Turbo C®

Reference Guide

Version 2.0

Copyright© 1988
All rights reserved

All Borland products are trademarks or registered trademarks of
Borland International, Inc. Other brand and product names are trademarks or registered
trademarks of their respective holders.
Copyright© 1988 Borland International.

This manual was produced with
Sprint® The Professional Word Processor

Table of Contents

Introduction 1
Volume II: The Reference Guide 1
Typographic Conventions 3
Borland's No-Nonsense License Statement 3
Acknowledgments 4
How to Contact Borland 4

Chapter 1 Using Turbo C Library Routines 5
In This Chapter 5
The Library Routine Lookup Section 6
Why You Might Want to Access the Turbo C Run-Time Library Source Code 7
The Turbo C Include Files 8
Library Routines by Category 10
The main Function 17
  The Arguments to main 17
    An Example Program Using argc, argv and env 18
    Wildcard Command-Line Arguments to main 19
  When You Compile Using –p (Pascal Calling Conventions) 20
  The Value main Returns 21
Global Variables 22
_argc 22
_argv 22
daylight 22
directvideo 23
_8087 23
environ 24
errno, _doserrno, sys_errlist, sys_nerr 24
_fmode 27
_heaplen 28
_osmajor, _osminor 29
_psp 29
_stklen 30
timezone 31
tzname 31
_version 31

Chapter 2 The Turbo C Library 33

function name 33
abort 35
abs 35
absread 36
abswrite 37
access 37
acos 39
allocmem 39
arc 40
asctime 43
asin 44
assert 45
atan 46
atan2 46
atexit 47
atof 48
atoi 49
atol 50
bar 50
bar3d 51
bdos 52
bdosptr 53
bioscom 54
biosdisk 57
biosequip 60
bioskey 61
biosmemory 63
biosprint 64
biostime 64
brk 65
bsearch 66
cabs 67
calloc 68
ceil 69
cgets 69
chdir 71
_chmod 71
chmod 72
chsize 73
circle 74
_clear87 75
cleardevice 75

clearerr 76
clearviewport 76
clock 77
_close 78
close 78
closegraph 79
clreol 79
clrscr 80
_control87 80
coreleft 82
cos 82
cosh 83
country 83
cprintf 85
cputs 85
_creat 86
creat 87
creatnew 88
creattemp 89
cscanf 90
ctime 91
ctrlbrk 91
delay 93
delline 93
detectgraph 94
difftime 96
disable 96
div 97
dosexterr 98
dostounix 98
drawpoly 99
dup 100
dup2 101
ecvt 101
ellipse 102
__emit__ 103
enable 105
eof 105
exec... 106
_exit 109
exit 110
exp 110
fabs 111

farcalloc 111
farcoreleft 112
farfree 113
farmalloc 113
farrealloc 115
fclose 116
fcloseall 116
fcvt 117
fdopen 117
feof 119
ferror 120
fflush 120
fgetc 121
fgetchar 121
fgetpos 122
fgets 122
filelength 123
fileno 123
fillellipse 124
fillpoly 124
findfirst 125
findnext 127
floodfill 127
floor 129
flushall 129
fmod 130
fnmerge 130
fnsplit 132
fopen 134
FP_OFF 136
_fpreset 136
fprintf 137
FP_SEG 138
fputc 138
fputchar 139
fputs 139
fread 140
free 140
freemem 141
freopen 141
frexp 142
fscanf 143
fseek 144

fsetpos 145
fstat 146
ftell 147
ftime 148
fwrite 149
gcvt 149
geninterrupt 150
getarccoords 150
getaspectratio 151
getbkcolor 152
getc 153
getcbrk 153
getch 154
getchar 154
getche 155
getcolor 155
getcurdir 156
getcwd 157
getdate 158
getdefaultpalette 159
getdfree 159
getdisk 160
getdrivername 160
getdta 161
getenv 161
getfat 163
getfatd 163
getfillpattern 164
getfillsettings 165
getftime 167
getgraphmode 168
getimage 169
getlinesettings 170
getmaxcolor 172
getmaxmode 172
getmaxx 173
getmaxy 173
getmodename 174
getmoderange 175
getpalette 175
getpalettesize 177
getpass 178
getpixel 178

getpsp 179
gets 179
gettext 180
gettextinfo 181
gettextsettings 182
gettime 183
getvect 184
getverify 185
getviewsettings 186
getw 187
getx 187
gety 188
gmtime 188
gotoxy 190
graphdefaults 191
grapherrormsg 191
_graphfreemem 192
_graphgetmem 193
graphresult 194
harderr 196
hardresume 198
hardretn 198
highvideo 199
hypot 199
imagesize 200
initgraph 201
inport 205
inportb 206
insline 206
installuserdriver 207
installuserfont 209
int86 209
int86x 211
intdos 212
intdosx 213
intr 215
ioctl 216
isalnum 218
isalpha 218
isascii 219
isatty 219
iscntrl 220
isdigit 220

isgraph 221
islower 221
isprint 222
ispunct 222
isspace 223
isupper 223
isxdigit 224
itoa 224
kbhit 225
keep 225
labs 226
ldexp 226
ldiv 227
lfind 228
line 228
linerel 229
lineto 229
localtime 230
lock 231
log 232
log10 232
longjmp 233
lowvideo 234
_lrotl 235
_lrotr 235
lsearch 236
lseek 238
ltoa 238
malloc 239
_matherr 241
matherr 242
max 245
memccpy 246
memchr 247
memcmp 247
memcpy 248
memicmp 248
memmove 249
memset 249
min 250
mkdir 250
MK_FP 251
mktemp 251

modf . . . 252
movedata . . . 252
moverel . . . 253
movetext . . . 253
moveto . . . 254
movmem . . . 254
normvideo . . . 255
nosound . . . 255
_open . . . 256
open . . . 257
outport . . . 259
outportb . . . 259
outtext . . . 260
outtextxy . . . 261
parsfnm . . . 261
peek . . . 262
peekb . . . 262
perror . . . 263
pieslice . . . 264
poke . . . 265
pokeb . . . 265
poly . . . 266
pow . . . 266
pow10 . . . 267
printf . . . 267
putc . . . 279
putch . . . 280
putchar . . . 280
putenv . . . 281
putimage . . . 281
putpixel . . . 282
puts . . . 283
puttext . . . 283
putw . . . 284
qsort . . . 284
raise . . . 286
rand . . . 287
randbrd . . . 288
randbwr . . . 288
random . . . 289
randomize . . . 290
_read . . . 290
read . . . 291

realloc 292
rectangle 293
registerbgidriver 293
registerbgifont 294
remove 295
rename 296
restorecrtmode 296
rewind 297
rmdir 297
_rotl 298
_rotr 299
sbrk 299
scanf 300
searchpath 310
sector 311
segread 312
setactivepage 312
setallpalette 313
setaspectratio 315
setbkcolor 315
setblock 316
setbuf 317
setcbrk 318
setcolor 318
setdate 320
setdisk 320
setdta 321
setfillpattern 321
setfillstyle 322
setftime 323
setgraphbufsize 324
setgraphmode 325
setjmp 325
setlinestyle 326
setmem 328
setmode 328
setpalette 329
setrgbpalette 330
settextjustify 331
settextstyle 332
settime 334
setusercharsize 334
setvbuf 336

setvect 338
setverify 338
setviewport 339
setvisualpage 339
setwritemode 340
signal 341
sin 347
sinh 347
sleep 348
sopen 348
sound 350
spawn... 352
sprintf 357
sqrt 357
srand 358
sscanf 358
stat 359
_status87 361
stime 361
stpcpy 362
strcat 362
strchr 362
strcmp 363
strcmpi 364
strcpy 364
strcspn 365
strdup 365
_strerror 365
strerror 366
stricmp 367
strlen 367
strlwr 368
strncat 368
strncmp 368
strncmpi 369
strncpy 370
strnicmp 370
strnset 371
strpbrk 371
strrchr 372
strrev 372
strset 372
strspn 373

strstr ... 373
strtod ... 374
strtok ... 375
strtol ... 376
strtoul ... 377
strupr ... 378
swab ... 378
system ... 379
tan ... 379
tanh ... 380
tell ... 380
textattr ... 381
textbackground ... 383
textcolor ... 384
textheight ... 386
textmode ... 386
textwidth ... 388
time ... 388
tmpfile ... 389
tmpnam ... 389
toascii ... 390
_tolower ... 390
tolower ... 391
_toupper ... 391
toupper ... 392
tzset ... 392
ultoa ... 394
ungetc ... 394
ungetch ... 395
unixtodos ... 395
unlink ... 396
unlock ... 396
va_... ... 397
vfprintf ... 399
vfscanf ... 400
vprintf ... 401
vscanf ... 402
vsprintf ... 403
vsscanf ... 404
wherex ... 405
wherey ... 405
window ... 406
_write ... 406

write 407

Appendix A The Turbo C Interactive Editor 409
Introduction 409
Turbo In, Turbo Out 409
The Edit Window Status Line 410
Editor Commands 411
Basic Cursor Movement Commands 413
Quick Cursor Movement Commands 414
Insert and Delete Commands 415
Block Commands 416
Miscellaneous Editing Commands 417
The Turbo C Editor Vs. WordStar 421

Appendix B Compiler Error Messages 423
Fatal Errors 424
Errors 424
Warnings 437

Appendix C TCC Command-Line Options 443
Turning Options On and Off 445
Syntax 445
Compiler Options 446
Memory Model 447
#defines 447
Code Generation Options 448
Optimization Options 449
Source Code Options 452
Error-Reporting Options 452
Segment-Naming Control 454
Compilation Control Options 455
Linker Options 455
Environment Options 455
Implicit *vs.* User-specified Library Files 457
The Include and Library File-Search Algorithms 457
Using –L and –I in Configuration Files 458
An Example With Notes 458

Appendix D Turbo C Utilities 461
CPP: The Turbo C Preprocessor Utility 461
CPP as a Macro Preprocessor 462
An Example 463
The Standalone MAKE Utility 463
A Quick Example 464

Creating a Makefile 466
Using a Makefile 467
Stepping Through 467
Creating Makefiles 468
Components of a Makefile 469
Comments 469
Explicit Rules 470
Special Considerations 471
Examples 471
Implicit Rules 472
Special Considerations 475
Examples 475
Command Lists 476
Prefix 476
Command body 477
Examples 478
Macros 478
Defining Macros 479
Using Macros 480
Special Considerations 480
Predefined Macros 480
Directives 483
File-Inclusion Directive 483
Conditional Execution Directives 484
Error Detection Directive 486
Macro Undefinition Directive 487
Using MAKE 487
Command-Line Syntax 487
A Note About Stopping MAKE 488
The BUILTINS.MAK File 488
How MAKE Searches for BUILTINS.MAK and Makefiles 489
MAKE Command-line Options 489
MAKE Error Messages 490
Fatal Error Messages 490
Errors 491
The TOUCH Utility 493
Turbo Link 493
Invoking TLINK 493
Using Response Files 495
Using TLINK with Turbo C Modules 496
Initialization Modules 497
Libraries 497
Using TLINK with TCC 498

TLINK Options 499
The /x, /m, /s Options 499
The /l Option 501
The /i Option 501
The /n Option 501
The /c Option 501
The /d Option 502
The /e Option 502
The /t Option 503
The /v Option 503
The /3 Option 503
Restrictions 504
Error Messages 504
Fatal Errors 504
Nonfatal Errors 507
Warnings 507
TLIB: The Turbo Librarian 508
The Advantages of Using Object Module Libraries 509
The Components of a TLIB Command Line 509
The Operation List 510
File and Module Names 511
TLIB Operations 511
Creating a Library 512
Using Response Files 513
Creating an Extended Dictionary: The /E Option 513
Advanced Operation: The /C Option 514
Examples 514
GREP: A File-Search Utility 515
The GREP Options 515
Order of Precedence 517
The Search String 517
Operators in Regular Expressions 518
The File Specification 519
Examples with Notes 519
BGIOBJ: Conversion Utility for Graphics Drivers and Fonts 522
Adding the New .OBJ Files to GRAPHICS.LIB 523
Registering the Drivers and Fonts 523
The /F option 525
Advanced BGIOBJ Features 526
OBJXREF: The Object Module Cross-Reference Utility 528
The OBJXREF Command Line 529
The OBJXREF Command-Line Options 530
Control Options 530

Report Options 530
Response Files 531
Freeform Response Files 531
Project Files 532
Linker Response Files 532
The /O Command 532
The /N Command 533
Sample OBJXREF Reports 533
Report by Public Names (/RP) 534
Report by Module (/RM) 534
Report by Reference (/RR) (Default) 535
Report by External References (/RX) 535
Report of Module Sizes (/RS) 536
Report by Class Type (/RC) 536
Report of Unreferenced Symbol Names (/RU) 537
Verbose Reporting (/RV) 537
Examples of How to Use OBJXREF 537
OBJXREF Error Messages and Warnings 539
Error Messages 539
Warnings 539

Appendix E Language Syntax Summary 541
Lexical Grammar 541
Tokens 541
Keywords 542
Identifiers 542
Constants 542
String Literals 544
Operators 545
Punctuators 545
Phrase Structure Grammar 545
Expressions 545
Declarations 548
Statements 551
External Definitions 552
Preprocessing Directives 552

Appendix F TCINST: Customizing Turbo C 555
Running TCINST 556
The TCINST Installation Menu 557
The Compile Menu 558
The Project Menu 558
Project Name 558
The Break Make On Menu 558

Auto Dependencies 559
Clear Project 559
The Options Menu 559
The Compiler Menu 559
Model 559
Defines 559
The Code Generation Menu 559
The Optimization Menu 560
The Source Menu 560
The Errors Menu 560
The Names Menu 561
The Linker Menu 561
Map File 561
Initialize Segments 561
Default Libraries 561
Graphics Library 562
Warn Duplicate Symbols 562
Stack Warning 562
Case-Sensitive Link 562
The Environment Menu 562
Message Tracking 562
Keep Messages 563
Config Auto Save 563
Edit Auto Save 563
Backup Source Files 563
Zoomed Windows 563
Full Graphics Save 563
The Screen Size Menu 563
The Options for Editor Menu 564
The Directories Menu 565
Include Directories 565
Library Directories 566
Output Directory 566
Turbo C Directory 566
Pick File Name 566
Arguments 567
The Debug Menu 567
Source Debugging 567
Display Swapping 567
The Editor Commands Option 567
Allowed Keystrokes 571
The Mode for Display Menu 572
The Set Colors Menu 573

Resize Windows 574
Quitting the Program 574

Appendix G MicroCalc 577
About MicroCalc 577
How to Compile and Run MicroCalc 578
With TC.EXE 578
With TCC.EXE 578
How to use MicroCalc 579
The MicroCalc Parser 582

Index 583

List of Figures

Figure F.1: The TCINST Installation Menu 557

List of Tables

Table A.1: Summary of Editor Commands 412
Table C.1: Correlation of Command-Line Options and Menu Selections . 444

INTRODUCTION

This is the second volume of documentation in the Turbo C package. This volume, the *Turbo C Reference Guide*, contains definitions of all the Turbo C library routines, common variables, and common defined types, along with example program code to illustrate how to use many of these routines, variables, and types.

If you are new to C programming, you should first read the other book in your Turbo C package—the *Turbo C User's Guide*. In that book you'll find instructions on how to install Turbo C on your system, an overview of Turbo C's window and menu system, and tutorial-style chapters designed to get you started programming in Turbo C. The user's guide also summarizes Turbo C's implementation of the C language and discusses some advanced programming techniques. For those of you who are Turbo Pascal and Turbo Prolog programmers, the user's guide provides information to help you integrate your understanding of those languages with your new knowledge of C.

You should refer to the "Introduction" in the *User's Guide* for information on the Turbo C implementation, a summary of the contents of Volume I, and a short bibliography.

Volume II: The Reference Guide

The *Turbo C Reference Guide* is written for experienced C programmers; it provides implementation-specific details about the language and the run-time environment. In addition, it provides definitions for each of the Turbo C functions, listed in alphabetical order.

These are the chapters and appendixes in the programmer's reference guide:

Chapter 1: Using Turbo C Library Routines summarizes Turbo C's input/output (I/O) support, lists and describes the #include (.h) files, and lists the Turbo C library routines by category. Then it explains the Turbo C **main** function and its arguments, and concludes with a lookup section describing each of the Turbo C global variables.

Chapter 2: The Turbo C Library is an alphabetical reference of all Turbo C library functions. For each function it gives the function prototype, the include file(s) containing the prototype, an operative description of what the function does, return values, portability information, and a list of related functions.

Appendix A: The Turbo C Interactive Editor gives a more thorough explanation of the editor commands—for those who need more information than that given in Chapter 5 of the *Turbo C User's Guide*.

Appendix B: Compiler Error Messages lists and explains each of the error messages and summarizes the possible or probable causes of the problem that generated the message.

Appendix C: Options describes each of the Turbo C user-selectable compiler options.

Appendix D: Turbo C Utilities discusses the standalone MAKE utility, the CPP preprocessor, the Turbo Linker TLINK, TLIB the Turbo Librarian, the file-searching utility GREP, BGIOBJ, a conversion utility for graphics drivers and fonts, and the object module cross-referencer OBJXREF.

Appendix E: Language Syntax Summary uses modified Backus-Naur Forms to detail the syntax of all Turbo C constructs.

Appendix F: Customizing Turbo C guides you through the customization program (TCINST), which lets you customize your keyboard, modify default values, change your screen colors, resize your Turbo C windows, and more.

Appendix G: MicroCalc introduces the spreadsheet program included with your Turbo C package and gives directions for compiling and running the program.

Typographic Conventions

All typefaces used in this manual were produced by Borland's Sprint: The Professional Word Processor on an Apple LaserWriter Plus. Their special uses are as follows:

`Monospaced type`	This typeface represents text as it appears on the screen or in your program and anything you must type (such as command-line options).
[ ]	Square brackets in text or DOS command lines enclose optional input or data that depends on your system, which should not be typed verbatim.
< >	Angle brackets in the function reference section enclose the names of include files.
Boldface	Turbo C function names (such as **printf**) are shown in boldface when mentioned within text (but not in program examples). This typeface represents Turbo C keywords (such as **char**, **switch**, **near**, and **cdecl**).
Italics	Italics indicate variable names (identifiers) within sections of text and to emphasize certain words (especially new terms).
Keycaps	This special typeface indicates a key on your keyboard. It is often used when describing a particular key you should type; for example, "Press *Esc* to cancel a menu."

Borland's No-Nonsense License Statement

This software is protected by both United States Copyright Law and International Treaty provisions. Therefore, you must treat this software *just like a book* with the following single exception: Borland International authorizes you to make archival copies of Turbo C for the sole purpose of backing up your software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of its being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

Acknowledgments

In this manual, we refer to several products:

- Turbo Pascal, Turbo Prolog and Sprint: The Professional Word Processor are registered trademarks of Borland International, Inc.
- WordStar is a trademark of MicroPro, Inc.
- IBM PC, XT, and AT are trademarks of International Business Machines, Inc.
- MS-DOS is a registered trademark of Microsoft Corporation.
- UNIX is a registered trademark of American Telephone and Telegraph.

How to Contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type `GO BOR` from the main CompuServe menu and choose "Borland Programming Forum B (Turbo Prolog & Turbo C)" from the Borland main menu. Leave your questions or comments there for the support staff to process.

Technical Support Department

INTERNATIONAL HEADQUARTERS :
BORLAND INTERNATIONAL
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001
U.S.A.
Tél. : (1) (408) 438-8400
Telex : 172373
Fax : (1) (408) 438-8696

EUROPEAN HEADQUARTERS :
BORLAND INTERNATIONAL FRANCE
43, Avenue de l'Europe
B.P. 6.
78141 VELIZY CEDEX
FRANCE
Tél. : (33) (1) 39-46-96-69
Telex : 698793
Fax : (33) (1) 39-46-81-60

U.K. OFFICES :
BORLAND INTERNATIONAL (U.K.) LTD
8 Pavilions
Ruscombe Business Park
TWYFORD, BERKSHIRE RG10 9NN
UNITED KINGDOM
Tel. : 0734-320022
Telex : 846616
Fax : 0734-320017

Please have the following information handy before you call:

- product name and version number
- computer make and model number
- operating system and version number

C H A P T E R

1

Using Turbo C Library Routines

Turbo C comes equipped with over 450 library routines—functions and macros that you call from within your C programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more.

Turbo C's routines are contained in the library files (Cx.LIB, MATHx.LIB, and GRAPHICS.LIB). Because Turbo C supports six distinct memory models, each model except the tiny model has its own library file and math file, containing versions of the routines written for that particular model. (The tiny model shares the small library and math files.)

Turbo C supports the draft ANSI C standard which, among other things, allows function prototypes to be given for the routines in your C programs. All of Turbo C's library routines are declared with prototypes in one or more header files (these are the .h or "include" files that were copied from the distribution disks into your INCLUDE directory during installation).

In This Chapter

This first part of the *Turbo C Reference Guide* provides an overview of the Turbo C library routines and include files.

In this chapter, we

- explain why you might want to obtain the source code for the Turbo C runtime library
- list and describe the include files
- describe the arguments to function **main**, and its return value
- summarize the different categories of tasks performed by the library routines
- describe (in lookup fashion) common global variables implemented in many of the library routines

The Library Routine Lookup Section

The second part of this reference guide is an alphabetical lookup; it contains a description of each of the Turbo C routines.

A few of the routines are grouped by "family" (the **exec...** and **spawn...** functions that create, load, and run programs, for example) because they perform similar or related tasks.

Otherwise, we have included an individual entry in the lookup for every routine. For instance, if you want to look up information about the **free** routine, you would look under **free**; there you would find a listing for **free** that

- summarizes what **free** does
- gives the syntax for calling **free**
- tells you which header file(s) contains the prototype for **free**
- gives a detailed description of how **free** is implemented and how it relates to the other memory-allocation routines
- lists other language compilers that include similar functions
- refers you to related Turbo C functions
- if appropriate, gives an example of how the function is used, or refers you to a function entry where there is such an example

The last part of this reference guide contains several appendices designed to give you detailed reference and usage information about some of Turbo C's special features:

- the Turbo C Interactive Editor
- Turbo C compiler error messages

- the TCC command-line options
- the Turbo C standalone utilities
- the Turbo C language syntax summary
- TCINST, the Turbo C customization program
- MicroCalc, a sample spreadsheet application

Why You Might Want to Access the Turbo C Run-Time Library Source Code

The Turbo C run-time library contains over 300 functions, covering a broad range of areas: low-level control of your IBM PC, interfacing with DOS, input/output, process management, string and memory manipulations, math, sorting and searching, and so on. There are several good reasons why you may wish to obtain the source code for these functions:

- You may find that a particular Turbo C function you want to write is similar to, but not the same as, a function in the library. With access to the run-time library source code, you can tailor the library function to your own needs, and avoid having to write a separate function of your own.
- Sometimes, when you are debugging code, you may wish to know more about the internals of a library function. Having the source code to the run-time library would be of great help in this situation.
- When you can't figure out what a library function is really supposed to do, it's useful to be able to take a quick look at that function's source code.
- You may dislike the underscore convention on C symbols, and wish you had a version of the libraries without leading underscores. Again, access to the source code to the run-time library will let you eliminate leading underscores.
- You can also learn a lot from studying tight, professionally written library source code.

For all these reasons, and more, you will want to have access to the Turbo C run-time library source code. Because Borland believes strongly in the concepts of "open architecture," we have made the Turbo C run-time library source code available for licensing. All you have to do is fill out the order form distributed with this documentation, include your payment, and we'll ship you the Turbo C run-time library source code.

The Turbo C Include Files

Header files provide function prototype declarations for library functions. Data types and symbolic constants used with the library functions are also defined in them, along with global
variables defined by Turbo C and by the library functions. The Turbo C library follows the ANSI C draft standard on names of header files and their contents. Header files defined by ANSI C are marked with an asterisk (*) in the list below.

alloc.h — Declares memory management functions (allocation, deallocation, etc.).

assert.h* — Defines the **assert** debugging macro.

bios.h — Declares various functions used in calling IBM-PC ROM BIOS routines.

conio.h — Declares various functions used in calling the DOS console I/O routines.

ctype.h* — Contains information used by the character classification and character conversion macros (such as **isalpha** and **toascii**).

dir.h — Contains structures, macros, and functions for working with directories and path names.

dos.h — Defines various constants and gives declarations needed for DOS and 8086-specific calls.

errno.h* — Defines constant mnemonics for the error codes.

fcntl.h — Defines symbolic constants used in connection with the library routine **open**.

float.h* — Contains parameters for floating-point routines.

graphics.h — Declares prototypes for the graphics functions.

io.h — Contains structures and declarations for low-level input/output routines.

limits.h* — Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.

math.h* — Declares prototypes for the math functions; also defines the macro HUGE_VAL, and declares the exception structure used by the **matherr** and **_matherr** routines.

mem.h — Declares the memory-manipulation functions. (Many of these are also defined in string.h.)

process.h — Contains structures and declarations for **spawn...** and **exec...** functions.

setjmp.h* — Defines a type *jmp_buf* used by the **longjmp** and **setjmp** functions and declares the routines **longjmp** and **setjmp**.

share.h — Defines parameters used in functions that make use of file-sharing.

signal.h* — Defines constants and declarations for use by the **signal** and **raise** functions.

stdargs.h* — Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as **vprintf**, **vscanf**, etc.).

stddef.h* — Defines several common data types and macros.

stdio.h* — Defines types and macros needed for the Standard I/O Package defined in Kernighan and Ritchie and extended under UNIX System V. Defines the standard I/O predefined streams *stdin*, *stdout*, *stdprn*, and *stderr*, and declares stream-level I/O routines.

stdlib.h* — Declares several commonly used routines: conversion routines, search/sort routines, and other miscellany.

string.h* — Declares several string-manipulation and memory-manipulation routines.

sys\stat.h — Defines symbolic constants used for opening and creating files.

sys\timeb.h — Declares the function **ftime** and the structure **timeb** that **ftime** returns.

sys\types.h — Declares the type *time_t* used with time functions.

time.h* — Defines a structure filled in by the time-conversion routines **asctime**, **localtime**, and **gmtime**, and a type used by the routines **ctime**, **difftime**, **gmtime**, **localtime**, and **stime**; also provides prototypes for these routines.

values.h — Defines important constants, including machine dependencies; provided for UNIX System V compatibility.

Library Routines by Category

The Turbo C library routines perform a variety of tasks. In this section, we list the routines, along with the include files in which they are declared, under several general categories of task performed. For complete information about any of the functions below, see the function entry in Chapter 2 of this manual.

