor function 45, 89
setblock function 130
setcolor function 90
setfillpattern function 91
setfillstyle function 92
setgraphbufsize function 93
setgraphmode function 93
setlinestyle function 94
setpalette function 95
setrgbpalette function 96
settextjustify function 97
settextstyle function 98
settings, graphics
clearing screen and 41
default 59, 71
restoring 38
setusercharsize function 99
setviewport function 100
setvisualpage function 101
setwritemode function 101
SI register 5

size
  color palette 68
  stack 139
small fonts 85
sound function 131
sounds
  functions (list) 136
  turning off 127
  turning on 131
SP register 5
  hardware error handlers and 123
speaker
  turning off 127
  turning on 131
special-purpose registers (8086) 5
_ss keyword 14
SS register 6
stack 119
  length 139
  pointers 5
  segment 6
  size 139
state queries 47-48
status bits 108, 117
status byte 110
_status87 function, floating point exceptions and 30
_stklen global variable 139
stop bits 108, 116
string instructions, registers 4
strings
  clipping 42
  displaying 81
  height, returning 102
  width, returning 102
structures
  complex 30
  graphics palette definition 59
suspended program execution 119
system, graphics
  controlling 37
  initializing 74
  shutting down 38, 52
  state queries 47

T

terminate-and-stay resident programs 126
text
  characteristics 98
  in graphics mode 41
  information on current settings 48
  justifying 42, 97
text strings
  clipping 42
  displaying 81
  size 42
textheight function 102
textwidth function 102
three-dimensional bars 50
time, delays in program execution 119
TLINK 18
TSR programs 126
two-dimensional bars 50

U

user-defined characters (fonts) 99
user-defined fill patterns 91, 92
user-loaded graphics driver code 85
UTIL.DOC 43

V

values, break 118, 130
vendor-added device driver 77
VGA *See* Video Graphics Array Adapter
video adapters 35-48
  graphics, compatible 38
  modes 35
Video Graphics Array Adapter (VGA) 45
  detecting presence of 53
visual page
  defined 41
  numbers setting 101
  setting 40
VROOMM 19

W

warning beeps 127, 131
width, strings, returning 102
window function 36
windows 36
  default type 36

X

x aspect factor 58
x-coordinate 65, 70

Y

-Y compiler option 23
y aspect factor 58
y-coordinate 66, 70
-Yo compiler option 22

Z

-zX compiler option 15

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1245WW21779 • BOR 7777