Classification Routines

These routines classify ASCII characters as letters, control characters, punctuation, uppercase, etc.

isalnum	(ctype.h)	**isdigit**	(ctype.h)	**ispunct**	(ctype.h)
isalpha	(ctype.h)	**isgraph**	(ctype.h)	**isspace**	(ctype.h)
isascii	(ctype.h)	**islower**	(ctype.h)	**isupper**	(ctype.h)
iscntrl	(ctype.h)	**isprint**	(ctype.h)	**isxdigit**	(ctype.h)

Conversion Routines

These routines convert characters and strings from alpha to different numeric representations (floating-point, integers, longs) and vice versa, and from uppercase to lowercase and vice versa.

atof	(stdlib.h)	**itoa**	(stdlib.h)	**toascii**	(ctype.h)
atoi	(stdlib.h)	**ltoa**	(stdlib.h)	**tolower**	(ctype.h)
atol	(stdlib.h)	**strtod**	(stdlib.h)	**_toupper**	(ctype.h)
ecvt	(stdlib.h)	**strtol**	(stdlib.h)	**toupper**	(ctype.h)
fcvt	(stdlib.h)	**strtoul**	(stdlib.h)	**ultoa**	(stdlib.h)
gcvt	(stdlib.h)	**_tolower**	(ctype.h)		

Directory Control Routines

These routines manipulate directories and path names.

chdir	(dir.h)	**getcurdir**	(dir.h)	**mktemp**	(dir.h)
findfirst	(dir.h)	**getcwd**	(dir.h)	**rmdir**	(dir.h)
findnext	(dir.h)	**getdisk**	(dir.h)	**searchpath**	(dir.h)
fnmerge	(dir.h)	**mkdir**	(dir.h)	**setdisk**	(dir.h)
fnsplit	(dir.h)				

Diagnostic Routines

These routines provide built-in troubleshooting capability.

assert	(assert.h)	**matherr**	(math.h)	**perror**	(errno.h)

Graphics Routines

These routines let you create onscreen graphics with text.

arc (graphics.h)
bar (graphics.h)
bar3d (graphics.h)
circle (graphics.h)
cleardevice (graphics.h)
clearviewport (graphics.h)
closegraph (graphics.h)
detectgraph (graphics.h)
drawpoly (graphics.h)
ellipse (graphics.h)
fillellipse (graphics.h)
fillpoly (graphics.h)
floodfill (graphics.h)
getarccoords (graphics.h)
getaspectratio (graphics.h)
getbkcolor (graphics.h)
getcolor (graphics.h)
getdefaultpalette (graphics.h)
getdrivername (graphics.h)
getfillpattern (graphics.h)
getfillsettings (graphics.h)
getgraphmode (graphics.h)
getimage (graphics.h)
getlinesettings (graphics.h)
getmaxcolor (graphics.h)
getmaxmode (graphics.h)
getmaxx (graphics.h)
getmaxy (graphics.h)
getmodename (graphics.h)
getmoderange (graphics.h)
getpalette (graphics.h)
getpalettesize (graphics.h)
getpixel (graphics.h)
gettextsettings (graphics.h)
getviewsettings (graphics.h)
getx (graphics.h)
gety (graphics.h)
graphdefaults (graphics.h)
grapherrormsg (graphics.h)
_graphfreemem (graphics.h)
_graphgetmem (graphics.h)
graphresult (graphics.h)
imagesize (graphics.h)
initgraph (graphics.h)
installuserdriver (graphics.h)
installuserfont
line (graphics.h)
linerel (graphics.h)
lineto (graphics.h)
moverel (graphics.h)
moveto (graphics.h)
outtext (graphics.h)
outtextxy (graphics.h)
pieslice (graphics.h)
putimage (graphics.h)
putpixel (graphics.h)
rectangle (graphics.h)
registerbgidriver (graphics.h)
registerbgifont (graphics.h)
restorecrtmode (graphics.h)
sector (graphics.h)
setactivepage (graphics.h)
setallpalette (graphics.h)
setaspectratio (graphics.h)
setbkcolor (graphics.h)
setcolor (graphics.h)
setfillpattern (graphics.h)
setfillstyle (graphics.h)
setgraphbufsize (graphics.h)
setgraphmode (graphics.h)
setlinestyle (graphics.h)
setpalette (graphics.h)
setrgbpalette (graphics.h)
settextjustify (graphics.h)
settextstyle (graphics.h)
setusercharsize (graphics.h)
setviewport (graphics.h)
setvisualpage (graphics.h)
setwritemode (graphics.h)
textheight (graphics.h)
textwidth (graphics.h)

Input/Output Routines

These routines provide stream-level and DOS-level I/O capability.

access (io.h)
cgets (conio.h)
_chmod (io.h)
chmod (io.h)
chsize (io.h)
clearerr (stdio.h)
close (io.h)
_close (io.h)
cprintf (conio.h)
cputs (conio.h)
creat (io.h)
_creat (io.h)
creatnew (io.h)
creattemp (io.h)
cscanf (conio.h)
dup (io.h)
dup2 (io.h)
eof (io.h)
fclose (stdio.h)
fcloseall (stdio.h)
fdopen (stdio.h)
feof (stdio.h)
ferror (stdio.h)
fflush (stdio.h)
fgetc (stdio.h)
fgetchar (stdio.h)
fgetpos (stdio.h)
fgets (stdio.h)
filelength (io.h)
fileno (stdio.h)
flushall (stdio.h)
fopen (stdio.h)
fprintf (stdio.h)
fputc (stdio.h)
fputchar (stdio.h)
fputs (stdio.h)
fread (stdio.h)
freopen (stdio.h)
fscanf (stdio.h)
fseek (stdio.h)
fsetpos (stdio.h)
fstat (sys\stat.h)
ftell (stdio.h)
fwrite (stdio.h)
getc (stdio.h)
getch (conio.h)
getchar (stdio.h)
getche (conio.h)
getftime (io.h)
getpass (conio.h)
gets (stdio.h)
getw (stdio.h)
gsignal (signal.h)
ioctl (io.h)
isatty (io.h)
kbhit (conio.h)
lock (io.h)
lseek (io.h)
_open (io.h)
open (io.h)
perror (stdio.h)
printf (stdio.h)
putc (stdio.h)
putch (conio.h)
putchar (stdio.h)
puts (stdio.h)
putw (stdio.h)
_read (io.h)
read (io.h)
remove (stdio.h)
rename (stdio.h)
rewind (stdio.h)
scanf (stdio.h)
setbuf (stdio.h)
setftime (io.h)
setmode (io.h)
setvbuf (stdio.h)
sopen (io.h)
sprintf (stdio.h)
sscanf (stdio.h)
stat (sys\stat.h)
_strerror (string.h, stdio.h)
strerror (stdio.h)
tell (io.h)
tmpfile (stdio.h)
tmpnam (stdio.h)
ungetc (stdio.h)
ungetch (conio.h)
unlock (io.h)
vfprintf (stdio.h)
vfscanf (stdio.h)
vprintf (stdio.h)
vscanf (stdio.h)
vsprintf (stdio.h)
vsscanf (io.h)
_write (io.h)
write (io.h)

Interface Routines (DOS, 8086, BIOS)

These routines provide DOS, BIOS and machine-specific capabilities.

absread	(dos.h)	**geninterrupt**	(dos.h)	**keep**	(dos.h)
abswrite	(dos.h)	**getcbrk**	(dos.h)	**MK_FP**	(dos.h)
bdos	(dos.h)	**getdfree**	(dos.h)	**outport**	(dos.h)
bdosptr	(dos.h)	**getdta**	(dos.h)	**outportb**	(dos.h)
bioscom	(bios.h)	**getfat**	(dos.h)	**parsfnm**	(dos.h)
biosdisk	(bios.h)	**getfatd**	(dos.h)	**peek**	(dos.h)
biosequip	(bios.h)	**getpsp**	(dos.h)	**peekb**	(dos.h)
bioskey	(bios.h)	**getvect**	(dos.h)	**poke**	(dos.h)
biosmemory	(bios.h)	**getverify**	(dos.h)	**pokeb**	(dos.h)
biosprint	(bios.h)	**harderr**	(dos.h)	**randbrd**	(dos.h)
biostime	(bios.h)	**hardresume**	(dos.h)	**randbwr**	(dos.h)
country	(dos.h)	**hardretn**	(dos.h)	**segread**	(dos.h)
ctrlbrk	(dos.h)	**inport**	(dos.h)	**setcbrk**	(dos.h)
disable	(dos.h)	**inportb**	(dos.h)	**setdta**	(dos.h)
dosexterr	(dos.h)	**int86**	(dos.h)	**setvect**	(dos.h)
enable	(dos.h)	**int86x**	(dos.h)	**setverify**	(dos.h)
FP_OFF	(dos.h)	**intdos**	(dos.h)	**sleep**	(dos.h)
FP_SEG	(dos.h)	**intdosx**	(dos.h)	**unlink**	(dos.h)
freemem	(dos.h)	**intr**	(dos.h)		

Manipulation Routines

These routines handle strings and blocks of memory: copying, comparing, converting, and searching.

memccpy	(mem.h, string.h)	**strchr**	(string.h)	**strncmpi**	(string.h)
memchr	(mem.h, string.h)	**strcmp**	(string.h)	**strncpy**	(string.h)
memcmp	(mem.h, string.h)	**stricmpi**	(string.h)	**strnicmp**	(string.h)
memcpy	(mem.h, string.h)	**strcpy**	(string.h)	**strnset**	(string.h)
memicmp	(mem.h, string.h)	**strcspn**	(string.h)	**strpbrk**	(string.h)
memmove	(mem.h, string.h)	**strdup**	(string.h)	**strrchr**	(string.h)
memset	(mem.h, string.h)	**strerror**	(string.h)	**strrev**	(string.h)
movedata	(mem.h, string.h)	**stricmp**	(string.h)	**strset**	(string.h)
movmem	(mem.h, string.h)	**strlen**	(string.h)	**strspn**	(string.h)
setmem	(mem.h)	**strlwr**	(string.h)	**strstr**	(string.h)
stpcpy	(string.h)	**strncat**	(string.h)	**strtok**	(string.h)
strcat	(string.h)	**strncmp**	(string.h)	**strupr**	(string.h)

Math Routines

These routines perform mathematical calculations and conversions.

abs	(stdlib.h)	**fcvt**	(stdlib.h)	**poly**	(math.h)
acos	(math.h)	**floor**	(math.h)	**pow**	(math.h)
asin	(math.h)	**fmod**	(math.h)	**pow10**	(math.h)
atan	(math.h)	**_fpreset**	(float.h)	**rand**	(stdlib.h)
atan2	(math.h)	**frexp**	(math.h)	**random**	(stdlib.h)
atof	(stdlib.h, math.h)	**gcvt**	(stdlib.h)	**randomize**	(stdlib.h)
		hypot	(math.h)	**_rotl**	(stdlib.h)
atoi	(stdlib.h)	**itoa**	(stdlib.h)	**_rotr**	(stdlib.h)
atol	(stdlib.h)	**labs**	(stdlib.h)	**sin**	(math.h)
cabs	(math.h)	**ldexp**	(math.h)	**sinh**	(math.h)
ceil	(math.h)	**ldiv**	(math)	**sqrt**	(math.h)
_clear87	(float.h)	**log**	(math.h)	**srand**	(stdlib.h)
_control87	(float.h)	**log10**	(math.h)	**_status87**	(float.h)
cos	(math.h)	**_lrotl**	(stdlib.h)	**strtod**	(stdlib.h)
cosh	(math.h)	**_lrotr**	(stdlib.h)	**strtol**	(stdlib.h)
div	(math.h)	**ltoa**	(stdlib.h)	**strtoul**	(stdlib.h)
ecvt	(stdlib.h)	**_matherr**	(math.h)	**tan**	(math.h)
exp	(math.h)	**matherr**	(math.h)	**tanh**	(math.h)
fabs	(math.h)	**modf**	(math.h)	**ultoa**	(stdlib.h)

Memory Allocation Routines

These routines provide dynamic memory allocation in the small-data and large-data models.

allocmem	(dos.h)	**farmalloc**	(alloc.h)
brk	(alloc.h)	**farrealloc**	(alloc.h)
calloc	(alloc.h)	**free**	(alloc.h, stdlib.h)
coreleft	(alloc.h, stdlib.h)	**malloc**	(alloc.h, stdlib.h)
farcalloc	(alloc.h)	**realloc**	(alloc.h, stdlib.h)
farcoreleft	(alloc.h)	**sbrk**	(alloc.h)
farfree	(alloc.h)	**setblock**	(dos.h)

Miscellaneous Routines

These routines provide nonlocal goto capabilities and sound effects.

delay	(dos.h)	**setjmp**	(setjmp.h)
longjmp	(setjmp.h)	**sound**	(dos.h)
nosound	(dos.h)		

Process Control Routines

These routines invoke and terminate new processes from within another.

abort	(process.h)	**raise**	(signal.h)
execl	(process.h)	**signal**	(signal.h)
execle	(process.h)	**spawnl**	(process.h)
execlp	(process.h)	**spawnle**	(process.h)
execlpe	(process.h)	**spawnlp**	(process.h)
execv	(process.h)	**spawnlpe**	(process.h)
execve	(process.h)	**spawnv**	(process.h)
execvp	(process.h)	**spawnve**	(process.h)
execvpe	(process.h)	**spawnvp**	(process.h)
_exit	(process.h)	**spawnvpe**	(process.h)
exit	(process.h)	**system**	(process.h)

Standard Routines

These are standard routines.

abort	(stdlib.h)	**fcvt**	(stdlib.h)	**putenv**	(stdlib.h)
abs	(stdlib.h)	**free**	(stdlib.h)	**qsort**	(stdlib.h)
atexit	(stdlib.h)	**gcvt**	(stdlib.h)	**rand**	(stdlib.h)
atof	(stdlib.h)	**getenv**	(stdlib.h)	**realloc**	(stdlib.h)
atoi	(stdlib.h)	**itoa**	(stdlib.h)	**srand**	(stdlib.h)
atol	(stdlib.h)	**labs**	(stdlib.h)	**strtod**	(stdlib.h)
bsearch	(stdlib.h)	**lfind**	(stdlib.h)	**strtol**	(stdlib.h)
calloc	(stdlib.h)	**lsearch**	(stdlib.h)	**swab**	(stdlib.h)
ecvt	(stdlib.h)	**ltoa**	(stdlib.h)	**system**	(stdlib.h)
_exit	(stdlib.h)	**malloc**	(stdlib.h)	**ultoa**	(stdlib.h)
exit	(stdlib.h)				

Text Window Display Routines

These routines output text to the screen.

clreol	(conio.h)	**insline**	(conio.h)	**textbackground**	(conio.h)
clrscr	(conio.h)	**lowvideo**	(conio.h)	**textcolor**	(conio.h)
delline	(conio.h)	**movetext**	(conio.h)	**textmode**	(conio.h)
gettext	(conio.h)	**normvideo**	(conio.h)	**wherex**	(conio.h)
gettextinfo	(conio.h)	**puttext**	(conio.h)	**wherey**	(conio.h)
gotoxy	(conio.h)	**textattr**	(conio.h)	**window**	(conio.h)
highvideo	(conio.h)				

Time and Date Routines

These are time conversion and time manipulation routines.

asctime	(time.h)	**getdate**	(dos.h)	**settime**	(dos.h)
ctime	(time.h)	**gettime**	(dos.h)	**stime**	(time.h)
difftime	(time.h)	**gmtime**	(time.h)	**time**	(time.h)
dostounix	(dos.h)	**localtime**	(time.h)	**tzset**	(time.h)
ftime	(sys\timeb.h)	**setdate**	(dos.h)	**unixtodos**	(dos.h)

Variable Argument List Routines

These routines are for use when accessing variable argument lists (such as with **vprintf**, etc).

va_arg	(stdarg.h)	**va_end**	(stdarg.h)	**va_start**	(stdarg.h)

The main Function

Every C program must have a **main** function; where you place it is a matter of preference. Some programmers place **main** at the beginning of the file, others at the very end. But regardless of its location, the following points about **main** always apply.

The Arguments to main

Three parameters (arguments) are passed to main by the Turbo C startup routine: *argc*, *argv*, and *env*.

- *argc*, an integer, is the number of command-line arguments passed to **main**.
- *argv* is an array of pointers to strings.
  - Under 3.x versions of DOS, *argv*[0] is defined as the full path name of the program being run.
  - Under versions of DOS before 3.0, *argv*[0] points to the null string ("").
  - *argv*[1] points to the first string typed on the DOS command line after the program name.
  - *argv*[2] points to the second string typed after the program name.
  - *argv*[*argc* – 1] points to the last argument passed to main.
  - *argv*[*argc*] contains NULL.
- *env* is also an array of pointers to strings. Each element of *env*[] holds a string of the form `ENVVAR=value`.
  - `ENVVAR` is the name of an environment variable, such as `PATH` or `87`.
  - `value` is the value to which an `ENVVAR` is set, such as `C:\DOS;C:\TURBOC` (for `PATH`) or `YES` (for `87`).

The Turbo C startup routine always passes these three arguments to **main**; you have the option of whether to declare them in your program. If you declare some (or all) of these arguments to **main**, they are made available as local variables to your **main** routine.

Note, however, that if you do declare any of these parameters, you *must* declare them exactly in the order given: *argc*, *argv*, *env*.

For example, the following are all valid declarations of **main**'s arguments:

```
main()
main(int argc)                                    /* legal but very unlikely */
main(int argc, char * argv[])
main(int argc, char * argv[], char * env[])
```

Note: The declaration `main(int argc)` is legal, but it's very unlikely that you would use *argc* in your program without also using the elements of *argv*.

Another Note: The argument *env* is also available via the global variable *environ*. Refer to the *environ* lookup entry (in this chapter) and the **putenv** and **getenv** lookup entries (in Chapter 2 of this manual) for more information. *argc* and *argv* are also available via the global variables *_argc* and *_argv*.

An Example Program Using argc, argv and env

Here is an example program, ARGS.EXE, that demonstrates a simple way of using these arguments passed to **main**.

```
                        /* Program ARGS.C */

#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[], char *env[])
{
   int i;

   printf("The value of argc is %d \n\n",argc);
   printf("These are the %d command-line arguments passed to main:\n\n",argc);

   for (i = 0; i <= argc; i++)
      printf("   argv[%d]: %s\n", i, argv[i]);

   printf("\nThe environment string(s) on this system are:\n\n");

   for (i = 0; env[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, env[i]);
}
```

Suppose you run ARGS.EXE at the DOS prompt with the following command line:

```
c:> args first_argument "argument with blanks" 3  4 "last but one" stop!
```

Note that you can pass arguments with embedded blanks by surrounding them with double quotes, as shown by `"argument with blanks"` and `"last but one"` in this example command line.

The output of ARGS.EXE (assuming that the environment variables are set as shown here) would then be like this:

```
The value of argc is 7

These are the 7 command-line arguments passed to main:
argv[0]: C:\TURBOC\TESTARGS.EXE
argv[1]: first_argument
argv[2]: argument with blanks
argv[3]: 3
argv[4]: 4
argv[5]: last but one
argv[6]: stop!
argv[7]: (null)

The environment string(s) on this system are:
env[0]: COMSPEC=C:\COMMAND.COM
env[1]: PROMPT=$p $g
env[2]: PATH=C:\SPRINT;C:\DOS;C:\TURBOC
```

Note: The maximum combined length of the command-line arguments passed to **main** (including the space between adjacent arguments and the name of the program itself) is 128 characters; this is a DOS limit.

Wildcard Command-Line Arguments to main

Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARGS.OBJ object file, which is included with Turbo C.

Once WILDARGS.OBJ is linked into your program code, you can send wildcard arguments of the type `*.*` to your **main** function. The argument will be expanded (in the *argv* array) to all files matching the wildcard mask. The maximum size of the *argv* array will vary, depending on the amount of memory available in your heap.

If no matching files are found, the argument is passed unchanged. (That is, a string consisting of the wildcard mask is passed to **main**.)

Arguments enclosed in quotes ("...") are not expanded.

An Example: The following commands will compile the file ARGS.C and link it with the wildcard expansion module WILDARGS.OBJ, then run the resulting executable file ARGS.EXE:

```
tcc  args  wildargs.obj
```

```
args  C:\TC\INCLUDE\*.H  "*.C"
```

When ARGS.EXE is run, the first argument is expanded to the names of all the *.H files in the C:\TC\INCLUDE directory. Note that the expanded argument strings include the entire path (for example, C:\TC\INCLUDE\ALLOC.H). The argument *.C will not be expanded, as it is enclosed in quotes.

In the Integrated Environment (TC.EXE), you simply specify a project file on the project menu, which contains the following lines:

```
ARGS
WILDARGS.OBJ
```

Then use the **Options/Args** option to set the command-line parameters.

Note: If you prefer the wildcard expansion to be the default so that you won't have to link your program explicitly with WILDARGS.OBJ, you can modify your standard C?.LIB library files to have WILDARGS.OBJ linked automatically. In order to accomplish that, you have to remove SETARGV from the libraries, and add WILDARGS. The following commands will invoke the Turbo librarian to modify all the standard library files (assuming the current directory contains the standard C libraries, and WILDARGS.OBJ):

```
tlib  cs  -setargv  +wildargs
tlib  cc  -setargv  +wildargs
tlib  cm  -setargv  +wildargs
tlib  cl  -setargv  +wildargs
tlib  ch  -setargv  +wildargs
```

When You Compile Using –p (Pascal Calling Conventions)

If you compile your program using Pascal calling conventions (described in detail in Chapter 12 of the *Turbo C User's Guide*), you *must* remember to explicitly declare **main** as a C type.
You do this with the **cdecl** keyword, like this:

```
cdecl  main(int argc, char * argv[], char * envp[])
```

The Value main Returns

The value returned by **main** is the status code of the program: an **int**. If, however, your program uses the routine **exit** (or **_exit**) to terminate, the value returned by **main** is the argument passed to the call to **exit** (or to **_exit**).

For example, if your program contains the call

```
exit(1)
```

the status is 1.

If you are using the Integrated Environment version of Turbo C (TC.EXE) to run your program, you can display the return value from **main** by selecting the Get Info item on the Compile menu (*Alt-C, G*).

Global Variables

_argc

Function Keeps a count of command-line arguments.

Syntax extern int *_argc;*

Declared in dos.h

Remarks *_argc* has the value of *argc* passed to **main** when the program starts.

_argv

Function An array of pointers to command-line arguments.

Syntax extern char **_argv*[];

Declared in dos.h

Remarks *_argv* points to an array containing the original command-line arguments (the elements of *argv[]*) passed to **main** when the program starts.

daylight

Function Indicates whether Daylight Savings Time is in effect.

Syntax extern int *daylight;*

Declared in time.h

Remarks *daylight* is used by the time-and-date functions. It is set by the **tzset, ftime,** and **localtime** functions to 1 for Daylight Savings Time, 0 for Standard Time.

directvideo

Function Flag that controls video output.

Syntax extern int *directvideo;*

Declared in conio.h

Remarks *directvideo* controls whether your program's console output (from **cputs**, for example) goes directly to the video RAM (*directvideo* = 1) or goes via ROM BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to video RAM). In order to use *directvideo* = 1, your system's video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 allows your console output to work on any system that is IBM BIOS-compatible.

_8087

Function Coprocessor chip flag.

Syntax extern int *_8087;*

Declared in dos.h

Remarks The *_8087* variable is set to a nonzero value (1, 2, or 3) if the startup code autodetection logic detects a floating-point coprocessor (an 8087, 80287, or 80387, respectively). The *_8087* variable is set to 0 otherwise.

The autodetection logic can be overridden by setting the 87 environment variable to `YES` or `NO`. (The commands are `SET 87=YES` and `SET 87=NO`; it is essential that there be no spaces before or after the equal sign.) In this case, the *_8087* variable will reflect the override, and be set to 1 or 0.

Refer to Chapter 12 in the *Turbo C User's Guide* for more information about the 87 environment variable.

You must have floating-point code in your program for the *_8087* variable to be defined properly.

environ

Function Accesses DOS environment variables.

Syntax extern char * *environ*[ ];

Declared in dos.h

Remarks *environ* is an array of pointers to strings; it is used to access and alter the DOS environment variables. Each string is of the form

envvar = *varvalue*

where *envvar* is the name of an environment variable (such as `PATH`), and *varvalue* is the string value to which *envvar* is set (such as `C:\BIN;C:\DOS`). The string *varvalue* may be empty.

When a program begins execution, the DOS environment settings are passed directly to the program. Note that *env*, the third argument to **main**, is equal to the initial setting of *environ*.

The *environ* array can be accessed by **getenv**; however, the **putenv** function is the only routine that should be used to add, change or delete the *environ* array entries. This is because modification can resize and relocate the process environment array, but *environ* is automatically adjusted so that it always points to the array.

errno, _doserrno, sys_errlist, sys_nerr

Function Enable **perror** to print error messages.

Syntax extern int *errno*;
extern int *_doserrno*;
extern char * *sys_errlist*[ ];
extern int *sys_nerr*;

Declared in errno.h, stdlib.h (*errno, _doserrno, sys_errlist, sys_nerr*)
dos.h (*_doserrno*)

Remarks *errno, sys_errlist,* and *sys_nerr* are used by **perror** to print error messages when certain library routines fail to

accomplish their appointed tasks. *_doserrno* is a variable that maps many DOS error codes to *errno*; however, **perror** does not use *_doserrno* directly.

_doserrno: When a DOS system call results in an error, *_doserrno* is set to the actual DOS error code. *errno* is a parallel error variable inherited from UNIX.

errno: When an error in a system call occurs, *errno* is set to indicate the type of error. Sometimes *errno* and *_doserrno* are equivalent. At other times, *errno* does not contain the actual DOS error code, which is contained in *_doserrno*. Still other errors might occur that set only *errno*, not *_doserrno*.

sys_errlist: To provide more control over message formatting, the array of message strings is provided in *sys_errlist*. *errno* can be used as an index into the array to find the string corresponding to the error number. The string does not include any newline character.

sys_nerr: This variable is defined as the number of error message strings in *sys_errlist*.

The following table gives mnemonics and their meanings for the values stored in *sys_errlist*.

Mnemonic	Meaning
E2BIG	Arg list too long
EACCES	Permission denied
EBADF	Bad file number
ECONTR	Memory blocks destroyed
ECURDIR	Attempt to remove CurDir
EDOM	Domain error
EEXIST	File already exists
EINVACC	Invalid access code
EINVAL	Invalid argument
EINVDAT	Invalid data
EINVDRV	Invalid drive specified
EINVENV	Invalid environment
EINVFMT	Invalid format
EINVFNC	Invalid function number
EINVMEM	Invalid memory block address
EMFILE	Too many open files
ENMFILE	No more files
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOFILE	No such file or directory
ENOMEM	Not enough memory
ENOPATH	Path not found
ENOTSAM	Not same device
ERANGE	Result out of range
EXDEV	Cross-device link
EZERO	Error 0

The following list gives mnemonics for the actual DOS error codes to which *_doserrno* can be set. (This value of *_doserrno* may or may not be mapped (through *errno*) to an equivalent error message string in *sys_errlist*.

Mnemonic	DOS error code
EINVAL	Bad function
E2BIG	Bad environ
EACCES	Access denied
EACCES	Bad access
EACCES	Is current dir
EBADF	Bad handle
EFAULT	Reserved
EINVAL	Bad data
EMFILE	Too many open
ENOENT	No such file or directory
ENOEXEC	Bad format
ENOMEM	Mcb destroyed
ENOMEM	Out of memory
ENOMEM	Bad block
EXDEV	Bad drive
EXDEV	Not same device

Refer to the Microsoft *MS-DOS Programmer's Reference Manual* for more information about DOS error return codes.

_fmode

Function Determines default file-translation mode.

Syntax extern int *_fmode;*

Declared in fcntl.h

Remarks *_fmode* determines in which mode (text or binary) files will be opened and translated. The value of *_fmode* is O_TEXT by default, which specifies that files will be read in text mode. If *_fmode* is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.)

In text mode, on input carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character

(LF). On output, the reverse is true: LF characters are translated to CR/LF combinations.

In binary mode, no such translation occurs.

You can override the default mode as set by *_fmode* by specifying a *t* (for text mode) or *b* (for binary mode) in the argument *type* in the library routines **fopen**, **fdopen**, and **freopen**. Also, in the routine **open**, the argument *access* can include either O_BINARY or O_TEXT, which will explicitly define the file being opened (given by the **open** *pathname* argument) to be in either binary or text mode.

_heaplen

Function Holds the length of the near heap.

Syntax extern unsigned *_heaplen;*

Declared in dos.h

Remarks *_heaplen* specifies the size 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), as 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.

```
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, so the data segment is simply

```
data segment [compact,large] = global data + stack
```

In the huge model, the stack is a separate segment, and each module has its own data segment.

_osmajor, _osminor

Function Contain the major and minor DOS version numbers.

Syntax extern unsigned char *_osmajor;*
extern unsigned char *_osminor;*

Declared in dos.h

Remarks The major and minor version numbers are available individually through *_osmajor* and *_osminor*. *_osmajor* is the major version number, and *_osminor* is the minor version number. For example, if you are running DOS version 3.2, *_osmajor* will be 3, and *_osminor* will be 20.

These variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x. (For example, refer to **_open**, **creatnew**, and **ioctl** in the lookup section of this *Reference Guide.*)

_psp

Syntax extern unsigned int *_psp;*

Declared in dos.h

Remarks *_psp* contains the segment address of the program segment prefix (PSP) for the current program. The PSP is

a DOS process descriptor; it contains initial DOS information about the program.

Refer to the Microsoft *MS-DOS Programmer's Reference Manual* for more information on the PSP.

_stklen

Function Holds size of the stack.

Syntax extern unsigned *_stklen;*

Declared in dos.h

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.

```
data segment[tiny] = 256 + code + global data
                   + heap + stack
```

In the compact and large models, there is no near heap, so the data segment is simply

```
data segment [compact,large] = global data + stack
```

In the huge model, the stack is a separate segment, and each module has its own data segment.

See also *_heaplen*

timezone

Function Contains difference in seconds between local time and GMT.

Syntax extern long *timezone;*

Declared in time.h

Remarks *timezone* is used by the time-and-date functions.

This variable is calculated by the **tzset** function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich Mean Time.

tzname

Function Array of pointers to time zone names.

Syntax extern char * tzname[2]

Declared in time.h

Remarks The global variable *tzname* is an array of pointers to strings containing abbreviations for time zone names. *tzname*[0] points to a three-character string with the value of the time zone name from the *TZ* environment string. The global variable *tzname*[1] points to a three-character string with the value of the daylight savings time zone name from the *TZ* environment string. If no daylight savings name is present, *tzname*[1] points to a null string.

_version

Function Contains the DOS version number.

Syntax extern unsigned int *_version;*

Declared in dos.h

Remarks

_version contains the DOS version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version *x.y*, the *x* is the major version number, and *y* is the minor.)

C H A P T E R

2

The Turbo C Library

This chapter contains a detailed description of each of the functions in the Turbo C library.

The following sample library look-up entry explains how to use this portion of the *Turbo C Reference Guide* to reference the Turbo C library functions.

function name

Function Summary of what **function** does.

Syntax #include <header.h>

(The header file(s) containing the prototype for **function** or definitions of constants, enumerated types, etc., used by the function; it is listed only if it must be #included in the routine calling **function**.)

```
function(modifier parameter[,...]);
```

(The declaration syntax for **function**; parameter names are *italicized*. The `[,...]` indicates that other parameters and their modifiers may follow.)

Prototype in header.h

(Header file(s) containing the prototype for **function**. The prototype of some functions is contained in more than one header file; in cases such as this, each of the files is listed.)

Remarks This describes what **function** does, the parameters it takes, and any details you need to use **function** and the related routines listed.

Return value The value that **function** returns (if any) is given here. If **function** sets the global variable *errno*, that value is also listed.

Portability The system(s) and language(s) that **function** is available for are listed here. These may include UNIX, IBM PC's and compatibles, and the ANSI C standard.

See also Routines related to **function** that you might wish to read about are listed here. **Note:** If a routine name contains an *ellipsis* (**funcname...**, **...funcname**, **func...name**), it indicates that you should refer to a family of functions (for example, **exec...**).

Example Some entries include a sample program demonstrating how **function** is used.

abort

Function Abnormally terminates a process.

Syntax void abort(void);

Prototype in stdlib.h, process.h

Remarks **abort** writes a termination message (`Abnormal program termination`) on *stderr* and aborts the program via a call to **_exit** with exit code 3.

Return value **abort** returns the exit code 3 to the parent process or to DOS.

Portability **abort** is available on UNIX systems and is compatible with ANSI C.

See also **assert, atexit, exit, _exit, raise, signal, spawn...**

abs

Function Returns the absolute value of an integer.

Syntax #include <math.h>
int abs(int *x*);

Prototype in math.h, stdlib.h

Remarks **abs** returns the absolute value of the integer argument *x*. If **abs** is called when stdlib.h has been included, it will be treated as a macro that expands to inline code.

If you want to use the **abs** function instead of the macro, include

```
#undef abs
```

in your program, after the `#include <stdlib.h>`.

Return value **abs** returns an integer in the range of 0 to 32,767, with the exception that an argument of –32,768 is returned as –32,768.

Portability **abs** is available on UNIX systems and is compatible with ANSI C.

See also **cabs, fabs, labs**

---

absread

Function Reads absolute disk sectors.

Syntax

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

Prototype in dos.h

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

absread reads specific disk sectors via DOS interrupt 0x25.

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 the amount of memory in the segment above *buffer*. Thus, 64K is the largest amount of memory that can be read in a single call to **absread**.

Return value If it is successful, **absread** returns 0.

On error, the routine returns –1 and sets *errno* to the value of the AX register returned by the system call. See the DOS documentation for the interpretation of *errno*.

Portability **absread** is unique to DOS.

See also **abswrite, biosdisk**

abswrite

Function Writes absolute disk sectors.

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

Prototype in dos.h

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 it is used improperly, **abswrite** can overwrite files, directories, and FATs.

abswrite writes specific disk sectors via DOS interrupt 0x26.

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 the amount of memory in the segment above *buffer*. Thus, 64K is the largest amount of memory that can be read in a single call to **abswrite**.

Return value If it is successful, **abswrite** returns 0.

On error, the routine returns –1 and sets *errno* to the value of the AX register returned by the system call. See the DOS documentation for the interpretation of *errno*.

Portability **abswrite** is unique to DOS.

See also **absread, biosdisk**

access

Function Determines accessibility of a file.

Syntax int access(const char **filename*, int *amode*);

Prototype in io.h

Remarks

access checks the file named by *filename* to determine if it exists, and whether it can be read, written to, or executed.

The list of *amode* values is as follows:

06 Check for read and write permission.
04 Check for read permission.
02 Check for write permission.
01 Execute (ignored).
00 Check for existence of file.

Note: Under DOS, all existing files have read access (*amode* equals 04), so 00 and 04 give the same result. In the same vein, *amode* values of 06 and 02 are equivalent because under DOS write access implies read access.

If *filename* refers to a directory, **access** simply determines whether the directory exists.

Return value

If the requested access is allowed, **access** returns 0; otherwise, it returns a value of –1, and *errno* is set to one of the following:

ENOENT	Path or file name not found
EACCES	Permission denied

Portability

access is available on UNIX systems.

See also

chmod, fstat, stat

Example

```
#include <stdio.h>
#include <io.h>

/* Returns 1 if file name exists, else 0 */
int file_exists(char *filename)
{
   return (access(filename, 0) == 0);
}

main()
{
   printf("Does NOTEXIST.FIL exist: %s\n",
          file_exists("NOTEXIST.FIL") ? "YES" : "NO");
}
```

Program output

```
Does NOTEXIST.FIL exist: NO
```

acos

Function Calculates the arc cosine.

Syntax

```
#include <math.h>
double acos(double x);
```

Prototype in math.h

Remarks **acos** returns the arc cosine of the input value. Arguments to **acos** must be in the range –1 to 1. Arguments outside that range will cause **acos** to return 0 and set *errno* to

EDOM	Domain error

Return value **acos** returns a value in the range 0 to *pi*.

Error-handling for this routine can be modified through the function **matherr**.

Portability **acos** is available on UNIX systems and is compatible with ANSI C.

See also **asin, atan, atan2, cos, cosh, matherr, sin, sinh, tan, tanh**

allocmem

Function Allocates DOS memory segment.

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

Prototype in dos.h

Remarks **allocmem** uses the DOS system call 0x48 to allocate a block of free memory and returns 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. No assignment is made to the word pointed to by *segp* if not enough room is available.

All allocated blocks are paragraph-aligned.

Return value **allocmem** returns –1 on success. In the event of error, a number (the size in paragraphs of the largest available block) is returned.

An error return from **allocmem** will set *_doserrno* and will set the global variable *errno* to

ENOMEM Not enough memory

Portability **allocmem** is unique to DOS.

See also **coreleft, freemem, malloc, setblock**

---

arc

Function Draws a circular arc.

Syntax

```
#include <graphics.h>
void far arc(int x, int y, int stangle,
             int endangle, int radius);
```

Prototype in graphics.h

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** will draw a complete circle.

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

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

Note: If you are using a CGA in high resolution mode or a monochrome graphics adapter, the examples in this book that show how to use graphics functions may not produce the expected results. If your system runs on a CGA or monochrome adapter, pass the value 1 to those functions (**setcolor, setfillstyle**, and **setlinestyle**, for example) that alter the fill or drawing color, instead of a symbolic color constant (defined in graphics.h). See the second example given here on how to use the **arc, circle,**

ellipse, getarccoords, getaspectratio, and **pieslice** functions with a CGA or monochrome adapter.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **circle, ellipse, fillellipse, getarccoords, sector**

Example Graphics functions on an EGA or VGA adapter

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* Will request autodetection */
   int graphdriver = DETECT, graphmode;
   struct arccoordstype arcinfo;
   int xasp, yasp;
   long xlong;

   /* Initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* Draw a 90 degree arc with radius of 50 */
   arc(150, 150, 0, 89, 50);

   /* Get the coordinates of the arc and connect ends */
   getarccoords(&arcinfo);
   line(arcinfo.xstart, arcinfo.ystart, arcinfo.xend,
        arcinfo.yend);

   /* Draw a circle */
   circle(150, 150, 100);

   /* Draw an ellipse inside the circle */
   ellipse(150, 150, 0, 359, 100, 50);

   /* Draw and fill a pieslice */
   /* white outline */
   setcolor(WHITE);
   setfillstyle(SOLID_FILL, LIGHTRED);
   pieslice(100, 100, 0, 135, 49);
   setfillstyle(SOLID_FILL, LIGHTBLUE);
   pieslice(100, 100, 135, 225, 49);
   setfillstyle(SOLID_FILL, WHITE);
   pieslice(100, 100, 225, 360, 49);

   /* Draw a "square" rectangle */
   getaspectratio(&xasp, &yasp);
   xlong = (100L * (long)yasp) / (long)xasp;
```

```
  rectangle(0, 0, (int)xlong, 100);
  getch();
  closegraph();
}
```

Example 2

Graphics functions on a CGA or monochrome graphics adapter.

```
#include <graphics.h>
#include <conio.h>

main()
{
  int graphdriver = DETECT, graphmode;

  struct arccoordstype    arcinfo;
  int     xasp, yasp;
  long    xlong;

  initgraph(&graphdriver, &graphmode, "");

  /* Draw a 90 degree arc with radius of 50 */
  arc( 100, 120, 0, 89, 50 );

  /* Get the coordinates of the arc and connect ends */
  getarccoords( & arcinfo );
  line( arcinfo.xstart, arcinfo.ystart, arcinfo.xend,
        arcinfo.yend );

  /* Draw a circle */
  circle( 100, 120, 80 );

  /* Draw an ellipse inside the circle */
  ellipse( 100, 120, 0, 359, 80, 20 );

  /* Draw and fill a pieslice */
  setfillstyle( HATCH_FILL, 1 );
  pieslice( 200,  50, 0, 134, 49 );
  setfillstyle( SLASH_FILL, 1 );
  pieslice( 200,  50, 135, 225, 49 );
  setfillstyle( WIDE_DOT_FILL, 1 );
  pieslice( 200,  50, 225, 360, 49 );

  /* Draw a "square" rectangle */
  getaspectratio( & xasp, & yasp );
  xlong = ( 50L * (long) yasp ) / (long) xasp;
  rectangle( 0, 0, (int) xlong, 50 );
  getch();
  closegraph();
}
```

asctime

Function Converts date and time. to ASCII

Syntax
#include <time.h>
char *asctime(const struct tm **tblock*);

Prototype in time.h

Remarks **asctime** converts a time stored as a structure in **tblock* to a 26-character string of the same form as the **ctime** string:

```
Sun Sep 16 01:03:52 1973\n\0
```

Return value **asctime** returns a pointer to the character string containing the date and time. This string is a static variable that is overwritten with each call to **asctime**.

Portability **asctime** is available on UNIX systems and is compatible with ANSI C.

See also **ctime, difftime, ftime, gmtime, localtime, stime, time, tzset**

Example

```
#include <stdio.h>
#include <time.h>

main()
{
   struct tm *tm_now;
   time_t secs_now;
   char *str_now;
   /* get time in seconds */
   time(&secs_now);
   /* make it a string */
   str_now = ctime(&secs_now);
   printf("The number of seconds since"
          "Jan 1, 1970 is %ld\n", secs_now);
   printf("In other words, the current time"
          "is %s\n", str_now);
   /* make it a structure */
   tm_now = localtime(&secs_now);
   printf("From the structure: day %d"
          "%02d-%02d-%02d %02d:%02d:%02d\n",
           tm_now->tm_yday, tm_now->tm_mon,
           tm_now->tm_mday, tm_now->tm_year,
           tm_now->tm_hour, tm_now->tm_min,
           tm_now->tm_sec);
```

```
   /* from structure to string */
   str_now = asctime(tm_now);
   printf("Once more, the current time is"
          "%s\n", str_now);
}
```

Program output

```
The number of seconds since Jan 1, 1970 is 315594553.
In other words, the current time is Tue Jan 01 12:09:12 1980

From the structure: day 0  00-01-80 12:09:13
Once more, the current time is Tue Jan 01 12:09:12 1980
```

asin

Function Calculates the arc sine.

Syntax
#include <math.h>
double asin(double *x*);

Prototype in math.h

Remarks **asin** returns the arc sine of the input value. Arguments to **asin** must be in the range –1 to 1. Arguments outside that range will cause **asin** to return 0 and set *errno* to

EDOM Domain error

Return value **asin** returns a value in the range *–pi*/2 to *pi*/2.

Error-handling for this routine can be modified through the function **matherr**.

Portability **asin** is available on UNIX systems and is compatible with ANSI C.

See also **acos, atan, atan2, cos, cosh, matherr, sin, sinh, tan, tanh**

assert

Function

Tests a condition and possibly aborts.

Syntax

#include <assert.h>
#include <stdio.h>
void assert(int *test*);

Prototype in

assert.h

Remarks

assert is a macro that expands to an **if** statement; if *test* evaluates to zero, **assert** prints a message on *stderr* and aborts the program (via a call to **abort**).

assert prints this message:

```
Assertion failed: <test>, file <filename>, line <linenum>
```

The *filename* and *linenum* listed in the message are the source file name and line number where the **assert** macro appears.

If you place the `#define NDEBUG` directive ("no debugging") in the source code before the `#include <assert.h>` directive, the effect is to comment out the **assert** statement.

Return value

None.

Portability

assert is available on some UNIX systems, including Systems III and V, and is compatible with ANSI C.

See also

abort

Example

```
/* ASSERTST.C: Add an item to a list,
   verify that the item is not NULL */
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

struct ITEM {
   int key;
   int value;
}

main()
{
   additem(NULL);
}
```

```
void additem(struct ITEM *itemptr) {
   assert(itemptr != NULL);             /* this is line 12 */
   /* ... add the item ... */
}
```

Program output

```
Assertion failed: itemptr != NULL,
file C:\TURBOC\ASSERTST.C, line 12
```

atan

Function Calculates the arc tangent.

Syntax #include <math.h>
double atan(double *x*);

Prototype in math.h

Remarks **atan** calculates the arc tangent of the input value.

Return value **atan** returns a value in the range *–pi*/2 to *pi*/2.

Error-handling for this routine can be modified through the function **matherr**.

Portability **atan** is available on UNIX systems and is compatible with ANSI C.

See also **acos, asin, atan2, cos, cosh, matherr, sin, sinh, tan, tanh**

atan2

Function Calculates the arc tangent of *y*/*x*.

Syntax #include <math.h>
double atan2(double *y*, double *x*);

Prototype in math.h

Remarks **atan2** returns the arc tangent of *y*/*x* and will produce correct results even when the resulting angle is near *pi*/2 or *–pi*/2 (*x* near 0).

If both *x* and *y* are set to 0, the function sets *errno* to EDOM.

Return value

atan2 returns a value in the range *–pi* to *pi*.

Error-handling for this routine can be modified through the function **matherr**.

Portability

atan2 is available on UNIX systems and is compatible with ANSI C.

See also

acos, asin, atan, cos, cosh, matherr, sin, sinh, tan, tanh

atexit

Function

Registers termination function.

Syntax

#include <stdlib.h>
int atexit(atexit_t *func*)

Prototype in

stdlib.h

Remarks

atexit registers the function pointed to by *func* as an exit function. Upon normal termination of the program, **exit** calls (**func*)() just before returning to the operating system. The called function is of type *atexit_t*, which is defined in a **typedef** in stdlib.h.

Each call to **atexit** registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed).

Return value

atexit returns 0 on success and nonzero on failure (no space left to register the function).

Portability

atexit is compatible with ANSI C.

See also

abort, _exit, exit, spawn...

Example

```
#include <stdlib.h>
#include <stdio.h>

atexit_t exit_fn1(void)
{
   printf("Exit Function 1 called\n");
}

atexit_t exit_fn2(void)
{
   printf("Exit Function 2 called\n");
```

```
}

main()
{
   /* post exit_fn1 */
   atexit(exit_fn1);
   /* post exit_fn2 */
   atexit(exit_fn2);
   printf("Main quitting ...\n");
}
```

Program output

```
Main quitting ...
Exit Function 2 called
Exit Function 1 called
```

atof

Function

Converts a string to a floating-point number.

Syntax

```
#include <math.h>
double atof(const char *s);
```

Prototype in

math.h, stdlib.h

Remarks

atof converts a string pointed to by *s* to **double**; this function recognizes the character representation of a floating-point number, made up of the following:

- an optional string of tabs and spaces
- an optional sign
- a string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
- an optional *e* or *E* followed by an optional signed integer

The characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

atof also recognizes +INF and –INF for plus and minus infinity, and +NAN and –NAN for Not-a-Number.

In this function, the first unrecognized character ends the conversion.

Return value **atof** returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**double**), the return value is 0.

If there is an overflow, **atof** returns plus or minus HUGE_VAL, and **matherr** is not called.

Portability **atof** is available on UNIX systems and is compatible with ANSI C.

See also **atoi, atol, ecvt, fcvt, gcvt, strtod**

---

atoi

Function Converts a string to an integer.

Syntax int atoi(const char *s);

Prototype in stdlib.h

Remarks **atoi** converts a string pointed to by *s* to **int**; **atoi** recognizes, in the following order,

- an optional string of tabs and spaces
- an optional sign
- a string of digits

The characters must match this generic format:

```
[ws] [sn] [ddd]
```

In this function, the first unrecognized character ends the conversion.

There are no provisions for overflow in **atoi**.

Return value **atoi** returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**int**), the return value is 0.

Portability **atoi** is available on UNIX systems and is compatible with ANSI C.

See also **atof, atol, ecvt, fcvt, gcvt**

atol

Function Converts a string to a long.

Syntax long atol(const char **s*);

Prototype in stdlib.h

Remarks **atol** converts the string pointed to by *s* to **long**. **atol** recognizes, in the following order,

- an optional string of tabs and spaces
- an optional sign
- a string of digits

The characters must match this generic format:

```
[ws] [sn] [ddd]
```

In this function, the first unrecognized character ends the conversion.

There are no provisions for overflow in **atol**.

Return value **atol** returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**long**), the return value is 0.

Portability **atol** is available on UNIX systems and is compatible with ANSI C.

See also **atof, atoi, ecvt, fcvt, gcvt, strtol, strtoul**

bar

Function Draws a two-dimensional bar.

Syntax #include <graphics.h>
void far bar(int *left*, int *top*, int *right*, int *bottom*);

Prototype in graphics.h
#include <conio.h>

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **bar3d, rectangle, setcolor, setfillstyle**

Example

```
#include <graphics.h>
main()
{
   /* Will request autodetection */
   int graphdriver = DETECT, graphmode;
   /* Initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   setfillstyle(SOLID_FILL, MAGENTA);
   bar3d(100, 10, 200, 100, 5, 1);
   setfillstyle(HATCH_FILL, RED);
   bar(30, 30, 80, 80);

   getche();
   closegraph();
}
```

bar3d

Function Draws a 3-D bar.

Syntax

#include <graphics.h>
void far bar3d(int *left,* int *top,* int *right,*
int *bottom,* int *depth,* int *topflag*);

Prototype in graphics.h

Remarks **bar3d** draws a three-dimensional rectangular bar, then fills it in 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **bar, rectangle, setcolor, setfillstyle, setlinestyle**

Example See **bar**

bdos

Function DOS system call.

Syntax int bdos(int *dosfun*, unsigned *dosdx*, unsigned *dosal*);

Prototype in dos.h

Remarks **bdos** provides direct access to many of the DOS system calls. See the *MS-DOS Programmer's Reference Manual* for details of each system call.

Those system calls that require an integer argument use **bdos.**

In the large data models (compact, large, and huge), it is important to use **bdosptr** instead of **bdos** for system calls that require a pointer as the call argument.

dosfun is defined in the *MS-DOS Programmer's Reference Manual.*

dosdx is the value of register DX.

dosal is the value of register AL.

Return value The return value of **bdos** is the value of AX set by the system call.

Portability **bdos** is unique to DOS.

See also **bdosptr, geninterrupt, int86, int86x, intdos, intdosx**

Example

```
#include <stdio.h>
#include <dos.h>

/* Get current drive as 'A', 'B', ... */
char current_drive(void)
{
   char curdrive;
   /* Get current disk as 0, 1,...*/
   curdrive = bdos(0x19,0,0);
   return( 'A' + curdrive );
}

main()
{
   printf("The current drive is %c:\n", current_drive());
}
```

Program output

```
The current drive is C:
```

bdosptr

Function DOS system call.

Syntax int bdosptr(int *dosfun*, void **argument*,
unsigned *dosal*);

Prototype in dos.h

Remarks **bdosptr** provides direct access to many of the DOS system calls. See the *MS-DOS Programmer's Reference Manual* for details of each system call.

Those system calls that require a pointer argument use **bdosptr**.

In the large data models (compact, large, and huge), it is important to use **bdosptr** for system calls that require a pointer as the call argument.

dosfun is defined in the *MS-DOS Programmer's Reference Manual*.

In the small data models, the *argument* parameter to **bdosptr** specifies DX; in the large data models, it gives the DS:DX values to be used by the system call.

dosal is the value of register AL.

Return value The return value of **bdosptr** is the value of AX on success, or –1 on failure. On failure, *errno* and *_doserrno* are set.

Portability **bdosptr** is unique to DOS.

See also **bdos, geninterrupt, int86, int86x, intdos, intdosx**

Example See **harderr**

bioscom

Function Performs serial I/O.

Syntax int bioscom(int *cmd*, char *abyte*, int *port*);

Prototype in bios.h

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
0x03	8 data bits
0x00	1 stop bit
0x04	2 stop bits
0x00	No parity

0x08	Odd parity
0x18	Even parity

0x00	110 baud
0x20	150 baud
0x40	300 baud
0x60	600 baud
0x80	1200 baud
0xA0	2400 baud
0xC0	4800 baud
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. Otherwise, the remaining upper and lower bits are appropriately set.

With a *cmd* value of 2, the byte read is in the lower bits of the return value if there was no error. If an error occurred, at least one of the upper bits is set. If no upper bits are set, 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

Portability

bioscom works with IBM PCs and compatibles only.

Example

```
/* bioscom example - Dumb Terminal Demo */

#include <bios.h>
#include <conio.h>

#define COM1 0
#define DATA_READY 0x100
/* 1200 baud, 7 bits, 1 stop, no parity */
#define SETTINGS (0x80|0x02|0x00|0x00)

main()
{
   int register in, out, status;

   bioscom(0, SETTINGS, COM1);

   cprintf("... BIOSCOM [ESC] to exit ...\n");

   while (1)
   {
      status = bioscom(3, 0, COM1);
      if (status & DATA_READY)
         if ( (out = bioscom(2, 0, COM1) & 0x7F) != 0 )
            putch(out);
         if (kbhit())
         {
            if ( (in = getch()) == '\x1B')
               return(0);
            bioscom(1, in, COM1);
         }
   }
}
```

biosdisk

Function BIOS disk services.

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

Prototype in bios.h

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, etc. 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 may or may not be needed.

Here are the possible values for *cmd* for the IBM PC, XT, AT, or PS/2, or any compatible system:

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:

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.

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, and it can destroy file contents and directories on a hard disk.

Return value

biosdisk returns a status byte composed of the following bits:

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	DMA overrun.
0x09	Attempt to 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.

Note that 0x11 is not an error because the data is correct. The value is returned anyway to give the application an opportunity to decide for itself.

Portability **biosdisk** works with IBM PCs and compatibles only.

See also **absread, abswrite**

biosequip

Function Checks equipment.

Syntax int biosequip(void);

Prototype in bios.h

Remarks **biosequip** returns an integer describing the equipment connected to the system. BIOS interrupt 0x11 is used for this.

Return value The return value is interpreted as a collection of bit-sized fields. The IBM PC values follow:

Bits 14-15	Number of parallel printers installed
Bit 13	Serial printer attached
Bit 12	Game I/O attached

Bits 9-11	Number of send ports
Bit 8	Not DMA 0 = Machine has DMA. 1 = Machine does not have DMA; for example, PC Jr.
Bits 6-7	Number of disks 00 = 1 drive 01 = 2 drives 10 = 3 drives 11 = 4 drives, only if bit 0 is 1
Bit 5 Bit 4	Initial Video mode 00 = Unused 01 = 40x25 BW with color card 10 = 80x25 BW with color card 11 = 80x25 BW with mono card
Bits 2-3 Bit 2	Motherboard RAM size RAM size 00 = 16K 01 = 32K 10 = 48K 11 = 64K
Bit 1 Bit 0	Floating-point coprocessor Boot from disk

Portability **biosequip** works with IBM PCs and compatibles only.

bioskey

Function Keyboard interface, using BIOS services directly.

Syntax int bioskey(int *cmd*);

Prototype in bios.h

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*:

cmd	Task Performed by bioskey
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 struck at the keyboard. If the lower 8 bits are zero, the upper 8 bits are the extended keyboard codes defined in the *Technical Reference Manual* for the IBM PC.
1	This tests whether a keystroke is available to be read. A return value of zero means no key is available. 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 composed from 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	*Left Shift* pressed
Bit 0	0x01	*Right Shift* pressed

Portability

bioskey works with IBM PCs and compatibles only.

Example

```
#include <stdio.h>
#include <bios.h>
#include <ctype.h>

#define RIGHT 0x0001
#define LEFT  0x0002
#define CTRL  0x0004
#define ALT   0x0008

main ()
{
```

```
   int key; int modifiers;

   /* Function 1 returns 0 until a key is struck. Wait
      for an input by repeatedly checking for a key. */
   while(bioskey(1) == 0) ;

   /* Now use function 0 to get the return value of
      the key. */
   key = bioskey(0);
   printf("Key Pressed was ");

   /* Use function 2 to determine if shift keys were used */
   modifiers = bioskey(2);
   if (modifiers) {
      printf("[");
      if (modifiers & RIGHT) printf("RIGHT ");
      if (modifiers & LEFT ) printf("LEFT ");
      if (modifiers & CTRL ) printf("CTRL ");
      if (modifiers & ALT  ) printf("ALT ");
      printf("] ");
   }

   if (isalnum(key & 0xFF))
      printf("'%c'\n",key);
   else
      printf("%#02x\n",key);
}
```

Program output

```
Key Pressed was [LEFT ] 'T'
```

biosmemory

Function Returns memory size.

Syntax int biosmemory(void);

Prototype in bios.h

Remarks **biosmemory** returns the size of RAM memory using BIOS interrupt 0x12. This does not include display adapter memory, extended memory, or expanded memory.

Return value **biosmemory** returns the size of RAM memory in 1K blocks.

Portability **biosmemory** works with IBM PCs and compatibles only.

biosprint

Function Printer I/O using BIOS services directly.

Syntax int biosprint(int *cmd*, int *abyte*, int *port*);

Prototype in bios.h

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 composed 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

Portability **biosprint** works with IBM PCs and compatibles only.

biostime

Function reads or sets the BIOS timer

Syntax long biostime(int *cmd*, long *newtime*);

Prototype in bios.h

Remarks **biostime** either reads or sets the BIOS timer. This is a timer counting ticks since midnight at a rate of roughly 18.2 ticks per second. **biostime** uses BIOS interrupt 0x1A.

If *cmd* equals 0, **biostime** returns the current value of the timer. If *cmd* equals 1, the timer is set to the **long** value in *newtime*.

Return value When **biostime** reads the BIOS timer (*cmd* = 0), it returns the timer's current value.

Portability **biostime** works with IBM PCs and compatibles only.

---

brk

Function Changes data-segment space allocation.

Syntax int brk(void **addr*);

Prototype in alloc.h

Remarks **brk** is used to change dynamically the amount of space allocated to the calling program's data segment. 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 *errno* is set to

ENOMEM Not enough memory

Portability **brk** is available on UNIX systems.

See also **coreleft, sbrk**

bsearch

Function

Binary search of an array.

Syntax

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nelem, size_t width,
              int (*fcmp)(const void *, const void *));
```

Prototype in

stdlib.h

Remarks

bsearch searches a table (array) of *nelem* elements in memory, and returns the address of the first entry in the table that matches the search key. If no match is found, **bsearch** returns 0.

The type *size_t* is defined as an unsigned integer.

- *nelem* gives the number of elements in the table.
- *width* specifies the number of bytes in each table entry.

The comparison routine **fcmp* is called with two arguments: *elem1* and *elem2*. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (**elem1* and **elem2*), and returns an integer based on the results of the comparison.

For **bsearch**, the **fcmp* return value is

```
<   0 if *elem1 <  *elem2
==  0 if *elem1 == *elem2
>   0 if *elem1 >  *elem2
```

Typically, *elem1* is the argument *key*, and *elem2* is a pointer to an element in the table being searched.

Return value

bsearch returns the address of the first entry in the table that matches the search key. If no match is found, **bsearch** returns 0.

Portability

bsearch is available on UNIX systems and is compatible with ANSI C.

See also

lfind, lsearch, qsort

Example

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))

int numarray[] = { 123, 145, 512, 627, 800, 993 };

int numeric(int *p1, int *p2)
{
   return(*p1 - *p2);
}

/* Return 1 if key is in the table, 0 if not */
int lookup(int key)
{
   int *itemptr;

   /* bsearch() returns a pointer to the
      item that is found */
   itemptr = (int *)
      bsearch(&key, numarray, NELEMS(numarray),
              sizeof(int), numeric);
   return (itemptr != NULL);
}

main()
{
   printf("Is 512 in table? ");
   printf("%s\n", lookup(512) ? "YES" : "NO");
}
```

Program output

```
Is 512 in table? YES
```

cabs

Function Absolute value of complex number.

Syntax #include <math.h>
double cabs(struct complex *z*);

Prototype in math.h

Remarks **cabs** is a macro that calculates the absolute value of *z*, a complex number. *z* is a structure with type **complex**; the structure is defined in math.h as

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

where *x* is the real part and *y* is the imaginary part.

Calling **cabs** is equivalent to calling **sqrt** with the real and imaginary components of *z*, as shown here:

```
sqrt(z.x * z.x + z.y * z.y)
```

If you want to use the function instead of the macro, include

```
#undef cabs
```

in your program.

Return value **cabs** returns the absolute value of *z*, a **double.** On overflow, **cabs** returns HUGE_VAL and sets *errno* to

ERANGE Result out of range

Error-handling for **cabs** can be modified through the function **matherr.**

Portability **cabs** is available on UNIX systems.

See also **abs, fabs, labs, matherr**

calloc

Function Allocates main memory.

Syntax #include <stdlib.h>
void *calloc(size_t *nitems*, size_t *size*);

Prototype in stdlib.h, alloc.h

Remarks **calloc** provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

All the space between the end of the data segment and the top of the program stack is available for use in the small data models (tiny, small, and medium), except for a 256-byte margin immediately before the top of the stack. This margin is intended to allow the application some room to grow on the stack, plus a small amount needed by DOS.

In the large data models (compact, large, and huge), all space beyond the program stack to the end of physical memory is available for the heap.

calloc allocates a block of size *nitems* × *size*. The block is cleared to 0.

Return value **calloc** returns a pointer to the newly allocated block. If not enough space exists for the new block, or *nitems* or *size* is 0, **calloc** returns NULL.

Portability **calloc** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **farcalloc, free, malloc, realloc**

ceil

Function Rounds up.

Syntax
```
#include <math.h>
double ceil(double x);
```

Prototype in math.h

Remarks **ceil** finds the smallest integer not less than *x*.

Return value **ceil** returns the integer found (as a **double**).

Portability **ceil** is available on UNIX systems and is compatible with ANSI C.

See also **floor, fmod**

cgets

Function Reads string from console.

Syntax
```
char *cgets(char *str);
```

Prototype in conio.h

Remarks **cgets** reads a string of characters from the console, storing the string (and the string length) in the location pointed to by *str*.

cgets reads characters until it encounters a CR/LF combination, or until the maximum allowable number of characters have been read. If **cgets** reads a CR/LF combination, it replaces the combination with a \0 (null terminator) before storing the string.

Before **cgets** is called, *str*[0] should be set to the maximum length of the string to be read. On return, *str*[1] is set to the number of characters actually read. The characters read start at *str*[2] and end with a null terminator. Thus, *str* must be at least *str*[0] plus 2 bytes long.

Return value On success, **cgets** returns a pointer to *str*[2]. There is no error return.

Portability This function works only with IBM PCs and compatibles equipped with supplied graphics display adapters.

See also **fgets, getch, getche, gets**

Example

```
#include <stdio.h>
#include <conio.h>

main()
{
   char buffer[82];
   char *p;
   buffer[0] = 80;        /* There's space for 80 characters */
   p = cgets(buffer);
   printf("/ncgets got %d characters: \"%s\"\n",
         buffer[1], p);
   printf("The returned pointer is %p,
         buffer[2] is at %p\n", p, &buffer)
   buffer[0] = 5             /* Leave space for 5 chars only */
   p = cgets(buffer);
   printf("/ncgets got %d characters: \"%s\"\n",
         buffer[1], p);
   printf("The returned pointer is %p, buffer[2] is at %p\n",
         p, &buffer)
}
```

Program output

```
abcdfghijklm
cgets got 12 characters: "abcdfghijklm"
The returned pointer is FEF6, buffer[2] is at FEF6
abcd
cgets got 4 characters: "abcd"
```

```
the returned pointer is FEF6, buffer[2] is at FEF6
```

chdir

Function Changes current directory.

Syntax int chdir(const char **path*);

Prototype in dir.h

Remarks **chdir** causes the directory specified by *path* to become the current working directory. *path* must specify an existing directory.

A drive can also be specified in the *path* argument, such as

```
chdir("a:\\turboc")
```

but this changes only the current directory on that drive; it doesn't change the active drive.

Return value Upon successful completion, **chdir** returns a value of 0. Otherwise, it returns a value of –1, and *errno* is set to

ENOENT Path or file name not found

Portability **chdir** is available on UNIX systems.

See also **getcurdir, getcwd, mkdir, rmdir, system**

_chmod

Function Changes file access mode.

Syntax #include <dos.h>
#include <io.h>
int _chmod(const char **path*, int *func* [, int *attrib*]);

Prototype in io.h

Remarks The **_chmod** function may either fetch or set the DOS file attributes. If *func* is 0, the function returns the current DOS attributes for the file. If *func* is 1, the attribute is set to *attrib*.

attrib can be one of the following symbolic constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return value Upon successful completion, **_chmod** returns the file attribute word; otherwise, it returns a value of –1.

In the event of an error, *errno* is set to one of the following:

ENOENT	Path or file name not found
EACCES	Permission denied

Portability **_chmod** is unique to DOS.

See also **chmod, _creat**

chmod

Function Changes file access mode.

Syntax

```
#include <sys\stat.h>
int chmod(const char *path, int amode);
```

Prototype in io.h

Remarks **chmod** sets the file-access permissions of the file given by *filename* according to the mask given by *amode*. *filename* points to a string; **filename* is the first character of that string.

amode can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\stat.h).

Value of *amode*	Access Permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD | S_IWRITE	Permission to read and write

Return value Upon successfully changing the file-access mode, **chmod** returns 0. Otherwise, **chmod** returns a value of –1.

In the event of an error, *errno* is set to one of the following:

ENOENT	Path or file name not found
EACCES	Permission denied

Portability **chmod** is available on UNIX systems.

See also **access, _chmod, fstat, open, sopen, stat**

Example

```
#include <stdio.h>
#include <sys\stat.h>
#include <io.h>

void make_read_only(char *filename)
{
   int stat;
   stat = chmod(filename, S_IREAD);
   if (stat)
      printf("couldn't make %s
             "read-only\n", filename);
   else
      printf("made %s read-only\n", filename);
}

main()
{
   make_read_only("NOTEXIST.FIL");
   make_read_only("MYFILE.FIL");
}
```

Program output

```
Couldn't make NOTEXIST.FIL read-only
made MYFILE.FIL read-only
```

chsize

Function Changes file size.

Syntax int chsize(int *handle*, long *size*);

Prototype in io.h

Remarks

chsize changes the size of the file associated with *handle*. It can truncate or extend the file, depending on the value of *size* compared to the file's original size.

The mode in which you open the file must allow writing.

If **chsize** extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

Return value

On success, **chsize** returns 0. On failure, it returns -1 and *errno* is set to one of the following:

EACCESS	Permission denied
EBADF	Bad file number
ENOSPC	UNIX—not DOS

Portability

chsize is unique to DOS.

See also

close, _creat, creat, open

circle

Function

Draws a circle of the given radius at (*x*,*y*).

Syntax

```
#include <graphics.h>
void far circle(int x, int y, int radius);
```

Prototype in

graphics.h

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 pieslices. Only the *thickness* parameter is used.

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

Return value

None.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

Examples See **arc**

_clear87

Function Clears floating-point status word.

Syntax unsigned int _clear87 (void);

Prototype in float.h

Remarks **_clear87** clears the floating-point status word, which is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler.

Return value The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.

See also **_control87, _fpreset, _status87**

Example See **_control87**

cleardevice

Function Clears the graphics screen.

Syntax

```
#include <graphics.h>
void far cleardevice(void);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PC's and compatibles equipped with supported graphics display adapters.

See also **clearviewport**

clearerr

Function Resets error indication.

Syntax #include <stdio.h>
void clearerr(FILE **stream*);

Prototype in stdio.h

Remarks **clearerr** resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations will continue to return error status until a call is made to **clearerr** or **rewind**.

The end-of-file indicator is reset with each input operation.

Return value None.

Portability **clearerr** is available on UNIX systems and is compatible with ANSI C.

See also **eof, feof, ferror, perror, rewind**

clearviewport

Function Clears the current viewport.

Syntax #include <graphics.h>
void far clearviewport(void);

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **cleardevice, getviewsettings, setviewport**

Example

```
#include <graphics.h>

main()
{
/* will request autodetection */
```

```
   int graphdriver = DETECT, graphmode;
   setviewport(30, 30, 130, 130, 0);
   outtextxy(10, 10, "Hit any key to clear viewport ...");
/* get a key */
   getch();
/* clear viewport when key is hit */
   clearviewport();
   closegraph();
}
```

clock

Function Determines processor time

Syntax
#include <time.h>
clock_t clock(void);

Prototype in time.h

Remarks **clock** can be used to determine the time interval between two events.

To determine the time in seconds, the value returned by **clock** should be divided by the value of the macro **CLK_TCK**.

Return value The **clock** function returns the processor time elapsed since the beginning of the program invocation. If the processor time is not available or its value cannot be represented, the function returns the value –1.

Portability **clock** is compatible with ANSI C.

Example

```
#include <time.h>
#include <stdio.h>

void main()
{
   clock_t start, end;

   start = clock();

/* Code to be timed goes here */

   end = clock();
   printf("The time was: %f\n", (end - start) / CLK_TCK);
}
```

_close

Function Closes a file.

Syntax int _close(int *handle*);

Prototype in io.h

Remarks **_close** closes the file associated with *handle*. *handle* is a file handle obtained from a **_creat**, **creat**, **creatnew**, **creattemp**, **dup**, **dup2**, **_open**, or **open** call.

Note: This function does not write a *Ctrl-Z* character at the end of the file. If you want to terminate the file with a *Ctrl-Z*, you must explicitly output one.

Return value Upon successful completion, **_close** returns 0. Otherwise, it returns a value of –1.

_close fails if *handle* is not the handle of a valid, open file, and *errno* is set to

EBADF	Bad file number

Portability **_close** is unique to DOS.

See also **close**, **_creat**, **open**, **read**, **write**

close

Function Closes a file.

Syntax int close(int *handle*);

Prototype in io.h

Remarks **close** closes the file associated with *handle*, a file handle obtained from a **_creat**, **creat**, **creatnew**, **creattemp**, **dup**, **dup2**, **_open**, or **open** call.

Note: This function does not write a *Ctrl-Z* character at the end of the file. If you want to terminate the file with a *Ctrl-Z*, you must explicitly output one.

Return value Upon successful completion, **close** returns 0. Otherwise, a value of –1 is returned.

close fails if *handle* is not the handle of a valid, open file, and *errno* is set to

EBADF Bad file number

Portability **close** is available on UNIX systems.

See also **chsize, _close, creat, creatnew, dup, fclose, open, sopen**

closegraph

Function Shuts down the graphics system.

Syntax

```
#include <graphics.h>
void far closegraph(void);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **initgraph, setgraphbufsize**

clreol

Function Clears to end of line in text window.

Syntax

```
void clreol(void);
```

Prototype in conio.h

Remarks **clreol** clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor.

Return value None.

Portability **clreol** works with IBM PCs and compatibles only.

See also	**clrscr, delline, window**

clrscr

Function	Clears the text mode window.
Syntax	void clrscr(void);
Prototype in	conio.h
Remarks	**clrscr** clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).
Return value	None.
Portability	**clrscr** works with IBM PCs and compatibles only.
See also	**clreol, delline, window**

_control87

Function Manipulates the floating-point control word.

Syntax

```
unsigned int _control87(unsigned int new,
                        unsigned int mask);
```

Prototype in float.h

Remarks

_control87 retrieves or changes the floating-point control word.

The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package, namely, the precision, infinity, and rounding modes. Changing these modes allows you to mask or unmask floating-point exceptions.

_control87 matches the bits in *mask* to the bits in *new*. If a *mask* bit equals 1, the corresponding bit in *new* contains the new value for the same bit in the floating-point control word, and **_control87** sets that bit in the control word to the new value.

Here's a simple illustration:

Original control word:	0100	0011	0110	0011
mask	1000	0001	0100	1111
new	1110	1001	0000	0101
Changing bits	1*xxx*	*xxx*1	*x*0*xx*	0101

If *mask* equals 0, **_control87** returns the floating-point control word without altering it.

Return value

The bits in the value returned reflect the new floating-point control word. For a complete definition of the bits returned by **_control87**, see the header file float.h.

See also

_clear87, _fpreset, signal, _status87

Example

```
/* _control87 example */

#include <math.h>
#include <float.h>
#include <stdio.h>

#define CW_NEW (CW_DEFAULT | EM_ZERODIVIDE | EM_OVERFLOW)
#define MASK_ALL (0xFFFF)

main()
{
   float a, b, c;

   _control87(CW_NEW|EM_INVALID, MASK_ALL);

   a = 1.0;
   b = 0.0;
   c = a/b;

   if(_status87() & SW_ZERODIVIDE)
   {
      fprintf(stderr, "DIVISION BY ZERO.\n");
      _clear87();
      return(1);
   }
}
```

coreleft

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);

Prototype in alloc.h

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 large data models, **coreleft** returns the amount of unused memory between the heap and the stack.

Portability **coreleft** is unique to DOS.

In the small data memory models, **coreleft** returns the amount of unused memory between the stack and the data segment minus 256 bytes.

See also **allocmem, brk, farcoreleft, malloc**

cos

Function Calculates the cosine.

Syntax #include <math.h>
double cos(double *x*);

Prototype in math.h

Remarks **cos** returns the cosine of the input value. The angle is specified in radians.

Return value **cos** returns a value in the range –1 to 1.

Error-handling for this routine can be modified through the function **matherr**.

Portability — **cos** is available on UNIX systems and is compatible with ANSI C.

See also — **acos, asin, atan, atan2, cosh, matherr, sin, sinh, tan, tanh**

---

cosh

Function — Calculates the hyperbolic cosine.

Syntax

```
#include <math.h>
double cosh(double x);
```

Prototype in — math.h

Remarks — **cosh** computes the hyperbolic cosine for a real argument.

Return value — **cosh** returns the hyperbolic cosine of the argument.

When the correct value would create an overflow, **cosh** returns the value HUGE_VAL with the appropriate sign, and *errno* is set to ERANGE.

Error-handling for **cosh** can be modified through the function **matherr**.

Portability — **cosh** is available on UNIX systems and is compatible with ANSI C.

See also — **acos, asin, atan, atan2, cos, matherr, sin, sinh, tan, tanh**

---

country

Function — Returns country-dependent information.

Syntax

```
#include <dos.h>
struct country *country(int xcode, struct country *cp);
```

Prototype in — dos.h

Remarks — **country** specifies how certain country-dependent data, such as dates, times, and currency, will be formatted. The values set by this function depend on the DOS version being used.

If *cp* has a value of –1, the current country is set to the value of *xcode*, which must be nonzero. Otherwise, the **country** structure pointed to by *cp* is filled with the country-dependent information of the current country (if *xcode* is set to 0), or the country given by *xcode*.

The structure **country** is defined as follows:

```
struct country {
   int co_date;                                   /* date format */
   char co_curr[5];                           /* currency symbol */
   char co_thsep[2];                      /* thousands separator */
   char co_desep[2];                        /* decimal separator */
   char co_dtsep[2];                           /* date separator */
   char co_tmsep[2];                           /* time separator */
   char co_currstyle;                          /* currency style */
   char co_digits;              /* significant digits in currency */
   char co_time;                                  /* time format */
   long co_case;                                     /* case map */
   char co_dasep[2];                           /* data separator */
   char co_fill[10];                                   /* filler */
};
```

The date format in *co_date* is

- 0 for the U.S. style of month, day, year
- 1 for the European style of day, month, year
- 2 for the Japanese style of year, month, day

Currency display style is given by *co_currstyle*, as follows:

0 Currency symbol precedes value with no spaces between the symbol and the number.

1 Currency symbol follows value with no spaces between the number and the symbol.

2 Currency symbol precedes value with a space after the symbol.

3 Currency symbol follows the number with a space before the symbol.

Return value On success, **country** returns the pointer argument *cp*. On error it returns NULL.

Portability **country** is available only with DOS version 3.0 and greater.

cprintf

Function Writes formatted output to the screen.

Syntax int cprintf(const char **format*[, *argument*, ...]);

Prototype in conio.h

Remarks **cprintf** accepts a series of arguments, applies to each a format specification contained in the format string pointed to by *format*, and outputs the formatted data directly to the screen, to the current text window. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification. Unlike **fprintf** and **printf**, **cprintf** does not translate linefeed characters (`\n`) into carriage-return/linefeed character pairs (`\r\n`).

Return value **cprintf** returns the number of characters output.

Portability **cprintf** works with IBM PCs and compatibles only.

See also *directvideo* (variable), **fprintf**, **printf**, **putch**, **sprintf**, **vprintf**

Example See **printf**

cputs

Function Writes a string to the screen.

Syntax int cputs(const char **str*);

Prototype in conio.h

Remarks **cputs** writes the null-terminated string *str* to the current text window. It does not append a newline character.

The string is written directly to screen memory by way of a BIOS call, depending on the value of *directvideo*.

Unlike **puts**, **cputs** does not translate linefeed characters (`\n`) into carriage-return/linefeed character pairs (`\r\n`).

Return value **cputs** returns the last character printed.

Portability	**cputs** works with IBM PCs and compatibles only.
See also	*directvideo* (variable), **putch, puts**

_creat

Function Creates a new file or rewrites an existing one.

Syntax

```
#include <dos.h>
int _creat(const char *path, int attrib);
```

Prototype in io.h

Remarks **_creat** accepts *attribute*, a DOS attribute word. Any attribute bits can be set in this call. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists, its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The *attribute* argument to **_creat** can be one of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return value Upon successful completion, **_creat** returns the new file handle, a nonnegative integer; otherwise, it returns –1.

In the event of error, *errno* is set to one of the following:

ENOENT	Path or file name not found
EMFILE	Too many open files
EACCES	Permission denied

Portability **_creat** is unique to DOS.

See also **_chmod, chsize, _close, close, creat, creatnew, creattemp**

creat

Function Creates a new file or rewrites an existing one.

Syntax

```
#include <sys\stat.h>
int creat(const char *path, int amode);
```

Prototype in io.h

Remarks **creat** creates a new file or prepares to rewrite an existing file given by *path*. *amode* applies only to newly created files.

A file created with **creat** is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

If the file exists and the write attribute is set, **creat** truncates the file to a length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, the **creat** call fails, and the file remains unchanged.

The **creat** call examines only the S_IWRITE bit of the access-mode word *amode*. If that bit is 1, the file is writable. If the bit is 0, the file is marked as read-only. All other DOS attributes are set to 0.

amode can be one of the following (defined in sys\stat.h):

Value of *amode*	Access Permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD | S_IWRITE	Permission to read and write

Note: In DOS, write permission implies read permission.

Return value Upon successful completion, **creat** returns the new file handle, a nonnegative integer; otherwise, it returns –1.

In the event of error, *errno* is set to one of the following:

ENOENT	Path or file name not found
EMFILE	Too many open files
EACCES	Permission denied

Portability **creat** is available on UNIX systems.

See also **chmod, chsize, close, _creat, creatnew, creattemp, dup, dup2,** *_fmode* (variable), **fopen, open, sopen, write**

creatnew

Function Creates a new file.

Syntax

```
#include <dos.h>
int creatnew(const char *path, int attrib);
```

Prototype in io.h

Remarks **creatnew** is identical to **_creat**, with the exception that, if the file exists, the **creatnew** call returns an error and leaves the file untouched.

The *mode* argument to **creatnew** can be one of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return value Upon successful completion, **creat** returns the new file handle, a nonnegative integer; otherwise, it returns –1.

In the event of error, *errno* is set to one of the following:

EEXIST	File already exists
ENOENT	Path or file name not found
EMFILE	Too many open files
EACCES	Permission denied

Portability **creatnew** is unique to DOS 3.0 and will not work on earlier DOS versions.

See also **close, _creat, creat, creattemp, dup,** *_fmode* (variable), **open**

creattemp

Function Creates a unique file in the directory associated with the path name.

Syntax

```
#include <dos.h>
int creattemp(char *path, int attrib);
```

Prototype in io.h

Remarks A file created with **creattemp** is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

path is a path name ending with a backslash (\). A unique file name is selected in the directory given by *path*. The newly created file name is stored in the *path* string supplied. *path* should be long enough to hold the resulting file name. The file is not automatically deleted when the program terminates.

creattemp accepts *amode*, a DOS attribute word. Any attribute bits can be set in this call. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing

The *amode* argument to **creattemp** can be one of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return value Upon successful completion, the new file handle, a non-negative integer, is returned; otherwise, a –1 is returned.

In the event of error, *errno* is set to one of the following:

ENOENT	Path or file name not found
EMFILE	Too many open files
EACCES	Permission denied

Portability **creattemp** is unique to DOS 3.0 and will not work on earlier versions.

See also **close**, **_creat**, **creat**, **creatnew**, **dup**, *_fmode* (variable), **open**

cscanf

Function Scans and formats input from the console.

Syntax int cscanf(char **format*[, *address*, ...]);

Prototype in conio.h

Remarks **cscanf** scans a series of input fields, one character at a time, reading directly from the console. Then each field is formatted according to a format specification passed to **cscanf** in the format string pointed to by *format*. Finally, **cscanf** stores the formatted input at an address passed to it as an argument following *format*, and echoes the input directly to the screen. There must be the same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

cscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value **cscanf** returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If **cscanf** attempts to read at end-of-file , the return value is EOF.

Portability **cscanf** is available on UNIX systems and is defined in Kernighan and Ritchie.

See also **fscanf, getche, scanf, sscanf**

ctime

Function Converts date and time to a string.

Syntax

```
#include <time.h>
char *ctime(const time_t *time);
```

Prototype in time.h

Remarks **ctime** converts a time value pointed to by *time* (the value returned by the function **time**) into a 26-character string in the following form, terminating with a newline character and a null character:

```
Mon Nov 21 11:31:54 1983\n\0
```

All the fields have constant width.

The global long variable *timezone* should be set to the difference in seconds between GMT and local standard time (in PST, *timezone* is 8 × 60 × 60). The global variable *daylight* is nonzero *if and only if* the standard USA Daylight Savings time conversion should be applied.

Return value **ctime** returns a pointer to the character string containing the date and time. The return value points to static data that is overwritten with each call to **ctime**.

Portability **ctime** is available on UNIX systems and is compatible with ANSI C.

See also **asctime**, *daylight* (variable), **difftime, ftime, getdate, gmtime, localtime, settime, time**, *timezone* (variable), **tzset**

Example See **asctime**

ctrlbrk

Function Sets control-break handler.

Syntax

```
void ctrlbrk(int (*handler)(void));
```

Prototype in dos.h

Remarks

ctrlbrk sets a new control-break handler function pointed to by *handler*. The interrupt vector 0x23 is modified to call the named function.

ctrlbrk establishes a DOS interrupt handler that calls the named function; the named function is not called directly.

The handler function can perform any number of operations and system calls. The handler does not have to return; it can use **longjmp** to return to an arbitrary point in the program. The handler function returns 0 to abort the current program; any other value will cause the program to resume execution.

Return value

ctrlbrk returns nothing.

Portability

ctrlbrk is unique to DOS.

See also

getcbrk, signal

Example

```
#include <stdio.h>
#include <dos.h>

#define ABORT 0
int c_break(void)
{
   printf("Control-Break hit.
           Program aborting ...\n");
   return(ABORT);
}

main()
{
   ctrlbrk(c_break);
   /* infinite loop */
   for (;;)
   {
      printf("Looping ...\n");
   }
}
```

Program output

```
Looping ...
Looping ...
Looping ...
^C
Control-Break hit.  Program aborting ...
```

delay

Function Suspends execution for an interval (milliseconds).

Syntax void delay(unsigned *milliseconds*);

Prototype in dos.h

Remarks With a call to **delay**, the current program is suspended from execution for the number of milliseconds specified by the argument *milliseconds*. The exact time may vary somewhat in different operating environments.

Return value None.

Portability This function works only with IBM PCs and compatibles.

See also **nosound, sleep, sound**

Example

```
/* Emits a 440 Hz tone for 500 milliseconds */
#include <dos.h>

main()
{
   sound(440);
   delay(500);
   nosound();
}
```

delline

Function Deletes line in text window.

Syntax void delline(void);

Prototype in conio.h

Remarks **delline** deletes the line containing the cursor and moves all lines below it one line up. **delline** operates within the currently active text window.

Return value None.

Portability This function works only with IBM PCs and compatibles.

See also **clreol, clrscr, insline, window**

detectgraph

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

Syntax

```
#include <graphics.h>
void far detectgraph(int far *graphdriver
                     int far *graphmode);
```

Prototype in graphics.h

Remarks **detectgraph** detects your system's graphics adapter and chooses the mode that provides the highest resolution for that adapter. If no graphics hardware was detected, **graphdriver* is set to –2, and **graphresult** will also return –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, defined in graphics.h and listed in the following table.

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 to the highest resolution available for the detected driver). You can give **graphmode* a value using a constant of the *graphics_modes* enumeration type, defined in graphics.h and listed in the following table.

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **graphresult, initgraph**

difftime

Function Computes the difference between two times.

Syntax

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Prototype in time.h

Remarks **difftime** calculates the elapsed time, in seconds, from *time1* to *time2*.

The global long variable *timezone* contains the difference in seconds between GMT and local standard time (in PST, *timezone* is 8 × 60 × 60. The global variable *daylight* is nonzero *only if* the standard U.S. Daylight Savings Time conversion should be applied.

Return value **difftime** returns the result of its calculation as a **double**.

Portability **difftime** is available on UNIX systems and is compatible with ANSI C.

See also **asctime, ctime,** *daylight* (variable), **time,** *timezone* (variable)

disable

Function Disables interrupts.

Syntax

```
#include <dos.h>
void disable(void);
```

Prototype in dos.h

Remarks

disable is designed to provide a programmer with flexible hardware interrupt control.

The **disable** macro disables interrupts. Only the NMI interrupt will still be allowed from any external device.

Return value

None.

Portability

This macro is unique to the 8086 architecture.

See also

enable, getvect

---

div

Function

Divides two integers, returning quotient and remainder.

Syntax

#include <stdlib.h>
div_t div(int *numer*, int *denom*);

Prototype in

stdlib.h

Remarks

div divides two integers and returns both the quotient and the remainder as a *div_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *div_t* type is a structure of integers defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
   int  quot;                                  /* quotient */
   int  rem;                                  /* remainder */
} div_t;
```

Return value

div returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

Portability

div is compatible with ANSI C.

See also

ldiv

Example

```
#include <stdlib.h>
div_t x;

main()
{
   x = div(10,3);
   printf("10 div 3 = %d remainder %d\n", x.quot, x.rem);
}
```

Program output

```
10 div 3 = 3 remainder 1
```

dosexterr

Function Gets extended DOS error information.

Syntax
#include <dos.h>
int dosexterr(struct DOSERROR **eblkp*);

Prototype in dos.h

Remarks This function fills in the **DOSERROR** structure pointed to by *eblkp* with extended error information after a DOS call has failed. The structure is defined as follows:

```
struct  DOSERROR {
   int exterror;                      /* extended error */
   char class;                           /* error class */
   char action;                               /* action */
   char locus;                           /* error locus */
};
```

The values in this structure are obtained via DOS call 0x59. An *exterror* value of 0 indicates that the prior DOS call did not result in an error.

Return value **dosexterr** returns the value *exterror*.

Portability **dosexterr** is unique to DOS 3.0 and will not work on earlier releases.

dostounix

Function Converts date and time to UNIX time format.

Syntax
#include <dos.h>
long dostounix(struct date **d*, struct time **t*);

Prototype in dos.h

Remarks **dostounix** converts a date and time as returned from **getdate** and **gettime** into UNIX time format. *d* points to

a **date** structure, and *t* points to a **time** structure containing valid DOS date and time information.

Return value UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT).

Portability **dostounix** is unique to DOS.

See also **unixtodos**

---

drawpoly

Function Draws the outline of a polygon.

Syntax #include <graphics.h>
void far drawpoly(int *numpoints*, int far **polypoints*);

Prototype in graphics.h

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.

Note: In order 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 If an error occurs while the polygon is being drawn, **graphresult** will return a value of –6.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **fillpoly, floodfill, graphresult, setwritemode**

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* Will request autodetection */
   int graphdriver = DETECT, graphmode;
   int triangle[ ] = {50,100, 100,100, 150,150, 50,100};
   int rhombus[ ]  = {50,10, 90,50, 50,90, 10,50};
```

```
   /* Initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* Draw a triangle */
   drawpoly(sizeof(triangle)/(2*sizeof(int)), triangle);

   /* Draw and fill a rhombus */
   fillpoly(sizeof(rhombus)/(2*sizeof(int)), rhombus);
   getche();
   closegraph();
}
```

dup

Function Duplicates a file handle.

Syntax int dup(int *handle*);

Prototype in io.h

Remarks **dup** creates a new file handle that has the following in common with the original file handle:

- same open file or device
- same file pointer (that is, changing the file pointer of one changes the other)
- same access mode (read, write, read/write)

handle is a file handle obtained from a **_creat**, **creat**, **_open**, **open**, **dup**, or **dup2** call.

Return value Upon successful completion, **dup** returns the new file handle, a nonnegative integer; otherwise, **dup** returns –1.

In the event of error, *errno* is set to one of the following:

EMFILE	Too many open files
EBADF	Bad file number

Portability **dup** is available on all UNIX systems.

See also **_close, close, _creat, creat, creatnew, creattemp, dup2, fopen, _open, open**

dup2

Function Duplicates a file handle (*oldhandle*) onto an existing file handle (*newhandle*).

Syntax int dup2(int *oldhandle*, int *newhandle*);

Prototype in io.h

Remarks **dup2** creates a new file handle that has the following in common with the original file handle:

- same open file or device
- same file pointer (that is, changing the file pointer of one changes the other)
- same access mode (read, write, read/write)

dup2 creates a new handle with the value of *newhandle*. If the file associated with *newhandle* is open when **dup2** is called, the file is closed.

newhandle and *oldhandle* are file handles obtained from a **creat**, **open**, **dup**, or **dup2** call.

Return value **dup2** returns 0 on successful completion, –1 otherwise.

In the event of error, *errno* is set to one of the following:

EMFILE	Too many open files
EBADF	Bad file number

Portability **dup2** is available on some UNIX systems, but not System III.

See also **_close, close, _creat, creat, creatnew, creattemp, dup, fopen, _open, open**

ecvt

Function Converts a floating-point number to a string.

Syntax char *ecvt(double *value*, int *ndig*, int **dec*, int **sign*);

Prototype in stdlib.h

Remarks

ecvt converts *value* to a null-terminated string of *ndig* digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative, the word pointed to by *sign* is nonzero; otherwise, it is 0. The low-order digit is rounded.

Return value

The return value of **ecvt** points to static data for the string of digits whose content is overwritten by each call to **ecvt**.

Portability

ecvt is available on UNIX.

See also

atof, atoi, atol, fcvt, gcvt, printf

ellipse

Function

Draws an elliptical arc.

Syntax

```
#include <graphics.h>
void far ellipse(int x, int y, int stangle,
                 int endangle, int xradius, int yradius);
```

Prototype in

graphics.h

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** will draw 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.

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

Return value

None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **arc**, **circle**, **fillellipse**, **getaspectratio**, **sector**, **setaspectratio**

Examples See **arc**

__emit__

Function Inserts literal values directly into code.

Syntax void __emit__(argument, ...);

Prototype in dos.h

Description **__emit__** is an inline function that allows you to insert literal values directly into object code as it is compiling. It is used to generate machine language instructions without using inline assembly language or an assembler. It can be used in the integrated development environment, which inline assembly code cannot.

Generally the arguments of an **__emit__** call are single-byte machine instructions. However, because of the capabilities of this function, more complex instructions, complete with references to C variables, can be constructed.

Warning! This function should only be used by programmers who feel comfortable with the machine language of the 80x86 processor family. You can use this function to place arbitrary bytes in the instruction code of a function; if any of these bytes are incorrect, the program will misbehave and may easily crash your machine. Turbo C does not attempt to analyze your calls for correctness in any way. If you encode instructions that change machine registers or memory, Turbo C will not be aware of it and may not properly preserve registers, as it would in many cases with inline assembly language (for example, it recognizes the usage of SI and DI registers in inline instructions). You are completely on your own with this function.

You must pass at least one argument to **__emit__**; any number may be given. The arguments to this function are not treated like any other function call arguments in the language. An argument passed to **__emit__** will not be converted in any way.

There are special restrictions on the form of the arguments to **__emit__**. They must be in the form of expressions that can be used to initialize a static object. This means that integer and floating point constants and the addresses of static objects may be used. The values of such expressions are written to the object code at the point of the call, exactly as if they were being used to initialize data. The address of an auto or parameter variable, plus or minus a constant offset, may also be used. For these arguments, the offset of the variable from BP is stored.

The number of bytes placed in the object code is determined from the type of the argument, except in the following cases:

- If a signed integer constant (i.e. 0x90) appears that fits within the range of 0 to 255, it is treated as if it were a character.
- If the address of an auto or parameter variable is used, a byte is written if the offset of the variable from BP is between –128 and 127; otherwise a word is written.

Simple bytes written as follows:

```
__emit__(0x90);
```

If you want a word written, but the value you are passing is under 255, simply cast it to unsigned, as follows:

```
__emit__(0xB8, (unsigned)17);
```

or

```
__emit__(0xB8, 17u);
```

Two- or four-byte address values can be forced by casting an address to **void near** * or **void far** * respectively.

Return value None.

Portability **__emit__** is unique to Intel 80x86 processors.

enable

Function Enables hardware interrupts.

Syntax
```
#include <dos.h>
void enable(void);
```

Prototype in dos.h

Remarks **enable** is designed to provide a programmer with flexible hardware interrupt control.

The **enable** macro enables interrupts, allowing any device interrupts to occur.

Return value None.

Portability **enable** is unique to the 80 × 86 architecture.

See also **disable, getvect**

eof

Function Checks for end-of-file.

Syntax int eof(int *handle*);

Prototype in io.h

Remarks **eof** determines whether the file associated with *handle* has reached end-of-file.

Return value If the current position is end-of-file, **eof** returns the value 1; otherwise, it returns 0. A return value of –1 indicates an error; *errno* is set to

EBADF	Bad file number

See also **clearerr, feof, ferror, perror**

exec...

Function Loads and runs other programs.

Syntax

int execl(char *path*, char *arg0*,
 arg1, ..., *argn*, NULL);
int execle(char *path*, char *arg0*,
 arg1, ..., *argn*, NULL, char ***env*);
int execlp(char *path*, char *arg0*,
 arg1, ..., *argn*, NULL);
int execlpe(char *path*, char *arg0*,
 arg1, ..., *argn*, NULL, char ***env*);

int execv(char *path*, char *argv[]*);
int execve(char *path*, char *argv[]*, char ***env*);
int execvp(char *path*, char *argv[]*);
int execvpe(char *path*, char *argv[]*, char ***env*);

Prototype in process.h

Remarks The functions in the **exec...** family load and run (execute) other programs, known as *child processes*. When an **exec...** call is successful, the child process overlays the *parent process*. There must be sufficient memory available for loading and executing the child process.

path is the file name of the called child process. The **exec...** functions search for *path* using the standard DOS search algorithm:

- If no explicit extension is given, the functions will search for the file as given. If the file is not found, they will add .COM and search again. If that search is not successful, they will add .EXE and search one last time.
- If an explicit extension or a period is given, the functions will search for the file exactly as given.
- If the file name has a period but no extension, the functions will look for a file with no extension.

The suffixes *l*, *v*, *p*, and *e* added to the **exec...** "family name" specify that the named function will operate with certain capabilities.

p The function will search for the file in those directories specified by the DOS PATH environment variable (without the *p* suffix, the function searches only the current working directory). If the *path* parameter does not contain an explicit directory, the function will search first the current directory, then the directories set with the DOS PATH environment variable.

l The argument pointers (*arg0, arg1, ..., argn*) are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.

v The argument pointers (*argv[0]* ..., *arg[n]*) are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.

e The argument *env* may be passed to the child process, allowing you to alter the environment for the child process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the **exec...** family *must* have one of the two argument-specifying suffixes (either *l* or *v*). The *path search* and *environment inheritance* suffixes (*p* and *e*) are optional.

For example:

- **execl** is an **exec...** function that takes separate arguments, searches only the root or current directory for the child, and passes on the parent's environment to the child.
- **execvpe** is an **exec...** function that takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *env* argument for altering the child's environment.

The **exec...** functions must pass at least one argument to the child process (*arg0* or *argv[0]*); this argument is, by convention, a copy of *path*. (Using a different value for this 0th argument won't produce an error.)

Under DOS 3.x, *path* is available for the child process; under earlier versions, the child process cannot use the passed value of the 0th argument (*arg0* or *argv[0]*).

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1, ..., argn* point to character strings that form the new list of arguments. A mandatory NULL following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *env*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

envvar = value

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *env* is NULL. When *env* is NULL, the child inherits the parents' environment settings.

The combined length of *arg0* + *arg1* + ... + *argn* (or of *argv[0]* + *argv[1]* + ... + *argn[n]*), including space characters that separate the arguments, must be less than 128 bytes. Null terminators are not counted.

When an **exec...** function call is made, any open files remain open in the child process.

Return value

If successful, the **exec...** functions return no value. On error, the **exec...** functions return –1, and *errno* is set to one of the following:

E2BIG	Arg list too long
EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough core

Portability

exec... is unique to DOS.

See also

abort, atexit, _exit, exit, _fpreset, searchpath, spawn..., system

Example

```
#include <stdio.h>
#include <process.h>
```

```
main()
{
   int stat;

   printf("About to exec child with arg1 arg2 ...\n");
   stat = execl("CHILD.EXE", "CHILD.EXE", "arg1", "arg2",
                NULL);

   /* execl will return only if it cannot run CHILD */
   printf("execl error = %d\n", stat);
   exit(1);
}

/* CHILD.C */
#include <stdio.h>

main(int argc, char *argv[])
{
   int i;

   printf("Child running ...\n");
   /* print out its arguments */
   for (i=0; i<argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);
}
```

Program output

```
About to exec child with arg1 arg2 ...
Child running ...
argv[0]: CHILD.EXE
argv[1]: arg1
argv[2]: arg2
```

_exit

Function Terminates program.

Syntax void _exit(int *status*);

Prototype in process.h, stdlib.h

Remarks **_exit** terminates execution without closing any files, flushing any output, or calling any exit functions.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

Return value None.

Portability **_exit** is available on UNIX systems.

See also **abort, atexit, exec..., exit, spawn...**

exit

Function Terminates program.

Syntax void exit(int *status*);

Prototype in process.h, stdlib.h

Remarks **exit** terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with **atexit**) are called.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

Return value None.

Portability **exit** is available on UNIX systems and is compatible with ANSI C.

See also **abort, atexit, exec..., _exit, keep, signal, spawn...**

exp

Function Calculates the exponential *e* to the xth power.

Syntax

```
#include <math.h>
double exp(double x);
```

Prototype in math.h

Remarks **exp** calculates the exponential function e^x.

Return value **exp** returns e^x.

Sometimes the arguments passed to **exp** produce results that overflow or are incalculable. When the correct value

overflows, **exp** returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to

ERANGE	Result out of range

On underflow, **exp** returns 0.0, and *errno* is not changed.

Error-handling for **exp** can be modified through the function **matherr**.

Portability **exp** is available on UNIX systems and is compatible with ANSI C.

See also **frexp, ldexp, log, log10, matherr, pow, pow10, sqrt**

fabs

Function Returns the absolute value of a floating-point number.

Syntax

```
#include <math.h>
double fabs(double x);
```

Prototype in math.h

Remarks **fabs** calculates the absolute value of *x*, a **double**.

Return value **fabs** returns the absolute value of *x*. There is no return on error.

Portability **fabs** is available on UNIX systems and is compatible with ANSI C.

See also **abs, cabs, labs**

farcalloc

Function Allocates memory from the far heap.

Syntax

```
void far *farcalloc(unsigned long nunits,
    unsigned long unitsz);
```

Prototype in alloc.h

Remarks **farcalloc** allocates memory from the far heap for an array containing *nunits* elements, each *unitsz* bytes long.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

In the compact, large, and huge memory models, **farcalloc** is similar, though not identical, to **calloc**. It takes **unsigned long** parameters, while **calloc** takes **unsigned** parameters.

A tiny model program cannot make use of **farcalloc** if it is to be converted to a .COM file.

Return value — **farcalloc** returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

Portability — **farcalloc** is unique to DOS.

See also — **calloc, farcoreleft, farfree, malloc**

---

farcoreleft

Function — Returns measure of unused memory in far heap.

Syntax — unsigned long farcoreleft(void);

Prototype in — alloc.h

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** if it is to be converted to a .COM file.

Return value — **farcoreleft** returns the total amount of space left in the far heap, between the highest allocated block and the end of memory.

Portability — **farcoreleft** is unique to DOS.

See also — **coreleft, farcalloc, farmalloc**

Example — See **farmalloc**

farfree

Function Frees a block from far heap.

Syntax void farfree(void far * *block*);

Prototype in alloc.h

Remarks **farfree** releases a block of memory previously allocated from the far heap.

A tiny model program cannot make use of **farfree** if it is to be converted to a .COM file.

In the small and medium memory models, blocks allocated by **farmalloc** can not be freed via normal **free**, and blocks allocated via **malloc** cannot be freed via **farfree**. In these models, the two heaps are completely distinct.

Return value None.

Portability **farfree** is unique to DOS.

See also **farcalloc, farmalloc**

Example see **farmalloc**

farmalloc

Function Allocates from far heap.

Syntax void far *farmalloc(unsigned long *nbytes*);

Prototype in alloc.h

Remarks **farmalloc** allocates a block of memory *nbytes* bytes long from the far heap.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

In the compact, large, and huge memory models, **farmalloc** is similar, though not identical, to **malloc**. It

takes **unsigned long** parameters, while **malloc** takes **unsigned** parameters.

A tiny model program cannot make use of **farmalloc** if it is to be converted to a .COM file.

Return value

farmalloc returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

Portability

farmalloc is unique to DOS.

See also

farcalloc, farcoreleft, farfree, farrealloc, malloc

Example

```
/* Far Memory Management
   farcoreleft - gets the amount of core memory left
   farmalloc   - allocates space on the far heap
   farrealloc  - adjusts allocated block in far heap
   farfree     - frees far heap */

#include <stdio.h>
#include <alloc.h>

main()
{
   char far * block;
   long size = 65000;

   /* Find out what's out there */
   printf("%lu bytes free\n", farcoreleft());

   /* Get a piece of it */
   block = farmalloc(size);
   if (block == NULL)
   {
      printf("failed to allocate\n");
      exit(1);
   }
   printf("%lu bytes allocated, ",size);
   printf("%lu bytes free\n", farcoreleft());

   /* Shrink the block */
   size /= 2;
   block = farrealloc(block, size);
   printf("block now reallocated to %lu bytes, ",size);
   printf("%lu bytes free\n", farcoreleft());

   /* Let it go entirely */
   printf("Free the block\n");
   farfree(block);
   printf("block now freed, ");
   printf("%lu bytes free\n", farcoreleft());
} /* End of main */
```

Program output

```
359616 bytes free
65000 bytes allocated, 294608 bytes free
block now reallocated to 32500 bytes, 262100 bytes free
Free the block
Block now freed, 359616 bytes free
```

farrealloc

Function Adjusts allocated block in far heap.

Syntax void far *farrealloc(void far **oldblock*,
unsigned long *nbytes*);

Prototype in alloc.h

Remarks **farrealloc** adjusts the size of the allocated block to *nbytes*, copying the contents to a new location, if necessary.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

A tiny model program cannot make use of **farrealloc** if it is to be converted to a .COM file.

Return value **farrealloc** returns the address of the reallocated block, which may be different than the address of the original block. If the block cannot be reallocated, **farrealloc** returns NULL.

Portability **farrealloc** is unique to DOS.

See also **farmalloc, realloc**

Example See **farmalloc**

fclose

Function	Closes a stream.
Syntax	#include <stdio.h> int fclose(FILE **stream*);
Prototype in	stdio.h
Remarks	**fclose** closes the named stream. Generally, all buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with **setbuf** or **setvbuf** are not automatically freed.
Return value	**fclose** returns 0 on success. It returns EOF if any errors were detected.
Portability	**fclose** is available on UNIX systems and is compatible with ANSI C.
See also	**close, fcloseall, fdopen, fflush, flushall, fopen, freopen**
Example	See **fopen**

fcloseall

Function	Closes open streams.
Syntax	int fcloseall(void);
Prototype in	stdio.h
Remarks	**fcloseall** closes all open streams except *stdin*, *stdout*, *stdprn*, *stderr*, and *stdaux*.
Return value	**fcloseall** returns the total number of streams it closed. It returns EOF if any errors were detected.
Portability	**fcloseall** is available on UNIX systems.
See also	**fclose, fdopen, flushall, fopen, freopen**

fcvt

Function Converts a floating-point number to a string.

Syntax

```
#include <stdlib.h>
char *fcvt(double value, int ndig,
        int *dec, int *sign);
```

Prototype in stdlib.h

Remarks **fcvt** converts *value* to a null-terminated string of *ndig* digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative, the word pointed to by *sign* is nonzero; otherwise, it is 0.

The correct digit has been rounded for the number of digits specified by *ndig*.

Return value The return value of **fcvt** points to static data whose content is overwritten by each call to **fcvt**.

Portability **fcvt** is available on UNIX.

See also **atof, atoi, atol, ecvt, gcvt**

fdopen

Function Associates a stream with a file handle.

Syntax

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

Prototype in stdio.h

Remarks **fdopen** associates a stream with a file handle obtained from **creat, dup, dup2,** or **open**. The type of stream must match the mode of the open *handle*.

The *type* string used in a call to **fdopen** is one of the following values:

r Open for reading only.

w Create for writing.

a Append; open for writing at end-of-file or create for writing if the file does not exist.

r+ Open an existing file for update (reading and writing).

w+ Create a new file for update.

a+ Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, append a *t* to the value of the *type* string (*rt*, *w+t*, etc.); similarly, to specify binary mode, append a *b* to the *type* string (*wb*, *a+b*, etc.).

If a *t* or *b* is not given in the *type* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files will be opened in binary mode. If *_fmode* is set to O_TEXT, they will be opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be directly followed by input without an intervening **fseek** or **rewind**, and input cannot be directly followed by output without an intervening **fseek**, **rewind**, or an input that encounters end-of-file.

Return value On successful completion, **fdopen** returns a pointer to the newly opened stream. In the event of error, it returns NULL.

Portability **fdopen** is available on UNIX systems.

See also **fclose, fopen, freopen, open**

Example

```
#include <stdio.h>
#include <fcntl.h>
/* Needed to define the mode used in open */

main()
{
   int handle;
   FILE *stream;

   /* Open a file */
```

```
   handle = open("MYFILE.TXT", O_CREAT);

   /* Now turn it into a stream */
   stream = fdopen(handle, "w");
   if (stream == NULL)
      printf("fdopen failed\n");
   else
   {
      fprintf(stream, "Hello, world\n");
      fclose(stream);
   }
}
```

feof

Function Detects end-of-file on a stream.

Syntax #include <stdio.h>
int feof(FILE **stream*);

Prototype in stdio.h

Remarks **feof** is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set, read operations on the file return the indicator until **rewind** is called or the file is closed.

The end-of-file indicator is reset with each input operation.

Return value **feof** returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached.

Portability **feof** is available on UNIX systems and is compatible with ANSI C.

See also **clearerr, eof, ferror, perror**

ferror

Function Detects errors on stream.

Syntax
```
#include <stdio.h>
int ferror(FILE *stream);
```

Prototype in stdio.h

Remarks **ferror** is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set, it remains set until **clearerr** or **rewind** is called, or until the stream is closed.

Return value **ferror** returns nonzero if an error was detected on the named stream.

Portability **ferror** is available on UNIX systems and is compatible with ANSI C.

See also **clearerr, eof, feof, fopen, gets, perror**

fflush

Function Flushes a stream.

Syntax
```
#include <stdio.h>
int fflush(FILE *stream);
```

Prototype in stdio.h

Remarks If the given stream is open for output, **fflush** writes the buffered output for *stream* to the associated file.

If *stream* is open for input, **fflush** clears the buffer contents.

The stream remains open after **fflush** has executed. **fflush** has no effect on an unbuffered stream.

Return value **fflush** returns 0 on success. It returns EOF if any errors were detected.

Portability **fflush** is available on UNIX systems and is compatible with ANSI C.

See also **fclose, flushall, setbuf, setvbuf**

fgetc

Function	Gets character from stream.
Syntax	#include <stdio.h> int fgetc(FILE **stream*);
Prototype in	stdio.h
Remarks	**fgetc** returns the next character on the named input stream.
Return value	On success, **fgetc** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.
Portability	**fgetc** is available on UNIX systems and is compatible with ANSI C.
See also	**fgetchar, fputc, getc, getch, getchar, getche, ungetc, ungetch**

fgetchar

Function	Gets character from *stdin.*
Syntax	int fgetchar(void);
Prototype in	stdio.h
Remarks	**fgetchar** returns the next character from *stdin*. It is defined as *fgetc(stdin).*
Return value	On success, **fgetchar** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF. Because EOF is a legitimate value for **fgetchar** to return, **feof** and **ferror** should be used to detect end-of-file or error.
Portability	**fgetchar** is available on UNIX systems.
See also	**fgetc, fputchar, getchar**

fgetpos

Function Gets the current file pointer.

Syntax #include <stdio.h>
int fgetpos(FILE **stream*, fpos_t **pos*);

Prototype in stdio.h

Remarks **fgetpos** stores the position of the file pointer associated with the given stream in the location pointed to by *pos*.

The type *fpos_t* is defined in stdio.h as

```
typedef long fpos_t;
```

Return value On success, **fgetpos** returns 0. On failure, it returns a nonzero value and sets *errno* to EBADF or EINVAL.

Portability **fgetpos** is compatible with ANSI C.

See also **fseek, fsetpos, ftell, tell**

fgets

Function Gets a string from a stream.

Syntax #include <stdio.h>
char *fgets(char **s*, int *n*, FILE **stream*);

Prototype in stdio.h

Remarks **fgets** reads characters from *stream* into the string *s*. The function stops reading when it reads either *n* – 1 characters or a newline character, whichever comes first. **fgets** does not place the newline character in the string. The last character read into *s* is followed by a null character.

Return value On success, **fgets** returns the string pointed to by *s*; it returns NULL on end-of-file or error.

Portability **fgets** is available on UNIX systems and is compatible with ANSI C. It is also defined in Kernighan and Ritchie.

See also **cgets, fputs, gets**

filelength

Function Gets file size in bytes.

Syntax

```
#include <io.h>
long filelength(int handle);
```

Prototype in io.h

Remarks **filelength** returns the length (in bytes) of the file associated with *handle*.

Return value On success, **filelength** returns a **long** value, the file length in bytes. On error, it returns –1, and *errno* is set to

EBADF	Bad file number

See also **fopen, lseek, open**

fileno

Function Gets file handle.

Syntax

```
#include <stdio.h>
int fileno(FILE *stream);
```

Prototype in stdio.h

Remarks **fileno** is a macro that returns the file handle for the given stream. If *stream* has more than one handle, **fileno** returns the handle assigned to the stream when it was first opened.

Return value **fileno** returns the integer file handle associated with *stream*.

Portability **fileno** is available on UNIX systems.

See also **fdopen, fopen, freopen**

fillellipse

Function Draws and fills an ellipse.

Syntax

```
#include <graphics.h>
void far fillellipse(int x, int y, int xradius,
                     int yradius);
```

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **arc, circle, ellipse, getaspectratio, pieslice, setaspectratio**

fillpoly

Function Draws and fills a polygon.

Syntax

```
#include <graphics.h>
void far fillpoly(int numpoints, int far *polypoints);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **drawpoly, floodfill, graphresult, setfillstyle**

findfirst

Function Searches a disk directory.

Syntax

```
#include <dir.h>
#include <dos.h>
int findfirst(const char *pathname,
              struct ffblk *ffblk, int attrib);
```

Prototype in dir.h

Remarks **findfirst** begins a search of a disk directory by using the DOS system call 0x4E.

pathname is a string with an optional drive specifier, path, and file name of the file to be found. The file name portion can contain wildcard match characters (such as ? or *). If a matching file is found, the **ffblk** structure is filled with the file-directory information.

The format of the structure **ffblk** is as follows:

```
struct ffblk {
   char ff_reserved[21];            /* reserved by DOS */
   char ff_attrib;                  /* attribute found */
   int ff_ftime;                          /* file time */
   int ff_fdate;                          /* file date */
   long ff_fsize;                         /* file size */
   char ff_name[13];               /* found file name */
};
```

attrib is a DOS file-attribute byte used in selecting eligible files for the search. *attrib* can be one of the following constants defined in dos.h:

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

For more detailed information about these attributes, refer to the *DOS Programmer's Reference Manual*.

Note that **findfirst** sets the DOS disk-transfer address (DTA) to the address of the **ffblk**.

If you need this DTA value, you should save it and restore it (using **getdta** and **setdta**) after each call to **findfirst**.

Return value

findfirst returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, –1 is returned, and the global variable *errno* is set to one of the following:

ENOENT	Path or file name not found
ENMFILE	No more files

Portability

findfirst is unique to DOS.

See also

findnext

Example

```
#include <stdio.h>
#include <dir.h>

main()
{
   struct ffblk ffblk;
   int done;
   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,0);
   while (!done)
   {
      printf("  %s\n", ffblk.ff_name);
      done = findnext(&ffblk);
   }
}
```

Program output

```
Directory listing of *.*
FINDFRST.C
FINDFRST.OBJ
FINDFRST.MAP
FINDFRST.EXE
```

findnext

Function Continues **findfirst** search.

Syntax

```
#include <dir.h>
int findnext(struct ffblk *ffblk);
```

Prototype in dir.h

Remarks **findnext** is used to fetch subsequent files that match the *pathname* given in **findfirst**. *ffblk* is the same block filled in by the **findfirst** call. This block contains necessary information for continuing the search. One file name for each call to **findnext** will be returned until no more files are found in the directory matching the *pathname*.

Note that **findnext** sets the DOS disk-transfer address (DTA) to the address of **ffblk**.

If you need this DTA value, you should save it and restore it (using **getdta** and **setdta**) after each call to **findnext**.

Return value **findnext** returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, –1 is returned, and the global variable *errno* is set to one of the following:

ENOENT	Path or file name not found
ENMFILE	No more files

Portability **findnext** is unique to DOS.

See also **findfirst**

Example See **findfirst**

floodfill

Function Flood-fills a bounded region.

Syntax

```
#include <graphics.h>
void far floodfill(int x, int y, int border);
```

Prototype in graphics.h

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** will return a value of –7.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

drawpoly, fillpoly, graphresult, setcolor, setfillstyle

Example

```
#include <graphics.h>

main()
{
   /* Will request autodetection */
   int graphdriver = DETECT, graphmode;

   /* Initialize graphics */
   initgraph(&graphdriver, &graphmode, "");

   /* Draw a bar, then flood-fill the side and top */
   setcolor(WHITE);
   setfillstyle(HATCH_FILL, LIGHTMAGENTA);
   bar3d(10, 10, 100, 100, 10, 1);
   setfillstyle(SOLID_FILL, LIGHTGREEN);
   /* Fill the side */
   floodfill(102, 50, WHITE);
   /* Fill the top */
   floodfill(50, 8, WHITE);

   closegraph();
}
```

floor

Function Rounds down.

Syntax

```
#include <math.h>
double floor(double x);
```

Prototype in math.h

Remarks **floor** finds the largest integer not greater than *x*.

Return value **floor** returns the integer found (as a **double**).

Portability **floor** is available on UNIX systems and is compatible with ANSI C.

See also **ceil, fmod**

flushall

Function Flushes all streams.

Syntax

```
int flushall(void);
```

Prototype in stdio.h

Remarks **flushall** clears all buffers associated with open input streams, and writes all buffers associated with open output streams to their respective files. Any read operation following **flushall** reads new data into the buffers from the input files.

Streams stay open after **flushall** has executed.

Return value **flushall** returns an integer, the number of open input and output streams.

Portability **flushall** is available on UNIX systems.

See also **fclose, fcloseall, fflush**

fmod

Function Calculates x modulo y, the remainder of x/y.

Syntax

```
#include <math.h>
double fmod(double x, double y);
```

Prototype in math.h

Remarks **fmod** calculates x modulo y (the remainder f where $x = iy + f$ for some integer i and $0 \le f < y$).

Return value **fmod** returns the remainder f, where $x = iy + f$ (as described).

Portability **fmod** is compatible with ANSI C.

See also **ciel, floor, modf**

fnmerge

Function Builds a path from component parts.

Syntax

```
#include <dir.h>
void fnmerge(char *path, const char *drive,
             const char *dir, const char *name,
             const char *ext);
```

Prototype in dir.h

Remarks **fnmerge** makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

drive = `X:`
dir = `\DIR\SUBDIR\`
name = `NAME`
ext = `.EXT`

fnmerge assumes there is enough space in *path* for the constructed path name. The maximum constructed length is MAXPATH. MAXPATH is defined in dir.h.

fnmerge and **fnsplit** are invertible; if you split a given *path* with **fnsplit**, then merge the resultant components with **fnmerge**, you end up with *path*.

Return value None.

Portability **fnmerge** is available on DOS systems only.

See also **fnsplit**

Example

```
#include <stdio.h>
#include <dir.h>

char drive[MAXDRIVE];
char dir[MAXDIR];
char file[MAXFILE];
char ext[MAXEXT];

main()
{
   char s[MAXPATH], t[MAXPATH];
   int flag;
   for (;;)
   {
      /* Print input prompt while */
      printf("> ");
      /* There is more input */
      if (!gets(s)) break;
      if (!gets[0]) break;
      flag = fnsplit(s,drive,dir,file,ext);

      /* Print the components */
      printf("drive: %s, dir: %s, file: %s, ext: %s, ",
             drive, dir, file, ext);
      printf("flags: ");
      if (flag & DRIVE)
         printf(":");
      if (flag & DIRECTORY)
         printf("d");
      if (flag & FILENAME)
         printf("f");
      if (flag & EXTENSION)
         printf("e");
         printf("\n");

      /* Glue the parts back together and
         compare to original */
      fnmerge(t,drive,dir,file,ext);
      /* Shouldn't happen! */
      if (strcmp(t,s) != 0)
         printf(" --> strings are different!");
```

```
    }
}
```

Program output

```
> C:\TURBOC\FN.C
  drive: C:, dir: \TURBOC\, file: FN, ext: .C,
  flags: :dfe
> FILE.C
  drive: , dir: , file: FILE, ext: .C, flags: fe
> \TURBOC\SUBDIR\NOEXT.
  drive: , dir: \TURBOC\SUBDIR\, file: NOEXT,
  ext: ., flags: dfe
> C:MYFILE
  drive: C:, dir: , file: MYFILE, ext: , flags: :f
```

fnsplit

Function Splits a full path name into its components.

Syntax

```
#include <dir.h>
int fnsplit(const char *path, char *drive, char *dir,
            char *name, char *ext);
```

Prototype in dir.h

Remarks **fnsplit** takes a file's full path name (*path*) as a string in the form

```
X:\DIR\SUBDIR\NAME.EXT
```

and splits *path* into its four components. It then stores those components in the strings pointed to by *drive, dir, name,* and *ext.* (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.)

The maximum sizes for these strings are given by the constants MAXDRIVE, MAXDIR, MAXPATH, MAXNAME, and MAXEXT (defined in dir.h), and each size includes space for the null-terminator.

Constant	(Max)	String
MAXPATH	(80)	*path*
MAXDRIVE	(3)	*drive*; includes colon (:)
MAXDIR	(66)	*dir*; includes leading and trailing backslashes (\)
MAXFILE	(9)	*name*
MAXEXT	(5)	*ext*; includes leading dot (.)

fnsplit assumes that there is enough space to store each non-NULL component.

When **fnsplit** splits *path*, it treats the punctuation as follows:

- *drive* includes the colon (C:, A:, etc.).
- *dir* includes the leading and trailing backslashes (\turboc\include\, \source\, etc.).
- *name* includes the file name.
- *ext* includes the dot preceding the extension (.C, .EXE, etc.).

fnmerge and **fnsplit** are invertible; if you split a given *path* with **fnsplit**, then merge the resultant components with **fnmerge**, you end up with *path*.

Return value

fnsplit returns an integer (composed of five flags, defined in dir.h) indicating which of the full path name components were present in *path*; these flags and the components they represent are

EXTENSION	An extension
FILENAME	A file name
DIRECTORY	A directory (and possibly subdirectories)
DRIVE	A drive specification (see dir.h)
WILDCARDS	Wildcards (* or ?)

Portability

fnsplit is available on DOS systems only.

See also

fnmerge

Example

See **fnmerge**

fopen

Function Opens a stream.

Syntax #include <stdio.h>
FILE *fopen(const char *_filename_, const char *_mode_);

Prototype in stdio.h

Remarks **fopen** opens the file named by _filename_ and associates a stream with it. **fopen** returns a pointer to be used to identify the stream in subsequent operations.

The _mode_ string used in calls to **fopen** is one of the following values:

_r_	Open for reading only.
_w_	Create for writing.
_a_	Append; open for writing at end-of-file or create for writing if the file does not exist.
_r+_	Open an existing file for update (reading and writing).
_w+_	Create a new file for update.
_a+_	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, you can append a _t_ to the _mode_ string (_rt_, _w+t_, etc.). Similarly, to specify binary mode, you can append a _b_ to the _mode_ string (_wb_, _a+b_, etc.). **fopen** also allows the _t_ or _b_ to be inserted between the letter and the + character in the mode string; for example, _rt+_ is equivalent to _r+t_.

If a _t_ or _b_ is not given in the _mode_ string, the mode is governed by the global variable __fmode_. If __fmode_ is set to O_BINARY, files will be opened in binary mode. If __fmode_ is set to O_TEXT, they will be opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be followed directly by input without an

intervening **fseek** or **rewind**, and input cannot be directly followed by output without an intervening **fseek, rewind**, or an input that encounters end-of-file.

Return value On successful completion, **fopen** returns a pointer to the newly opened stream. In the event of error, it returns NULL.

Portability **fopen** is available on UNIX systems and is compatible with ANSI C. It is defined by Kernighan and Ritchie.

See also **creat, dup, fclose, fdopen, ferror,** *_fmode* (variable), **fopen, fread, freopen, fseek, fwrite, open, rewind, setbuf, setmode**

Example

```
/* Program to create a backup of the AUTOEXEC.BAT file */

#include <stdio.h>

main()
{
   FILE *in, *out;

   if ((in = fopen("\\AUTOEXEC.BAT", "rt")) == NULL)
   {
      fprint(stderr, "Cannot open input file.\n");
      return (1);
   }

   if ((out = fopen("\\AUTOEXEC.BAK", "wt")) == NULL)
   {
      fprint(stderr, "Cannot open output file.\n");
      return (1);
   }

   while (!feof(in))
      fputc(fgetc(in), out);

   fclose(in);
   fclose(out);
}
```

FP_OFF

Function Gets a far address offset.

Syntax #include <dos.h>
unsigned FP_OFF(*farpointer*);

Prototype in dos.h

Remarks The **FP_OFF** macro can be used to get the offset of the far pointer *farpointer*.

Return value **FP_OFF** returns an **unsigned** integer value representing an offset value.

See also **FP_SEG, MK_FP, movedata, segread**

Example

```
#include <stdio.h>
#include <dos.h>

main ()
{
   char far *ptr;
   unsigned seg, off;
   ptr = MK_FP(0xB000,0);
   seg = FP_SEG(ptr);
   off = FP_OFF(ptr);
   printf("far ptr %Fp, segment %04x,"
          "offset %04x\n", ptr,seg,off);
}
```

Program output

```
far ptr B000:0000, segment b000, offset 0000
```

_fpreset

Function Reinitializes floating-point math package.

Syntax void _fpreset(void);

Prototype in float.h

Remarks **_fpreset** reinitializes the floating-point math package. This function is usually used in conjunction with **system** or the **exec...** or **spawn...** functions.

Note: Under DOS versions prior to 2.0 and 3.0, if an 8087/80287 coprocessor is used in a program, a child process (executed by **system** or by an **exec...** or **spawn...** function) might alter the parent process's floating-point state.

If you use an 8087/80287, take the following precautions:

- Do not call **system**, or an **exec...** or **spawn...** function, while a floating-point expression is being evaluated.
- Call **_fpreset** to reset the floating-point state after using **system**, **exec...**, or **spawn...** if there is *any* chance that the child process performed a floating-point operation with the 8087/80287.

Return value None.

See also **_clear87, _control87, exec..., spawn..., _status87, system**

fprintf

Function Writes formatted output to a stream.

Syntax

```
#include <stdio.h>

int fprintf(FILE *stream,
        const char *format[, argument, ...]);
```

Prototype in stdio.h

Remarks **fprintf** accepts a series of arguments, applies to each a format specification contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification.

Return value **fprintf** returns the number of bytes output. In the event of error, it returns EOF.

Portability **fprintf** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **cprintf, fscanf, printf, putc, sprintf**

Example See **printf**

FP_SEG

Function Gets far address segment.

Syntax #include <dos.h>
unsigned FP_SEG(*farpointer*);

Prototype in dos.h

Remarks **FP_SEG** is a macro that gets the segment value of the far pointer *farpointer*.

Return value **FP_SEG** returns an **unsigned** integer representing a segment value.

See also **FP_OFF, MK_FP**

Example See **FP_OFF**

fputc

Function Puts a character on a stream.

Syntax #include <stdio.h>
int fputc(int *c*, FILE **stream*);

Prototype in stdio.h

Remarks **fputc** outputs character *c* to the named stream.

Return value On success, **fputc** returns the character *c*. On error, it returns EOF.

Portability **fputc** is available on UNIX systems and is compatible with ANSI C.

See also **fgetc, putc**

fputchar

Function Outputs a character on *stdout*.

Syntax
```
#include <stdio.h>
int fputchar(int c);
```

Prototype in stdio.h

Remarks **fputchar** outputs character *c* to *stdout*. **fputchar**(*c*) is the same as **fputc**(*c*, *stdout*).

Return value On success, **fputchar** returns the character *c*. On error, it returns EOF.

Portability **fputc** is available on UNIX systems.

See also **fgetchar, putchar**

fputs

Function Outputs a string on a stream.

Syntax
```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Prototype in stdio.h

Remarks **fputs** copies the null-terminated string *s* to the given output stream; it does not append a newline character, and the terminating null character is not copied.

Return value On successful completion, **fputs** returns the last character written. Otherwise, it returns a value of EOF.

Portability **fputs** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **fgets, gets, puts**

fread

Function Reads data from a stream.

Syntax

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n,
             FILE *stream);
```

Prototype in stdio.h

Remarks **fread** reads *n* items of data, each of length *size* bytes, from the given input stream, into a block pointed to by *ptr*.

The total number of bytes read is ($n \times size$).

Return value On successful completion, **fread** returns the number of items (not bytes) actually read. It returns a short count (possibly 0) on end-of-file or error.

Portability **fread** is available on all UNIX systems and is compatible with ANSI C.

See also **fopen, fwrite, printf, read**

free

Function Frees allocated block.

Syntax

```
void free(void *block);
```

Prototype in stdlib.h, alloc.h

Remarks **free** deallocates a memory block allocated by a previous call to **calloc, malloc,** or **realloc**.

Return value None.

Portability **free** is available on UNIX systems and is compatible with ANSI C.

See also **calloc, freemem, malloc, realloc, strdup**

freemem

Function Frees a previously allocated DOS memory block.

Syntax int freemem(unsigned *segx*);

Prototype in dos.h

Remarks **freemem** frees a memory block allocated by a previous call to **allocmem**. *segx* is the segment address of that block.

Return value **freemem** returns 0 on success. In the event of error, it returns –1, and *errno* is set to

ENOMEM Insufficient memory

See also **allocmem, free**

freopen

Function Replaces a stream.

Syntax

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

Prototype in stdio.h

Remarks **freopen** substitutes the named file in place of the open *stream*. It closes *stream*, regardless of whether the open succeeds. **freopen** is useful for changing the file attached to *stdin*, *stdout*, or *stderr*.

The *mode* string used in calls to **fopen** is one of the following values:

- *r* Open for reading only.
- *w* Create for writing.
- *a* Append; open for writing at end-of-file or create for writing if the file does not exist.
- *r+* Open an existing file for update (reading and writing).

w+ Create a new file for update.

a+ Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, you can append a *t* to the *mode* string (*rt*, *w+t*, etc.); similarly, to specify binary mode, you can append a *b* to the *mode* string (*wb*, *a+b*, etc.).

If a *t* or *b* is not given in the *mode* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files will be opened in binary mode. If *_fmode* is set to O_TEXT, they will be opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be directly followed by input without an intervening **fseek** or **rewind**, and input cannot be directly followed by output without an intervening **fseek**, **rewind**, or an input that encounters end-of-file.

Return value On successful completion, **freopen** returns the argument *stream*. In the event of error, it returns NULL.

Portability **freopen** is available on UNIX systems and is compatible with ANSI C.

See also **fclose**, **fdopen**, **fopen**, **open**, **setmode**

Example See **fopen**

frexp

Function Splits a **double** number into mantissa and exponent.

Syntax

```
#include <math.h>
double frexp(double x, int *exponent);
```

Prototype in math.h

Remarks **frexp** calculates the mantissa *m* (a **double** greater than or equal to 0.5, and less than 1) and the integer value *n* such that *x* (the original **double** value) equals $m \times 2^n$. **frexp** stores *n* in the integer that *exponent* points to.

Return value **frexp** returns the mantissa *m*.

Error-handling for **frexp** can be modified through the function **matherr**.

Portability **frexp** is available on UNIX systems and is compatible with ANSI C.

See also **exp, ldexp**

---

fscanf

Function Scans and formats input from a stream.

Syntax

```
#include <stdio.h>
int fscanf(FILE *stream,
          const char *format[, address, ...]);
```

Prototype in stdio.h

Remarks **fscanf** scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specification passed to **fscanf** in the format string pointed to by *format*. Finally, **fscanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

fscanf may stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value **fscanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If **fscanf** attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

Portability **fscanf** is available on UNIX systems and is defined in Kernighan and Ritchie. It is compatible with ANSI C.

See also **atof, cscanf, fprintf, printf, scanf, sscanf, vfscanf, vscanf, vsscanf**

fseek

Function Repositions a file pointer on a stream.

Syntax #include <stdio.h>
int fseek(FILE **stream*, long int *offset*, int *whence*);

Prototype in stdio.h

Remarks **fseek** sets the file pointer associated with *stream* to a new position that is *offset* bytes beyond the file location given by *whence*. For text fode streams, *offset* should be 0 or a value returned by **ftell**.

whence must be one of the values 0, 1, or 2, which represent three symbolic constants (defined in stdio.h) as follows:

whence		**File Location**
SEEK_SET	(0)	File beginning
SEEK_CUR	(1)	Current file pointer position
SEEK_END	(2)	End-of-file

fseek discards any character pushed back using **ungetc**.

fseek is used with stream I/O. For file handle I/O, use **lseek**.

After **fseek**, the next operation on an update file can be either input or output.

Return value **fseek** returns 0 if the pointer is successfully moved and returns a nonzero value on failure.

Portability **fseek** is available on all UNIX systems and is compatible with ANSI C.

See also **fgetpos, fopen, fsetpos, ftell, lseek, rewind, setbuf, tell**

Example

```
#include <stdio.h>
/* Returns the number of bytes in file stream */
long filesize(FILE *stream)
{
   long curpos,length;
   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return(length);
}

main ()
{
   FILE *stream;
   stream = fopen("MYFILE.TXT", "r");
   printf("filesize of MYFILE.TXT is %ld"
          "bytes\n", filesize(stream));
}
```

Program output

```
filesize of MYFILE.TXT is 15 bytes
```

fsetpos

Function Positions the file pointer of a stream.

Syntax #include <stdio.h>
int fsetpos(FILE **stream*, const fpos_t **pos*);

Prototype in stdio.h

Remarks **fsetpos** sets the file pointer associated with *stream* to a new position. The new position is the value obtained by a previous call to **fgetpos** on that stream. It also clears the end-of-file indicator on the file that *stream* points to and undoes any effects of **ungetc** on that file. After a call to **fsetpos**, the next operation on the file can be input or output.

Return value On success **fsetpos** returns 0. On failure it returns a nonzero value, and sets *errno* to a nonzero value.

Portability **fsetpos** is compatible with ANSI C.

See also **fgetpos, fseek, ftell**

fstat

Function Gets open file information.

Syntax

```
#include <sys\stat.h>
int fstat(int handle, struct stat *statbuf);
```

Prototype in sys\stat.h

Remarks **fstat** stores information in the **stat** structure about the open file or directory associated with *handle*.

statbuf points to the **stat** structure (defined in sys\stat.h). That structure contains the following fields:

st_mode	Bit mask giving information about the open file's mode
st_dev	Drive number of disk containing the file, or file handle if the file is on a device
st_rdev	Same as *st_dev*
st_nlink	Set to the integer constant 1
st_size	Size of the open file in bytes
st_atime	Most recent time the open file was modified
st_mtime	Same as *st_atime*
st_ctime	Same as *st_atime*

The **stat** structure contains three more fields not mentioned here. They contain values that are not meaningful under DOS.

The bit mask that gives information about the mode of the open file includes the following bits.

One of the following bits will be set:

S_IFCHR	Set if *handle* refers to a device.
S_IFREG	Set if an ordinary file is referred to by *handle*.

One or both of the following bits will be set:

S_IWRITE	Set if user has permission to write to file.
S_IREAD	Set if user has permission to read to file.

The bit mask also includes the read/write bits; these are set according to the file's permission mode.

Return value `fstat` returns 0 if it has successfully retrieved the information about the open file. On error (failure to get the information), it returns –1 and sets *errno* to

EBADF	Bad file handle

See also **access, chmod, stat**

ftell

Function Returns the current file pointer.

Syntax
#include <stdio.h>
long int ftell(FILE **stream*);

Prototype in stdio.h

Remarks **ftell** returns the current file pointer for *stream*. The offset is measured in bytes from the beginning of the file.

The value returned by **ftell** can be used in a subsequent call to **fseek**.

Return value **ftell** returns the current file pointer position on success. It returns –1L on error, and sets *errno* to a positive value.

Portability **ftell** is available on all UNIX systems and is compatible with ANSI C.

See also **fgetpos, fseek, fsetpos, lseek, rewind, tell**

Example See **fseek**

ftime

Function Stores current time in **timeb** structure.

Syntax #include <sys\timeb.h>
void ftime(struct timeb **buf*)

Prototype in sys\timeb.h

Remarks **ftime** determines the current time and fills in the fields in the **timeb** structure pointed to by *buf*. The **timeb** structure contains four fields: *time, millitm, timezone,* and *dstflag*.

- The *time* field provides the time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970.
- The *millitm* field is the fractional part of a second in milliseconds.
- The *timezone* field is the difference in minutes between GMT and the local time. This value is computed going west from GMT. **ftime** gets this field from the global variable *timezone,* which is set by the **tzset** function.
- The *dstflag* field is set to 0 if daylight savings time *is not* in effect for the local time zone, and to a nonzero value if daylight savings time *is* in effect for the local time zone. This field will be set to nonzero only if the global variable *daylight* (set by the **tzset** function) is nonzero, indicating that daylight savings is in effect for the given date and time.

Note: ftime will call **tzset**. It isn't necessary to call **tzset** explicitly when you use **ftime**.

Return value None.

Portability **ftime** is available on UNIX System V systems.

See also **asctime, ctime, gmtime, localtime, stime, time, tzset**

Example

```
#include <stdio.h>
#include <sys\timeb.h>

main()
{
   struct timeb buf;

   ftime(&buf);
```

```
   printf("%ld Seconds since 1-1-70 GMT\n", buf.time);
   printf("plus %d milliseconds\n", buf.millitm);
   printf("%d Minutes from GMT\n", buf.timezone);
   printf("Daylight savings %s in effect\n",
      buf.dstflag ? "is" : "is not");
}
```

fwrite

Function Writes to a stream.

Syntax
#include <stdio.h>
size_t fwrite(const void **ptr*, size_t *size*,
size_t *n*, FILE **stream*);

Prototype in stdio.h

Remarks **fwrite** appends *n* items of data, each of length *size* bytes, to the given output file. The data written begins at *ptr*.

The total number of bytes written is ($n \times size$).

ptr in the declarations is a pointer to any object.

Return value On successful completion, **fwrite** returns the number of items (not bytes) actually written. It returns a short count on error.

Portability **fwrite** is available on all UNIX systems and is compatible with ANSI C.

See also **fopen, fread**

gcvt

Function Converts floating-point number to a string.

Syntax
#include <dos.h>
char *gcvt(double *value*, int *ndec*, char **buf*);

Prototype in stdlib.h

Remarks **gcvt** converts *value* to a null-terminated ASCII string and stores the string in *buf*. It produces *ndec* significant digits in Fortran F-format, if possible; otherwise, it returns the

value in the **printf** E-format (ready for printing). It may suppress trailing zeros.

Return value **gcvt** returns the address of the string pointed to by *buf*.

Portability **gcvt** is available on UNIX.

See also **ecvt, fcvt**

geninterrupt

Function Generates a software interrupt.

Syntax #include <dos.h>
void geninterrupt(int *intr_num*);

Prototype in dos.h

Remarks The **geninterrupt** macro triggers a software trap for the interrupt given by *intr_num*. The state of all registers after the call depends on the interrupt called.

Return value None.

Portability **geninterrupt** is unique to the 8086 architecture.

See also **bdos, bdosptr, getvect, int86, int86x, intdos, intdosx, intr**

getarccoords

Function Gets coordinates of the last call to **arc**.

Syntax #include <graphics.h>
void far getarccoords(struct arccoordstype
far **arccoords*);

Prototype in graphics.h

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. These values are useful if you need to make a line meet at the end of an arc.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **arc, fillellipse, sector**

Examples See **arc**

getaspectratio

Function Retrieves the current graphics mode's aspect ratio.

Syntax #include <graphics.h>
void far getaspectratio(int far **xasp*, int far **yasp*);

Prototype in graphics.h

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.

Portability A similar routine exists in Turbo Pascal 4.0.

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

Examples See **arc**

getbkcolor

Function Returns the current background color.

Syntax #include <graphics.h>
int far getbkcolor(void);

Prototype in graphics.h

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

Return value **getbkcolor** returns the current background color.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getcolor, getmaxcolor, getpalette, setbkcolor**

Example

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   int svcolor;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* save current bkcolor */
   svcolor = getbkcolor();
   /* change bkcolor */
   setbkcolor(svcolor ^ 1);
   /* wait 5 seconds */
   delay(5000);
   /* restore old bkcolor */
   setbkcolor(svcolor);
   getche();
   closegraph();
}
```

getc

Function Gets character from stream.

Syntax
```
#include <stdio.h>
int getc(FILE *stream);
```

Prototype in stdio.h

Remarks **getc** is a macro that returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Return value On success, **getc** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

Portability **getc** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **fgetc, getch, getchar, getche, gets, putc, putchar, ungetc**

getcbrk

Function Gets control-break setting.

Syntax
```
int getcbrk(void);
```

Prototype in dos.h

Remarks **getcbrk** uses the DOS system call 0x33 to return the current setting of control-break checking.

Return value **getcbrk** returns 0 if control-break checking is off, or 1 if checking is on.

Portability **getcbrk** is unique to DOS.

See also **ctrlbrk, setcbrk**

getch

Function	Gets character from keyboard, does not echo to screen.
Syntax	int getch(void);
Prototype in	conio.h
Remarks	**getch** reads a single character directly from the console, without echoing to the screen. **getch** uses *stdin*.
Return value	**getch** returns the character read from the keyboard.
Portability	**getch** is unique to DOS.
See also	**cgets, fgetc, getc, getchar, getche, getpass, kbhit, putch, ungetch**

getchar

Function	Gets character from *stdin*.
Syntax	#include <stdio.h> int getchar(void);
Prototype in	stdio.h
Remarks	**getchar** is a macro that returns the next character on the named input stream *stdin*. It is defined to be **getc**(*stdin*).
Return value	On success, **getchar** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.
Portability	**getchar** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.
See also	**fgetc, fgetchar, getc, getch, getche, putc, putchar, ungetc**

getche

Function Gets character from the console, echoes to screen.

Syntax int getche(void);

Prototype in conio.h

Remarks **getche** reads a single character from the console and echoes it to the current text window, using direct video or BIOS.

Return value **getche** returns the character read from the keyboard.

Portability **getche** is unique to DOS.

See also **cgets, cscanf, fgetc, getc, getch, getchar, kbhit, putch, ungetch**

getcolor

Function Returns the current drawing color.

Syntax

```
#include <graphics.h>
int far getcolor(void);
```

Prototype in graphics.h

Remarks **getcolor** returns the current drawing color.

The drawing color is the value to which pixels are set when lines, etc., 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getbkcolor, getmaxcolor, getpalette, setcolor**

Example

```
#include <graphics.h>
#include <conio.h>
```

```
main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   int svcolor;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* save current drawing color */
   svcolor = getcolor();
   /* set drawing color to color stored in palette entry #3 */
   setcolor(3);
   /* small colored circle */
   circle(100, 100, 5);
   /* restore old drawing color */
   setcolor(svcolor);
   getche();
   closegraph();
}
```

getcurdir

Function Gets current directory for specified drive.

Syntax int getcurdir(int *drive*, char **directory*);

Prototype in dir.h

Remarks **getcurdir** gets the name of the current working directory for the drive indicated by *drive*.

drive specifies a drive number (0 for default, 1 for A, etc.).

directory points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.

Return value **getcurdir** returns 0 on success or –1 in the event of error.

Portability **getcurdir** is unique to DOS.

See also **chdir, getcwd, getdisk, mkdir, rmdir**

Example

```
#include <dir.h>
#include <stdio.h>
#include <string.h>
```

```
char *current_directory(char *path)
{
   strcpy(path, "X:\\");
   path[0] = 'A' + getdisk();
   getcurdir(0, path+3);
   return(path);
}

main()
{
   char curdir[MAXPATH];
   current_directory(curdir);
   printf("The current directory is %s\n", curdir);
}
```

Program output

```
The current directory is C:\TURBOC
```

getcwd

Function Gets current working directory.

Syntax #include <dir.h>
char *getcwd(char **buf*, int *buflen*);

Prototype in dir.h

Remarks **getcwd** gets the full path name of the current working directory up to *buflen* bytes long, including the drive, and stores it in *buf*. If the full path name length (including the null-terminator) is longer than *buflen* bytes, an error occurs.

If *buf* is NULL, a buffer *buflen* bytes long will be allocated for you with **malloc**. You can later free the allocated buffer by passing the return value of **getcwd** to the function **free**.

Return value **getcwd** returns the following values:

- If *buf* is not NULL on input, **getcwd** returns *buf* on success, NULL on error.
- If *buf* is NULL on input, **getcwd** returns a pointer to the allocated buffer.

In the event of an error return, the global variable *errno* is set to one of the following:

ENODEV	No such device
ENOMEM	Not enough core
ERANGE	Result out of range

Portability **getcwd** is unique to DOS.

See also **chdir, getcurdir, getdisk, mkdir, rmdir**

getdate

Function Gets system date.

Syntax #include <dos.h>
void getdate(struct date **datep*);

Prototype in dos.h

Remarks **getdate** fills in the **date** structure (pointed to by *datep*) with the system's current date.

The **date** structure is defined as follows:

```
struct date {
   int da_year;                     /* current year */
   char da_day;                 /* day of the month */
   char da_mon;                  /* month (1 = Jan) */
};
```

Return value None.

Portability **getdate** is unique to DOS.

See also **ctime, gettime, setdate, settime**

Example

```
#include <stdio.h>
#include <dos.h>

main()
{
   struct date today;
   struct time now;
   getdate(&today);
   printf("Today's date is %d/%d/%d\n",
          today.da_mon, today.da_day,
          today.da_year);
   gettime(&now);
```

```
    printf("The time is %02d:%02d:%02d.%02d\n",
           now.ti_hour, now.ti_min, now.ti_sec,
           now.ti_hund);
}
```

Program output

```
Today's date is 1/1/1980
The time is 17:08:22.42
```

getdefaultpalette

Function Returns the palette definition structure.

Syntax #include <graphics.h>
void far *far getdefaultpalette(void);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getpalette, initgraph**

struct pallettetype * far _Cdecl getdefaultpallette (void);

getdfree

Function Gets disk free space.

Syntax #include <dos.h>
void getdfree(unsigned char *drive*,
struct dfree **dtable*);

Prototype in dos.h

Remarks **getdfree** accepts a drive specifier in *drive* (0 for default, 1 for A, etc.) and fills in the *dfree* structure pointed to by *dtable* with disk characteristics.

The *dfree* structure is defined as follows:

```
struct dfree {
   unsigned df_avail;                /* available clusters */
   unsigned df_total;                    /* total clusters */
   unsigned df_bsec;                   /* bytes per sector */
   unsigned df_sclus;                /* sectors per cluster */
};
```

Return value **getdfree** returns no value. In the event of an error, *df_sclus* in the *dfree* structure is set to –1.

Portability **getdfree** is unique to DOS.

See also **getfat, getfatd**

getdisk

Function Gets current drive number.

Syntax int getdisk(void);

Prototype in dir.h

Remarks **getdisk** gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, etc. (equivalent to DOS function 0x19).

Return value **getdisk** returns the current drive number.

Portability **getdisk** is unique to DOS.

See also **getcurdir, getcwd, setdisk**

Example See **getcurdrive**

getdrivername

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

Syntax #include <graphics.h>
char *far getdrivername(void);

Prototype in graphics.h

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

Return value **getdrivename** returns a pointer to a string with the name of the currently loaded graphics driver.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **initgraph**

---

getdta

Function Gets disk-transfer address.

Syntax char far *getdta(void);

Prototype in dos.h

Remarks **getdta** returns the current setting of the disk-transfer address (DTA).

In the small and medium memory models, it is assumed that the segment is the current data segment. If C is used exclusively, this will be the case, but assembly routines can set the disk transfer address to any hardware address.

In the compact, large, or huge memory models, the address returned by **getdta** is the correct hardware address and can be located outside the program.

Return value **getdta** returns a far pointer to the current disk-transfer address.

Portability **getdta** is unique to DOS.

See also **fcb** (structure), **setdta**

---

getenv

Function Gets a string from environment.

Syntax char *getenv(const char **name*);

Prototype in stdlib.h

Remarks

getenv returns the value of a specified variable. The variable name can be in either uppercase or lowercase, but it must not include the equal sign (=) character. If the specified environment variable does not exist, **getenv** returns an empty string.

Return value

On success, **getenv** returns the value associated with *name*. If the specified *name* is not defined in the environment, **getenv** returns an empty string.

Note: Environment entries must not be changed directly. If you want to change an environment value, you must use the **putenv** function.

Portability

getenv is available on UNIX systems and is compatible with ANSI C.

See also

environ (variable), **getpsp**, **putenv**

Example

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *path, *dummy = NULL;
   path = getenv("PATH");
   dummy = getenv("DUMMY");
   printf("PATH = %s\n", path);
   printf("old value of DUMMY: %s\n",
         (dummy == NULL) ? "*none*" : dummy);
   putenv("DUMMY=TURBOC");
   dummy = getenv("DUMMY");
   printf("new value of DUMMY: %s\n", dummy);
}
```

Program output

```
PATH = C:\BIN;C:\BIN\DOS;C:\
old value of DUMMY: *none*
new value of DUMMY: TURBOC
```

getfat

Function Gets file-allocation table information for given drive.

Syntax #include <dos.h>
void getfat(unsigned char *drive*,
struct fatinfo **dtable*);

Prototype in dos.h

Remarks **getfat** gets information from the file-allocation table for the drive specified by *drive* (0 for default, 1 for A, 2 for B, etc.). *dtable* points to the **fatinfo** structure to be filled in.

The **fatinfo** structure filled in by **getfat** is defined as follows:

```
struct fatinfo {
   char fi_sclus;                    /* sectors per cluster */
   char fi_fatid;                        /* the FAT id byte */
   int fi_nclus;                      /* number of clusters */
   int fi_bysec;                        /* bytes per sector */
};
```

Return value None.

Portability **getfat** is unique to DOS.

See also **getdfree, getfatd**

getfatd

Function Gets file-allocation table information.

Syntax #include <dos.h>
void getfatd(struct fatinfo **dtable*);

Prototype in dos.h

Remarks **getfatd** gets information from the file-allocation table of the default drive. *dtable* points to the **fatinfo** structure to be filled in.

The **fatinfo** structure filled in by **getfatd** is defined as follows:

```
struct fatinfo {
```

```
    char fi_sclus;                    /* sectors per cluster */
    char fi_fatid;                    /* the FAT id byte */
    int fi_nclus;                     /* number of clusters */
    int fi_bysec;                     /* bytes per sector */
};
```

Return value None.

Portability **getfatd** is unique to DOS.

See also **getdfree, getfat**

---

getfillpattern

Function Copies a user-defined fill pattern into memory.

Syntax [#include <graphics.h>
void far getfillpattern(char far **pattern*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getfillsettings, setfillpattern**

getfillsettings

Function Gets information about current fill pattern and color.

Syntax

#include <graphics.h>
void far getfillsettings(struct fillsettingstype far **fillinfo*);

Prototype in graphics.h

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, cross-hatch, dotted, etc.). 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 cross-hatch 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 uses the current background color.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getfillpattern, setfillpattern, setfillstyle**

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   struct fillsettingstype save;
   char savepattern[8];
   char gray50[] = { 0xaa, 0x55, 0xaa, 0x55, 0xaa,
                     0x55, 0xaa, 0x55 };
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* retrieve current settings */
   getfillsettings(&save);
   /* if user-defined pattern */
   if (save.pattern == USER_FILL)
   /* then save user fill pattern */
   getfillpattern(savepattern);
```

```
   /* change fill style */
   setfillstyle(SLASH_FILL, BLUE);
   /* draw slash-filled blue bar */
   bar(0, 0, 100, 100);
   /* custom fill pattern */
   setfillpattern(gray50, YELLOW);
   /* draw customized yellow bar */
   bar(100, 100, 200, 200);
   /* if user-defined pattern */
   if (save.pattern == USER_FILL)
      /* then restore user fill pattern */
      setfillpattern(savepattern, save.color);
   else
      /* restore old style */
      setfillstyle(save.pattern, save.color);
   getche();
   closegraph();
}
```

getftime

Function Gets file date and time.

Syntax #include <io.h>
int getftime(int *handle*, struct ftime **ftimep*);

Prototype in io.h

Remarks **getftime** retrieves the file time and date for the disk file associated with the open *handle*. The **ftime** structure pointed to by *ftimep* is filled in with the file's time and date.

The **ftime** structure is defined as follows:

```
struct ftime {
   unsigned ft_tsec: 5;                    /* two seconds */
   unsigned ft_min: 6;                         /* minutes */
   unsigned ft_hour: 5;                          /* hours */
   unsigned ft_day: 5;                            /* days */
   unsigned ft_month: 4;                        /* months */
   unsigned ft_year: 7;                     /* year - 1980*/
};
```

Return value **getftime** returns 0 on success.

In the event of an error return, –1 is returned, and the global variable *errno* is set to one of the following:

EINVFNC	Invalid function number
EBADF	Bad file number

Portability **getftime** is unique to DOS.

See also **open, setftime**

---

getgraphmode

Function Returns the current graphics mode.

Syntax

```
#include <graphics.h>
int far getgraphmode(void);
```

Prototype in graphics.h

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**.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmoderange, restorecrtmode, setgraphmode**

Example

```
int cmode;
/* save current mode */
cmode = getgraphmode();
/* switch to text */
restorecrtmode();
printf("Now in text mode - press"
       "any key to go back to graphics ...");
getch();
/* back to graphics */
setgraphmode(cmode);
```

getimage

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

Syntax
#include <graphics.h>
void far getimage(int *left*, int *top*, int *right*,
int *bottom*, void far **bitmap*);

Prototype in graphics.h

Remarks **getimage** copies an image from the screen to memory.

left, *top*, *right*, and *bottom* define the area of the screen to which the rectangle is to be 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **imagesize, putimage, putpixel**

Example

```
#include <alloc.h>
#include <graphics.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   void * buffer;
   unsigned size;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   size = imagesize(0,0,20,10);
   /* get memory for image */
   buffer = malloc (size);
   /* save bits */
   getimage(0,0,20,10,buffer);

   /*   ...   */

   /* restore bits */
   putimage(0,0,buffer,COPY_PUT);
   /* free buffer */
   free(buffer);

   closegraph();
```

getlinesettings

Function Gets the current line style, pattern, and thickness.

Syntax

```
#include <graphics.h>
void far getlinesettings(struct linesettingstype
                                    far *lineinfo);
```

Prototype in graphics.h

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 it is ignored.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **setlinestyle**

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   struct linesettingstype saveline;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* save current line style */
   getlinesettings(&saveline);
   setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
   /* draw a little thick box */
   rectangle(10, 10, 17, 15);
   /* restore old line settings */
   setlinestyle(saveline.linestyle, saveline.upattern,
                saveline.thickness);
   getche();
   closegraph();
}
```

getmaxcolor

Function

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

Syntax

#include <graphics.h>
int far getmaxcolor(void);

Prototype in

graphics.h

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** will always return 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 because these adapters only support draw colors of 0 or 1.

Return value

getmaxcolor returns the highest available color value.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

getbkcolor, getcolor, getpalette, getpalettesize, setcolor

getmaxmode

Function

Returns the maximum mode number for the current driver.

Syntax

#include <graphics.h>
int far getmaxmode(void);

Prototype in

graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmodename, getmoderange**

---

getmaxx

Function Returns maximum *x* screen coordinate.

Syntax
```
#include <graphics.h>
int far getmaxx(void);
```

Prototype in graphics.h

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 on the screen, and so on.

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

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmaxy, getx**

Example
```
printf("The screen resolution is %d pixels by %d pixels.\n",
   getmaxx()+1, getmaxy()+1);
```

---

getmaxy

Function Returns maximum *y* screen coordinate.

Syntax
```
#include <graphics.h>
int far getmaxy(void);
```

Prototype in graphics.h

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 on the screen, and so on.

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

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmaxx, getx**

Example See **getmaxx**

getmodename

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

Syntax

```
#include <graphics.h>
char *far getmodename(int mode_number);
```

Prototype in graphics.h

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 imbedded in each driver. The return values ("320 × 200 CGA P1," "640 × 200 CGA", etc.) are useful for building menus or displaying status.

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

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmaxmode, getmoderange**

getmoderange

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

Syntax #include <graphics.h>
void far getmoderange(int *graphdriver*, int far **lomode*,
int far **himode*);

Prototype in graphics.h

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 in **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 will be given.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getgraphmode, getmaxmode, getmodename, initgraph, setgraphmode**

Example

```
#include <graphics.h>

main()
{
   int lo, hi;

   getmoderange(CGA, &lo, &hi);
   printf("CGA supports modes %d through %d\n", lo, hi);
}
```

getpalette

Function Gets information about the current palette.

Syntax #include <graphics.h>
void far getpalette(struct palettetype far **palette*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

Example

```
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   struct palettetype palette;
   int color;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* get current palette */
   getpalette(&palette);
   for(color=0; color<palette.size; color++)
   {
      /* draw some colorful bars */
      setfillstyle(SOLID_FILL, color);
```

```
      bar(20*(color-1), 0, 20*color, 20);
   }

   /* only if more than 1 color */
   if (palette.size > 1)
   {
      /* switch colors randomly */
      do
         setpalette(random(palette.size),
                    random(palette.size));
      /* until a key is hit */
      while(!kbhit());
      /* discard keystroke */
      getch();
   }

   /* restore original palette */
   setallpalette(&palette);

   closegraph();
}
```

getpalettesize

Function Returns size of palette color lookup table.

Syntax #include <graphics.h>
int far getpalettesize(void);

Prototype in graphics.h

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 will return 16.

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

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **setpalette, setallpalette**

getpass

Function Reads a password.

Syntax char *getpass(const char *prompt*);

Prototype in conio.h

Remarks **getpass** reads a password from the system console, after prompting with the null-terminated string *prompt* and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null-terminator).

Return value The return value is a pointer to a static string, which is overwritten with each call.

Portability **getpass** is available on UNIX systems.

See also **getch**

getpixel

Function Gets the color of a specified pixel.

Syntax #include <graphics.h>
unsigned far getpixel(int *x*, int *y*);

Prototype in graphics.h

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

Return value **getpixel** returns the color of the given pixel.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getimage, putpixel**

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   int i, color, max;
```

```
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   max = getmaxcolor() + 1;

   /* Change color of pixels in a diagonal line */
   for (i=1; i<200; i++) {
     color = getpixel(i,i);
     putpixel(i, i, (color ^ i) % max);
   }
   getche();
   closegraph();
}
```

getpsp

Function Gets the program segment prefix.

Syntax unsigned getpsp(void);

Prototype in dos.h

Remarks **getpsp** gets the segment address of the program segment prefix (PSP) using DOS call 0x62.
This call exists only in DOS 3.x. For versions of DOS 2.x and 3.x, the global variable *_psp* set by the start-up code can be used instead.

Return value **getpsp** returns the segment address of the PSP.

Portability **getpsp** is unique to DOS 3.x and is not available under earlier versions of DOS.

See also **getenv**, *_psp* (variable)

gets

Function Gets a string from *stdin*.

Syntax char *gets(char *s);

Prototype in stdio.h

Remarks **gets** collects a string of characters, terminated by a carriage return, from the standard input stream *stdin*,

and puts it into *s*. The carriage return is replaced by a null character (\0) in *s*.

Unlike **scanf, gets** allows input strings to contain some whitespace characters (spaces, tabs). **gets** returns when it encounters a carriage return; everything up to the carriage return is copied into *s*.

Return value **gets**, on success, returns the string argument *s*; it returns NULL on end-of-file or error.

Portability **gets** is available on UNIX systems and is compatible with ANSI C.

See also **cgets, ferror, fgets, fputs, getc, puts**

Example

```
#include <stdio.h>

main()
{
   char buff[133];

   puts("Enter a string: ");
   if (gets(buff) != NULL)
      printf("String = '%s'\n", buff);
}
```

gettext

Function Copies text from text mode screen to memory.

Syntax int gettext(int *left*, int *top*, int *right*, int *bottom*,
void **destin*);

Prototype in conio.h

Remarks **gettext** stores the contents of an onscreen text rectangle defined by *left*, *top*, *right*, and *bottom*, into the area of memory pointed to by *destin*.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

gettext reads the contents of the rectangle into memory sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the

cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as

$$bytes = (h \text{ rows}) \times (w \text{ columns}) \times 2$$

Return value **gettext** returns 1 if the operation succeeds. It returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

Portability **gettext** works only on IBM PCs and BIOS-compatible systems.

See also **movetext, puttext**

Example

```
char buf[20*10*2];
/* save rectangle */
gettext(1,1,20,10,buf);

/* ... */

/* restore screen */
puttext(1,1,buf);
```

gettextinfo

Function Gets text mode video information.

Syntax #include <conio.h>
void gettextinfo(struct text_info *r);

Prototype in conio.h

Remarks **gettextinfo** fills in the **text_info** structure pointed to by *r* with the current text video information.

The **text_info** structure is defined in conio.h as follows:

```
struct text_info {
   unsigned char winleft;      /* left window coordinate */
   unsigned char wintop;        /* top window coordinate */
   unsigned char winright;    /* right window coordinate */
   unsigned char winbottom;  /* bottom window coordinate */
   unsigned char attribute;            /* text attribute */
   unsigned char normattr;           /* normal attribute */
   unsigned char currmode;    /* BW40, BW80, C40, or C80 */
   unsigned char screenheight;           /* bottom - top */
   unsigned char screenwidth;            /* right - left */
   unsigned char curx; /* x coordinate in current window */
```

```
    unsigned char cury;  /* y coordinate in current window */
};
```

Return value **gettextinfo** returns nothing; the results are returned in the structure pointed to by *r*.

Portability **gettextinfo** works only with IBM PCs and compatibles.

See also **textattr, textbackground, textcolor, textmode, wherex, wherey, window**

Example

```
#include <conio.h>
struct text_info initial_info;

main()
{
   gettextinfo(&initial_info);

   /* ... */

   /* Restore text mode to original value */
   textmode(initial_info.currmode);
}
```

gettextsettings

Function Gets information about the current graphics text font.

Syntax

```
#include <graphics.h>
void far gettextsettings(struct textsettingstype
                            far *texttypeinfo);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   struct textsettingstype oldtext;
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");
   /* get current settings */
   gettextsettings(&oldtext);

   /* Switch to horizontal, upper left-justified,
      Gothic font, scaled by a factor of 5 */

   settextjustify(LEFT_TEXT, TOP_TEXT);
   settextstyle(GOTHIC_FONT, HORIZ_DIR, 5);
   outtext("Gothic Text");

   /* Restore previous settings */

   settextjustify(oldtext.horiz, oldtext.vert);
   settextstyle(oldtext.font, oldtext.direction,
             oldtext.charsize);
   getche();
   closegraph();
}
```

gettime

Function Gets system time.

Syntax #include <dos.h>
void gettime(struct time **timep*);

Prototype in dos.h

Remarks **gettime** fills in the **time** structure pointed to by *timep* with the system's current time.

The **time** structure is defined as follows:

```
struct time {
   unsigned char ti_min;                           /* minutes */
   unsigned char ti_hour;                            /* hours */
   unsigned char ti_hund;         /* hundredths of seconds */
   unsigned char ti_sec;                           /* seconds */
};
```

Return value None.

Portability **gettime** is unique to DOS.

See also **getdate, setdate, settime, stime, time**

Example See **getdate**

getvect

entry

Function Gets interrupt vector.

Syntax void interrupt(*getvect(int *interruptno*)) ( );

Prototype in dos.h

Remarks Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

getvect reads the value of the interrupt vector given by *interruptno* and returns that value as a (**far**) pointer to an interrupt function. The value of *interruptno* can be from 0 to 255.

Return value **getvect** returns the current 4-byte value stored in the interrupt vector named by *interruptno*.

Portability **getvect** is unique to DOS.

See also **disable, enable, geninterrupt, setvect**

Example

```
#include <stdio.h>
#include <dos.h>
```

```
/* getvect example */

void interrupt (*oldfunc)();
int looping = 1;

/* get_out - this is our new interrupt routine */

void interrupt get_out()
{
   /* restore to original interrupt routine */
   setvect(5,oldfunc);
   looping = 0;
}

/* capture_prtscr - installs a new interrupt for
   <Shift><PrtSc> */

/* arguments :  func  -- new interrupt function pointer */

void capture_prtscr(void interrupt  (*func)())
{
   /* save the old interrupt */
   oldfunc  = getvect(5);
   /* install our interrupt handler */
   setvect(5,func);
}

void main ()
{

   puts("Press <Shift><Prt Sc> to terminate");
   /* capture the print screen interrupt */
   capture_prtscr(get_out);

   /* do nothing */
   while (looping);

   puts("Success");
}
```

getverify

Function Returns the state of the DOS verify flag.

Syntax int getverify(void);

Prototype in dos.h

Remarks **getverify** gets the current state of the verify flag.

The verify flag controls output to the disk. When verify is off, writes are not verified; when verify is on, all disk writes are verified to insure proper writing of the data.

Return value **getverify** returns the current state of the verify flag, either 0 or 1.

A return of 0 = verify flag off.
A return of 1 = verify flag on.

Portability **getverify** is unique to DOS.

See also **setverify**

getviewsettings

Function Gets information about the current viewport.

Syntax #include <graphics.h>
void far getviewsettings(struct viewporttype far **viewport*);

Prototype in graphics.h

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 clipflag;
};
```

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **clearviewport, getx, gety, setviewport**

Example

```
struct viewporttype view;

/* get current setting */
getviewsettings(&view);
/* if clipping not on */
if (!view.clip)
```

```
/* turn it on */
setviewport(view.left,view.top,view.right,view.bottom,1);
```

getw

Function Gets integer from stream.

Syntax #include <stdio.h>
int getw(FILE **stream*);

Prototype in stdio.h

Remarks **getw** returns the next integer in the named input stream. It assumes no special alignment in the file.

getw should not be used when the stream is opened in text mode.

Return value **getw** returns the next integer on the input stream. On end-of-file or error, **getw** returns EOF. Because EOF is a legitimate value for **getw** to return, **feof** or **ferror** should be used to detect end-of-file or error.

Portability **getw** is available on UNIX systems.

See also **putw**

getx

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

Syntax #include <graphics.h>
int far getx(void);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getmaxx, getmaxy, getviewsettings, gety**

Example

```
int oldx, oldy;

/* Save current position */
oldx = getx();
oldy = gety();
/* draw a blob at [100,100] */
circle(100, 100, 2);
moveto(99,100);
linerel(2,0);
/* back to the old position */
moveto(oldx, oldy);
```

gety

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

Syntax #include <graphics.h>
int far gety(void);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getx**, **getviewsettings**

Example See **getx**

gmtime

Function Converts date and time to Greenwich Mean Time (GMT).

Syntax #include <time.h>
struct tm *gmtime(const time_t **timer*);

Prototype in time.h

Remarks **gmtime** accepts the address of a value returned by **time** and returns a pointer to the structure of type **tm** con-

taining the broken-down time. **gmtime** converts directly to GMT.

The global long variable *timezone* should be set to the difference in seconds between GMT and local standard time (in PST, *timezone* is $8 \times 60 \times 60$). The global variable *daylight* should be set to nonzero *only if* the standard U.S. Daylight Savings time conversion should be applied.

The **tm** structure declaration from the time.h include file is

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month (0-11), weekday (Sunday equals 0), year − 1900, day of year (0-365), and a flag that is nonzero if daylight savings time is in effect.

Return value **gmtime** returns a pointer to the structure containing the broken down time. This structure is a static that is over-written with each call.

Portability **gmtime** is available on UNIX systems and is compatible with ANSI C.

See also **asctime, ctime, ftime, localtime, stime, time, tzset**

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
   struct tm         *timeptr;
   time_t            secsnow;

   timezone = 8 * 60 * 60;
   /* get seconds since 00:00:00, 1-1-70 */
   time(&secsnow);
```

```
   /* convert to GMT */
   timeptr = gmtime(&secsnow);
   printf("The date is %d-%d-19%02d\n",
          (timeptr -> tm_mon) + 1, timeptr -> tm_mday,
          timeptr -> tm_year);
   printf("Greenwich Mean Time is %02d:%02d:%02d\n\n",
         timeptr -> tm_hour, timeptr -> tm_min,
         timeptr -> tm_sec);
}
```

Program output

```
The date is 2-2-1988
Greenwich Mean Time is 20:44:36
```

gotoxy

Function Positions cursor in text window.

Syntax void gotoxy(int *x*, int *y*);

Prototype in conio.h

Remarks **gotoxy** moves the cursor to the given position in the current text window. If the coordinates are in any way invalid, the call to **gotoxy** is ignored. An example of this is a call to **gotoxy**(40,30) when (35,25) is the bottom right position in the window.

Return value None.

Portability **gotoxy** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **wherex, wherey, window**

Example

```
gotoxy(10,20);         /* position cursor at col 10, row 20 */
```

graphdefaults

Function Resets all graphics settings to their defaults.

Syntax

```
#include <graphics.h>
void far graphdefaults(void);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **initgraph**

grapherrormsg

Function Returns a pointer to an error message string.

Syntax

```
#include <graphics.h>
char * far grapherrormsg(int errorcode);
```

Prototype in graphics.h

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 Chapter 1 for a list of error messages and mnemonics.

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

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **graphresult**

_graphfreemem

Function User hook into graphics memory deallocation.

Syntax #include <graphics.h>
void far _graphfreemem(void far **ptr*, unsigned *size*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **_graphgetmem, setgraphbufsize**

Example

```
/* Example of user-defined graph management routines */
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>

main()
{
   int errorcode;
   int graphdriver;
   int graphmode;
   graphdriver = DETECT;
   initgraph(&graphdriver, &graphmode, "c:\\");
   errorcode = graphresult();
   if (errorcode != grOk)
   {
      printf("graphics error: %s\n",grapherrormsg(errorcode));
```

```
      exit(1);
   }
   settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
   outtextxy( 100, 100, "BGI TEST");
   getche();
   closegraph();
}

void far * far _graphgetmem(unsigned size) {
   printf("_graphgetmem  called [size=%d] -- hit any"
          "key",size);
   getch(); printf("\n");
   /* use "far" heap */
   return(farmalloc(size));
}

void far _graphfreemem(void far *ptr, unsigned size) {
   printf("_graphfreemem called [size=%d] -- hit any"
          "key",size);
   getch(); printf("\n");
   /* "size" not used */
   farfree(ptr);
}
```

_graphgetmem

Function User hook into graphics memory allocation.

Syntax #include <graphics.h>
void far * far _graphgetmem(unsigned *size*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **_graphfreemem, initgraph, setgraphbufsize**

Example See **_graphfreemem**

graphresult

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

Syntax

```
#include <graphics.h>
int far graphresult(void);
```

Prototype in graphics.h

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	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	grInvalidVersionnum	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 will return 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**.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

harderr

Function Establishes a hardware error handler.

Syntax void harderr(int (**handler*)());

Prototype in dos.h

Remarks **harderr** establishes a hardware error handler for the current program. This error handler is invoked whenever an interrupt 0x24 occurs. (See the *MS-DOS Programmer's Reference Manual* for a discussion of the interrupt.)

The function pointed to by *handler* will be called when such an interrupt occurs. The handler function will be 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 non-negative, a disk error was encountered; otherwise, the error was a device error. For a disk error, *ax* ANDed with 0x00FF will give the failing drive number (1 equals A, 2 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.

peek and **peekb** can be used to retrieve device information from this driver header.

The driver header cannot be altered via **poke** or **pokeb**.

The handler can issue DOS calls 1 through 0xC; any other DOS call will corrupt 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.

Portability **harderr** is unique to DOS.

See also **hardresume, hardretn, peek, poke**

Example

```
#include <stdio.h>
#include <dos.h>

#define DISPLAY_STRING 0x09
#define IGNORE 0
#define RETRY  1
#define ABORT  2

int handler(int errval, int ax, int bp, int si)
{
   char msg[25];  int drive;
   /* device error */
   if (ax < 0)
   {
      /* Can only use DOS functions 0 - 0x0C */
      bdosptr(DISPLAY_STRING, "device error$", 0);
      hardretn(-1);            /* return to calling program */
   }
   drive = (ax & 0x00FF);
   sprintf(msg, "disk error on drive %c$", 'A' + drive);
   bdosptr(DISPLAY_STRING, msg, 0);
   return(ABORT);                  /* abort calling program */
}

main()
{
   harderr(handler);
   printf("Make sure there is no disk in drive A:\n");
   printf("Press a key when ready...\n");
   getch();
   printf("Attempting to access A:\n");
   fopen("A:ANY.FIL","r");
```

```
}
```

Program output

```
Make sure there is no disk in drive A:
Press a key when ready...
Attempting to access A:
disk error on drive A
```

hardresume

Function Hardware error handler.

Syntax void hardresume(int *axret*);

Prototype in dos.h

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 handler must return 0 for ignore, 1 for retry, and 2 for abort.

Return value None.

Portability **hardresume** is unique to DOS.

See also **harderr, hardretn**

hardretn

Function Hardware error handler.

Syntax void hardretn(int *retn*);

Prototype in dos.h

Remarks The error handler established by **harderr** can return directly to the application program by calling **hardretn.**

The handler must return 0 for ignore, 1 for retry, or 2 for abort.

Return value None.

Portability **hardretn** is unique to DOS.

See also **harderr, hardresume**

Example See **harderr**

---

highvideo

Function Selects high-intensity characters.

Syntax void highvideo(void);

Prototype in conio.h

Remarks **highvideo** selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently on the screen, but does affect those displayed by functions (such as **cprintf**) that perform direct video, text mode output *after* **highvideo** is called.

Return value None.

Portability **highvideo** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **lowvideo, normvideo, textattr, textcolor**

---

hypot

Function Calculates hypotenuse of a right triangle.

Syntax
```
#include <math.h>
double hypot(double x, double y);
```

Prototype in math.h

Remarks **hypot** calculates the value z where

$z^2 = x^2 + y^2$ and $z >= 0$

This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are *x* and *y*.

Return value On success, **hypot** returns *z*, a **double**. On error (such as an overflow), **hypot** sets *errno* to

ERANGE   Result out of range

and returns the value HUGE_VAL.

Error-handling for **hypot** can be modified through the function **matherr**.

Portability **hypot** is available on UNIX systems.

imagesize

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

Syntax

```
#include <graphics.h>
unsigned far imagesize(int left, int top,
                       int right, int bottom);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getimage, putimage**

initgraph

Function Initializes the graphics system.

Syntax

```
#include <graphics.h>
void far initgraph(int far *graphdriver,
        int far *graphmode,
        char far *pathtodriver);
```

Prototype in graphics.h

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, 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. **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 Appendix D for more information on BGIOBJ.

pathtodriver specifies the directory path where **initgraph** will look for the 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** will search for the stroked character font (*.CHR) files.

**graphdriver* is an integer that specifies the graphics driver to be used. You can give it a value using a con-

stant of the *graphics_drivers* enumeration type, defined in graphics.h and listed in the following table.

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, defined in graphics.h and listed in the following table.

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-	EGAMONOHI	3	640×350	2 color	1*
MONO	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: *graphdriver* and *graphmode* must be set to valid values from the tables above, or you will get unpredictable results. The exception is *graphdriver* = DETECT.

In the previous table, 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 8 of the *Turbo C User's Guide* (under "Color Control") and summarized in the following table.

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.

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:

- –2 cannot detect a graphics card
- –3 cannot find driver file
- –4 invalid driver
- –5 insufficient memory to load driver

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

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

Example

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
```

```
main()
{
   int g_driver, g_mode, g_error;
   detectgraph(&g_driver, &g_mode);
   if (g_driver < 0)
   {
      printf("No graphics hardware detected !\n");
      exit(1);
   }

   printf("Detected graphics driver #%d,"
          "mode #%d\n",g_driver,g_mode);
   getch();
   if (g_mode == EGAHI)
      /* override mode if EGA detected */
      g_mode = EGALO;
   initgraph(&g_driver, &g_mode, "");
   g_error = graphresult();

   if (g_error < 0)
   {
      printf("initgraph error: %s.\n",
             grapherrormsg(g_error));
      exit(1);
   }

   bar(0, 0, getmaxx()/2, getmaxy());
   getch();

   closegraph();
}
```

inport

Function Reads a word from a hardware port.

Syntax #include <dos.h>
int inport(int *portid*);

Prototype in dos.h

Remarks **inport** reads a word from the input port specified by *portid*.

Return value **inport** returns the value read.

Portability **inport** is unique to the 8086 family.

See also **inportb, outport, outportb**

inportb

Function Reads a byte from a hardware port.

Syntax unsigned char inportb(int *portid*);

Prototype in dos.h

Remarks **inportb** is a macro that reads a byte from the input port specified by *portid*.

If **inportb** is called when dos.h has been included, it will be treated as a macro that expands to inline code.

If you don't include dos.h, or if you do include dos.h and `#undef` the macro **inportb**, you will get the **inportb** function.

Return value **inportb** returns the value read.

Portability **inportb** is unique to the 8086 family.

See also **inport, outport, outportb**

insline

Function Inserts a blank line in the text window.

Syntax void insline(void);

Prototype in conio.h

Remarks **insline** inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line and the bottom line scrolls off the bottom of the window.

insline is used in text mode.

Return value None.

Portability **insline** works with IBM PCs and compatibles only; a corresponding function exists in Turbo Pascal.

See also **delline, window**

installuserdriver

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

Syntax

```
#include <graphics.h>
int far installuserdriver(char far *name,
                          int huge (*detect)(void));
```

Prototype in graphics.h

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

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, you 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 may 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 non-negative 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.

Up to ten drivers can be installed 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.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

initgraph, registerbgidriver

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>

int  Driver, Mode;

int huge detectSGA( void )              /* Autodetection logic */
{
   int found, defaultmode;

/* Detect hardware as needed...

   found = .....
*/

   if( !found ) return( grError );      /* If not present, give
                                                         error */

/* Determine default graphics mode...

   defaultmode = .... */

   return( defaultmode );

}

main()
{

   Driver = installuserdriver( "SGA", detectSGA );

   if( grOk != graphresult() ){           /* Is table full? */
      printf( "Error installing user driver SGA.\n" );
      exit( 1 );
   }
```

```
   Driver = DETECT;                      /* Do autodetection */
   initgraph( &Driver, &Mode, "" ); /* Detection is overridden
                                                            */
   if( grOk != graphresult() ) exit( 1 );

   outtext( "User Installed Drivers Supported" );

   getchar();
   closegraph();

}
```

installuserfont

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

Syntax #include <graphics.h>
int far installuserfont(char far **name*);

Prototype in graphics.h

Remarks *name* is a path name to 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) will be returned.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **settextstyle**

int86

Function General 8086 software interrupt.

Syntax #include <dos.h>
int int86(int *intno*, union REGS **inregs*,
union REGS **outregs*);

Prototype in dos.h

Remarks

int86 executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

After the software interrupt returns, **int86** copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it usually indicates that an error has occurred.

Note that *inregs* can point to the same structure that *outregs* points to.

Return value

int86 returns the value of AX after completion of the software interrupt. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, this function sets *_doserrno* to the error code.

Portability

int86 is unique to the 8086 family of processors.

See also

bdos, bdosptr, geninterrupt, int86x, intdos, intdosx, intr

Example

```
#include <dos.h>

#define VIDEO 0x10

/* Positions cursor at line y, column x */
void gotoxy(int x, int y)
{
   union REGS regs;
   regs.h.ah = 2;                    /* set cursor position */
   regs.h.dh = y;
   regs.h.dl = x;
   regs.h.bh = 0;                           /* video page 0 */
   int86(VIDEO, &regs, &regs);
}
```

int86x

Function General 8086 software interrupt interface.

Syntax

```
#include <dos.h>
int int86x(int intno, union REGS *inregs,
           union REGS *outregs,
           struct SREGS *segregs);
```

Prototype in dos.h

Remarks **int86x** executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

In addition, **int86x** copies the *segregs -> x.ds* and *segregs -> x.es* values into the corresponding registers before executing the software interrupt. This feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the software interrupt.

After the software interrupt returns, **int86x** copies the current register values to *outregs*, the status of the carry flag to the *x.cflag* field in *outregs*, and the value of the 8086 flags register to the *x.flags* field in *outregs*. In addition, **int86x** restores DS and sets the *segregs -> es* and *segregs -> ds* fields to the values of the corresponding segment registers. If the carry flag is set, it usually indicates that an error has occurred.

int86x allows you to invoke an 8086 software interrupt that takes a value of DS different from the default data segment, and/or that takes an argument in ES.

Note that *inregs* can point to the same structure that *outregs* points to.

Return value **int86x** returns the value of AX after completion of the software interrupt. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, this function sets *_doserrno* to the error code.

Portability **int86x** is unique to the 8086 family of processors.

See also **bdos, bdosptr, geninterrupt, intdos, intdosx, int86, intr, segread**

---

intdos

Function General DOS interrupt interface.

Syntax

```
#include <dos.h>
int intdos(union REGS *inregs,
           union REGS *outregs);
```

Prototype in dos.h

Remarks **intdos** executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.al* specifies the DOS function to be invoked.

After the interrupt 0x21 returns, **intdos** copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it indicates that an error has occurred.

Note that *inregs* can point to the same structure that *outregs* points to.

Return value **intdos** returns the value of AX after completion of the DOS function call. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, it sets *_doserrno* to the error code.

Portability **intdos** is unique to DOS.

See also **bdos, geninterrupt, int86, int86x, intdosx, intr**

Example

```
#include <stdio.h>
#include <dos.h>

/* Deletes file name; returns 0 on success,
   nonzero error code on failure */
int delete_file(char near *filename)
{
   union REGS regs;
   int ret;
   regs.h.ah = 0x41; /* delete file */
   regs.x.dx = (unsigned) filename;
```

```
   ret = intdos(&regs, &regs);

   /* If carry flag is set, there was an error */
   return(regs.x.cflag ? ret : 0);
}

main()
{
   int err;
   err = delete_file("NOTEXIST.$$$");
   printf("Able to delete NOTEXIST.$$$: %s\n",
          (!err) ? "YES" : "NO");
}
```

Program output

```
Able to delete NOTEXIST.$$$: NO
```

intdosx

Function General DOS interrupt interface.

Syntax #include <dos.h>
int intdosx(union REGS **inregs*, union REGS **outregs*,
struct SREGS **segregs*);

Prototype in dos.h

Remarks **intdosx** executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.al* specifies the DOS function to be invoked.

In addition, **intdosx** copies the *segregs -> x.ds* and *segregs -> x.es* values into the corresponding registers before invoking the DOS function. This feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the function execution.

After the interrupt 0x21 returns, **intdosx** copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. In addition, **intdosx** sets the *segregs -> es* and *segregs -> ds* fields to the values of the corresponding segment registers and then restores DS. If the carry flag is set, it indicates that an error occurred.

intdosx allows you to invoke a DOS function that takes a value of DS different from the default data segment, and/or that takes an argument in ES.

Note that *inregs* can point to the same structure that *outregs* points to.

Return value

intdosx returns the value of AX after completion of the DOS function call. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, it sets *_doserrno* to the error code.

Portability

intdosx is unique to DOS.

See also

bdos, geninterrupt, int86, int86x, intdos, intr, segread

Example

```
#include <stdio.h>
#include <dos.h>

/* Deletes file name; returns 0 on success,
   nonzero error code on failure */
int delete_file(char far *filename)
{
   union REGS regs; struct SREGS sregs;
   int ret;
   regs.h.ah = 0x41; /* delete file */
   regs.x.dx = FP_OFF(filename);
   sregs.ds = FP_SEG(filename);
   ret = intdosx(&regs, &regs, &sregs);

   /* If carry flag is set, there was an error */
   return(regs.x.cflag ? ret : 0);
}

main()
{
   int err;
   err = delete_file("NOTEXIST.$$$");
   printf("Able to delete NOTEXIST.$$$: %s\n",
          (!err) ? "YES" : "NO");
}
```

Program output

```
Able to delete NOTEXIST.$$$: NO
```

intr

Function Alternate 8086 software interrupt interface.

Syntax #include <dos.h>
void *intr*(int *intno*, struct REGPACK **preg*);

Prototype in dos.h

Remarks The **intr** function is an alternate interface for executing software interrupts. It generates an 8086 software interrupt specified by the argument *intno*.

intr copies register values from the **REGPACK** structure **preg* into the registers before executing the software interrupt. After the software interrupt completes, **intr** copies the current register values into **preg*, including the flags.

The arguments passed to **intr** are as follows:

intno the interrupt number to be executed

preg the address of a structure containing

(a) the input registers before the call
(b) the value of the registers after the interrupt call

The **REGPACK** structure (defined in dos.h) has the following format:

```
struct  REGPACK
   {
   unsigned  r_ax, r_bx, r_cx, r_dx;
   unsigned  r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

Return value No value is returned. The **REGPACK** structure **preg* contains the value of the registers after the interrupt call.

Portability **intr** is unique to the 8086 family of processors.

See also **geninterrupt, int86, int86x, intdos, intdosx**

ioctl

Function Controls I/0 device.

Syntax int ioctl(int *handle*, int *func*
[, void *argdx, int argcx]);

Prototype in io.h

Remarks This is a direct interface to the DOS call 0x44 (IOCTL).

The exact function depends on the value of *func*, as follows:

- 0 Get device information.
- 1 Set device information (in *argdx*).
- 2 Read *argcx* bytes into the address pointed to by *argdx*.
- 3 Write *argcx* bytes from the address pointed to by *argdx*.
- 4 Same as 2 except *handle* is treated as a drive number (0 equals default, 1 equals A, and so on).
- 5 Same as 3 except *handle* is a drive number (0 equals default, 1 equals A, and so on).
- 6 Get input status.
- 7 Get output status.
- 8 Test removability; DOS 3.0 only.
- 11 Set sharing conflict retry count; DOS 3.0 only.

ioctl can be used to get information about device channels.
Regular files can also be used, but only *func* values 0, 6, and 7 are defined for them. All other calls return an EINVAL error for files.

See the documentation for system call 0x44 in the *MS-DOS Programmer's Reference Manual* for detailed information on argument or return values.

The arguments *argdx* and *argcx* are optional.

ioctl provides a direct interface to DOS device drivers for special functions. As a result, the exact behavior of this function will vary across different vendors' hardware and in different devices. Also, several vendors do

not follow the interfaces described here. Refer to the vendor BIOS documentation for exact use of **ioctl**.

Return value

For *func* 0 or 1, the return value is the device information (DX of the IOCTL call).

For *func* values of 2 through 5, the return value is the number of bytes actually transferred.

For *func* values of 6 or 7, the return value is the device status.

In any event, if an error is detected, a value of –1 is returned, and *errno* is set to one of the following:

EINVAL	Invalid argument
EBADF	Bad file number
EINVDAT	Invalid data

Portability

ioctl is available on UNIX systems, but not with these parameters or functionality. UNIX version 7 and System III differ from each other in their use of **ioctl**. **ioctl** calls are not portable to UNIX and are rarely portable across DOS machines.

DOS 3.0 extends **ioctl** with *func* values of 8 and 11.

Example

```
#include <stdio.h>
#include <io.h>
#include <dir.h>

main()
{
   int stat;

   /* Use function 8 to determine if the default
      drive is removable */
   stat = ioctl(0, 8, 0, 0);
   printf("Drive %c %s changeable\n", getdisk() + 'A',
          (stat == 0) ? "is" : "is not");
}
```

Program output

```
Drive C is not changeable
```

isalnum

Function Character classification macro.

Syntax
```
#include <ctype.h>
int isalnum(int c);
```

Prototype in ctype.h

Remarks **isalnum** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isalnum** returns nonzero if *c* is a letter (*A*-*Z* or *a*-*z*) or a digit (0-9).

Portability **isalnum** is available on UNIX machines.

isalpha

Function Character classification macro.

Syntax
```
#include <ctype.h>
int isalpha(int c);
```

Prototype in ctype.h

Remarks **isalpha** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isalpha** returns nonzero if *c* is a letter (*A*-*Z* or *a*-*z*).

Portability **isalpha** is available on UNIX machines and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

isascii

Function Character classification macro.

Syntax
```
#include <ctype.h>
int isascii(int c);
```

Prototype in ctype.h

Remarks **isascii** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.

isascii is defined on all integer values.

Return value **isascii** returns nonzero if the low order byte of *c* is in the range 0-127 (0x00-0x7F).

Portability **isascii** is available on UNIX machines.

isatty

Function Checks for device type.

Syntax int isatty(int *handle*);

Prototype in io.h

Remarks **isatty** determines whether *handle* is associated with any one of the following character devices:

- a terminal
- a console
- a printer
- a serial port

Return value If the device is a character device **isatty** returns a nonzero integer. If it is not such a device, **isatty** returns 0.

iscntrl

Function Character classification macro.

Syntax

```
#include <ctype.h>
int iscntrl(int c);
```

Prototype in ctype.h

Remarks **iscntrl** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **iscntrl** returns nonzero if *c* is a delete character or ordinary control character (0x7F or 0x00-0x1F).

Portability **iscntrl** is available on UNIX machines and is compatible with ANSI C.

isdigit

Function Character classification macro.

Syntax

```
#include <ctype.h>
int isdigit(int c);
```

Prototype in ctype.h

Remarks **isdigit** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isdigit** returns nonzero if *c* is a digit ('0'-'9').

Portability **isdigit** is available on UNIX machines and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

isgraph

Function Character classification macro.

Syntax #include <ctype.h>
int isgraph(int *c*);

Prototype in ctype.h

Remarks **isgraph** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isgraph** returns nonzero if *c* is a printing character, like **isprint**, except that a space character is excluded.

Portability **isgraph** is available on UNIX machines and is compatible with ANSI C.

islower

Function Character classification macro.

Syntax #include <ctype.h>
int islower(int *c*);

Prototype in ctype.h

Remarks **islower** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **islower** returns nonzero if *c* is a lowercase letter (*a*-*z*).

Portability **islower** is available on UNIX machines and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

isprint

Function Character classification macro.

Syntax

```
#include <ctype.h>
int isprint(int c);
```

Prototype in ctype.h

Remarks **isprint** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isprint** returns nonzero if *c* is a printing character (0x20 – 0x7E).

Portability **isprint** is available on UNIX machines and is compatible with ANSI C.

ispunct

Function Character classification macro.

Syntax

```
#include <ctype.h>
int ispunct(int c);
```

Prototype in ctype.h

Remarks **ispunct** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **ispunct** returns nonzero if *c* is a punctuation character (**iscntrl** or **isspace**).

Portability **ispunct** is available on UNIX machines and is compatible with ANSI C.

isspace

Function Character classification macro.

Syntax

```
#include <ctype.h>
int isspace(int c);
```

Prototype in ctype.h

Remarks **isspace** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isspace** returns nonzero if *c* is a space, tab, carriage return, newline, vertical tab, or formfeed (0x09-0x0D, 0x20).

Portability **isspace** is available on UNIX machines and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

isupper

Function Character classification macro.

Syntax

```
#include <ctype.h>
int isupper(int c);
```

Prototype in ctype.h

Remarks **isupper** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isupper** returns nonzero if *c* is an uppercase letter (*A*-*Z*).

Portability **isupper** is available on UNIX machines and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

isxdigit

Function Character classification macro.

Syntax

```
#include <ctype.h>
int isxdigit(int c);
```

Prototype in ctype.h

Remarks **isxdigit** is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false. It is defined only when **isascii**(*c*) is true or *c* is EOF.

Return value **isxdigit** returns nonzero if *c* is a hexadecimal digit (0-9, *A-F*, *a-f*).

Portability **isxdigit** is available on UNIX machines and is compatible with ANSI C.

itoa

Function Converts an integer to a string.

Syntax

```
char *itoa(int value, char *string, int radix);
```

Prototype in stdlib.h

Remarks This function converts *value* to a null-terminated string and stores the result in *string*. With **itoa**, *value* is an integer.

radix specifies the base to be used in converting *value;* it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (–).

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). **itoa** can return up to 17 bytes.

Return value **itoa** returns a pointer to *string*. There is no error return.

See also **ltoa, ultoa**

kbhit

Function Checks for currently available keystrokes.

Syntax int kbhit(void);

Prototype in conio.h

Remarks **kbhit** checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with **getch** or **getche**.

Return value If a keystroke is available, **kbhit** returns a nonzero value. If not, it returns 0.

See also **getch, getche**

keep

Function Exits and remains resident.

Syntax void keep(unsigned char *status*, unsigned *size*);

Prototype in dos.h

Remarks **keep** returns 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 can be used when installing a TSR program. **keep** uses DOS function 0x31.

Return value None.

Portability **keep** is unique to DOS.

See also **abort, exit**

labs

Function Gives long absolute value.

Syntax
```
#include <math.h>
long int labs(long int x);
```

Prototype in math.h, stdlib.h

Remarks **labs** computes the absolute value of the parameter *x*.

Return value On success, **labs** returns the absolute value of *x*. There is no error return.

Portability **labs** is available on UNIX systems and is compatible with ANSI C.

See also **abs, cabs, fabs**

ldexp

Function Calculates $x \times 2^{exp}$.

Syntax
```
#include <math.h>
double ldexp(double x, int exp);
```

Prototype in math.h

Remarks **ldexp** calculates the **double** value $x \times 2^{exp}$.

Return value On success, **ldexp** returns the value it calculated, $x \times 2^{exp}$.

Error-handling for **ldexp** can be modified through the function **matherr.**

Portability **ldexp** is available on UNIX systems and is compatible with ANSI C.

See also **exp, frexp, modf**

ldiv

Function Divides two longs, returns quotient and remainder.

Syntax #include <stdlib.h>
ldiv_t ldiv(long int *numer*, long int *denom*);

Prototype in stdlib.h

Remarks **ldiv** divides two longs and returns both the quotient and the remainder as an *ldiv_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *ldiv_t* type is a structure of longs defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
   long int quot;                        /* quotient */
   long int rem;                        /* remainder */
} ldiv_t;
```

Return value **ldiv** returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

Portability **ldiv** is compatible with ANSI C.

See also **div**

Example

```
#include <stdlib.h>
ldiv_t lx;

main()
{
   lx = ldiv(100000L, 30000L);
   printf("100000 div 30000 = %ld remainder %ld\n",
          lx.quot, lx.rem);
}
```

lfind

Function Performs a linear search.

Syntax

```
#include <stdlib.h>
void *lfind(const void *key, const void *base,
            size_t *num, size_t width,
            int (*fcmp)(const void *, const void *));
```

Prototype in stdlib.h

Remarks **lfind** makes a linear search for the value of *key* in an array of sequential records. It uses a user-defined comparison routine (*fcmp*).

The array is described as having **num* records that are *width* bytes wide, and begins at the memory location pointed to by *base*.

Return value **lfind** returns the address of the first entry in the table that matches the search key. If no match is found, **lfind** returns NULL. The comparison routine must return 0 if **elem1* == **elem2*, and nonzero otherwise (*elem1* and *elem2* are its two parameters).

See also **bsearch, lsearch**

line

Function Draws a line between two specified points.

Syntax

```
#include <graphics.h>
void far line(int x1, int y1, int x2, int y2);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **linerel, lineto, setcolor, setlinestyle, setwritemode**

linerel

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

Syntax
```
#include <graphics.h>
void far linerel(int dx, int dy);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **line, lineto, setcolor, setlinestyle, setwritemode**

lineto

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

Syntax
```
#include <graphics.h>
void far lineto(int x, int y);
```

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **line, linerel, setcolor, setlinestyle, setvisualpage, setwritemode**

localtime

Function

Converts date and time to a structure.

Syntax

#include <time.h>
struct tm *localtime(const time_t **timer*);

Prototype in

time.h

Remarks

localtime accepts the address of a value returned by **time** and returns a pointer to the structure of type **tm** containing the broken-down time. It corrects for the time zone and possible daylight savings time.

The global long variable *timezone* should be set to the difference in seconds between GMT and local standard time (in PST, *timezone* is $8 \times 60 \times 60$). The global variable *daylight* should be set to nonzero *only if* the standard U.S. Daylight Savings time conversion should be applied.

The **tm** structure declaration from the time.h include file follows:

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month (0-11), weekday (Sunday equals 0), year – 1900, day of year (0-365), and a flag that is nonzero if daylight savings time is in effect.

Return value

localtime returns a pointer to the structure containing the broken-down time. This structure is a static that is overwritten with each call.

Portability

localtime is available on UNIX systems, and it is compatible with ANSI C.

See also

asctime, ctime, ftime, gmtime, stime, time, tzset

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
   struct tm         *timeptr;
   time_t            secsnow;

   timezone = 8 * 60 * 60;
   time(&secsnow);
   timeptr = localtime(&secsnow);
   printf("The date is %d-%d-19%02d\n",
          ((timeptr -> tm_mon) + 1), timeptr -> tm_mday,
          timeptr -> tm_year);
   printf("Local time is %02d:%02d:%02d\n",
          timeptr -> tm_hour, timeptr -> tm_min,
          timeptr -> tm_sec);
}
```

Program output

```
The date is 2-2-88
Local time is 12:44:36
```

lock

Function Sets file-sharing locks.

Syntax int lock(int *handle*, long *offset*, long *length*);

Prototype in io.h

Remarks **lock** provides an interface to the DOS 3.x file-sharing mechanism.

lock can be placed on arbitrary, non-overlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

Return value **lock** returns 0 on success, –1 on error.

Portability **lock** is unique to DOS 3.x. Older versions of DOS do not support it.

See also **open, sopen, unlock**

log

Function Calculates the natural logarithm of *x*.

Syntax
```
#include <math.h>
double log(double x);
```

Prototype in math.h

Remarks **log** calculates the natural logarithm of *x*.

Return value On success, **log** returns the value calculated, *ln*(*x*).

If the argument *x* passed to **log** is less than or equal to 0, *errno* is set to

EDOM Domain error

When this error occurs, **log** returns the value negative HUGE_VAL.

Error-handling for **log** can be modified through the function **matherr**.

Portability **log** is available on UNIX systems and is compatible with ANSI C.

See also **exp, log10, sqrt**

log10

Function Calculates $\log_{10}(x)$.

Syntax
```
#include <math.h>
double log10(double x);
```

Prototype in math.h

Remarks **log10** calculates the base 10 logarithm of *x*.

Return value On success, **log10** returns the value calculated, $log_{10}(x)$.

If the argument *x* passed to **log10** is less than or equal to 0, *errno* is set to

EDOM Domain error

When this error occurs, **log10** returns the value negative HUGE_VAL.

Error-handling for **log10** can be modified through the function **matherr**.

Portability **log10** is available on UNIX systems and is compatible with ANSI C.

See also **exp, log**

longjmp

Function Performs nonlocal goto.

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf jmpb, int retval);
```

Prototype in setjmp.h

Remarks A call to **longjmp** restores the task state captured by the last call to **setjump** with the argument *jmpb*. It then returns in such a way that **setjmp** appears to have returned with the value *retval*.

A task state is

- all segment registers (CS, DS, ES, SS)
- register variables (SI, DI)
- stack pointer (SP)
- frame base pointer (BP)
- flags

A task state is complete enough that **setjmp** and **longjmp** can be used to implement co-routines. These routines are useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

setjmp must be called before **longjmp**. The routine that called **setjmp** and set up *jmpb* must still be active and cannot have returned before the **longjmp** is called. If this happens, the results are unpredictable.

longjmp cannot pass the value 0; if 0 is passed in *retval*, **longjmp** will substitute 1.

Return value None.

Portability **longjmp** is available on UNIX systems and is compatible with ANSI C.

See also **setjmp, signal**

Example

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jumper;

main()
{
   int value;

   value = setjmp(jumper);
   if (value != 0)
   {
      printf("Longjmp with value %d\n", value);
      exit(value);
   }
   printf("About to call subroutine ... \n");
   subroutine();
}

subroutine()
{
   longjmp(jumper,1);
}
```

Program output

```
About to call subroutine ...
Longjmp with value 1
```

lowvideo

Function Selects low-intensity characters.

Syntax void lowvideo(void);

Prototype in conio.h

Remarks **lowvideo** selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently on the screen, only those displayed by functions that

perform text mode, direct console output *after* this function is called.

Return value None.

Portability **lowvideo** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **highvideo, normvideo, textattr, textcolor**

_lrotl

Function Rotates an unsigned long integer value to the left.

Syntax unsigned long _lrotl(unsigned long *val*, int *count*);

Prototype in stdlib.h

Remarks **_lrotl** rotates the given *val* to the left *count* bits; *val* is an **unsigned long.**

Return value **_lrotl** returns the value of *val* left-rotated *count* bits.

See also **_rotr**

_lrotr

Function Rotates an unsigned long integer value to the right.

Syntax unsigned long _lrotr(unsigned long *val*, int *count*);

Prototype in stdlib.h

Remarks **_lrotr** rotates the given *val* to the right *count* bits; *val* is an **unsigned long.**

Return value **_lrotr** returns the value of *val* right-rotated *count* bits.

See also **_rotl**

lsearch

Function Performs a linear search.

Syntax

```
#include <stdlib.h>
void *lsearch(const void *key, void *base,
              size_t *num, size_t width,
              int (*fcmp)(const void *,
              const void *));
```

Prototype in stdlib.h

Remarks

lsearch searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to **lsearch**. If the item that *key* points to is not in the table, **lsearch** appends that item to the table.

- *base* points to the base (0th element) of the search table.
- *num* points to an integer containing the number of entries in the table.
- *width* contains the number of bytes in each entry.
- *key* points to the item to be searched for (the *search key*).

The argument *fcmp* points to a user-written comparison routine, which compares two items and returns a value based on the comparison.

To search the table, **lsearch** makes repeated calls to the routine whose address is passed in *fcmp*.

On each call to the comparison routine, **lsearch** passes two arguments: *key*, a pointer to the item being searched for; and *elem*, a pointer to the element of *base* being compared.

fcmp is free to interpret the search key and the table entries in any way.

Return value

lsearch returns the address of the first entry in the table that matches the search key.

If the search key is not identical to **elem*, *fcmp* returns a nonzero integer. If the search key is identical to **elem*, *fcmp* returns 0.

Portability **lsearch** is available on UNIX systems.

See also **bsearch, lfind, qsort**

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>                /* for strcmp declaration */

/* Initialize number of colors */
char *colors[10] = { "Red", "Blue", "Green" };
int ncolors = 3;

int colorscmp(char **arg1, char **arg2)
{
   return(strcmp(*arg1, *arg2));
}

int addelem(char *color)
{
   int oldn = ncolors;
   lsearch(&color, colors, (size_t *)&colors,
           sizeof(char *), colorscmp);
   return(ncolors == oldn);
}

main()
{
   int i;
   char *key = "Purple";

   if (addelem(key))
      printf("%s already in colors table\n", key);
   else
      printf("%s added to colors table,"
             "now %d colors\n", key, ncolors);
   printf("The colors:\n");
   for (i = 0; i < ncolors; i++)
      printf("%s\n", colors[i]);
}
```

Program output

```
Purple added to colors table,
now 4 colors
```

lseek

Function Moves file pointer.

Syntax

```
#include <io.h>
long lseek(int handle, long offset, int fromwhere);
```

Prototype in io.h

Remarks **lseek** sets the file pointer associated with *handle* to a new position *offset* bytes beyond the file location given by *fromwhere*. It is a good idea to set *fromwhere* using one of three symbolic constants (defined in io.h) instead of a specific number. The constants are as follows:

fromwhere		**File Location**
SEEK_SET	(0)	file beginning
SEEK_CUR	(1)	current file pointer position
SEEK_END	(2)	end-of-file

Return value **lseek** returns the offset of the pointer's new position, measured in bytes from the file beginning. **lseek** returns –1L on error, and *errno* is set to one of the following:

EBADF	Bad file number
EINVAL	Invalid argument

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

Portability **lseek** is available on all UNIX systems.

See also **filelength, fseek, ftell, sopen, _write, write**

ltoa

Function Converts a long to a string.

Syntax

```
#include <stdlib.h>
char *ltoa(long value, char *string, int radix);
```

Prototype in stdlib.h

Remarks

ltoa converts *value* to a null-terminated string and stores the result in *string*. *value* is a long integer.

radix specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (–).

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). **ltoa** can return up to 33 bytes.

Return value

ltoa returns a pointer to *string*. There is no error return.

See also

itoa, ultoa

---

malloc

Function

Allocates main memory.

Syntax

#include <stdlib.h> or #include<alloc.h>
void *malloc(size_t *size*);

Prototype in

stdlib.h, alloc.h

Remarks

malloc allocates a block of *size* bytes from the C memory heap. It allows a program to allocate memory explicitly, as it is needed and in the exact amounts needed.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures such as trees and lists naturally employ heap memory allocation.

All the space between the end of the data segment and the top of the program stack is available for use in the small data models, except for a 256-byte margin immediately before the top of the stack. This margin is intended to allow the application some room to grow the stack, in addition to a small amount needed by DOS.

In the large data models, all the space beyond the program stack to the end of physical memory is available for the heap.

Return value

On success, **malloc** returns a pointer to the newly allocated block of memory. If not enough space exists for the new block, it returns null. The contents of the block are left unchanged. If the argument *size* == 0, **malloc** returns null.

Portability

malloc is available on UNIX systems and is compatible with ANSI C.

See also

allocmem, calloc, coreleft, farcalloc, farmalloc, free, realloc

Example

```
#include <stdio.h>
#include <stdlib.h>

typedef struct  {
  /* ... */
} OBJECT;

OBJECT *NewObject()
{
   return ((OBJECT *) malloc(sizeof(OBJECT)));
}

void FreeObject(OBJECT *obj)
{
   free(obj);
}

main()
{
   OBJECT *obj;
   obj = NewObject();
   if (obj == NULL) {
      printf("failed to create a new object\n");
      exit(1);
   }
   /* ... */
   FreeObject(obj);
}
```

_matherr

Function Floating-point error handling.

Syntax

```
#include <math.h>
double _matherr(_mexcep why, char *fun,
                double *arg1p, double *arg2p,
                double retval);
```

Prototype in math.h

Remarks **_matherr** serves as a focal point for error-handling in all math library functions; it calls **matherr** and processes the return value from **matherr**. **_matherr** should never be called directly by user programs. Instead, the math library error-handling can be customized by replacing the library **matherr**.

Whenever an error occurs in one of the math library routines, **_matherr** is called with several arguments. **_matherr** does four things:

- It uses its arguments to fill out an **exception** structure.
- It calls **matherr** with *e*, a pointer to the **exception** structure, to
  see if **matherr** can resolve the error.
- It examines the return value from **matherr** as follows:

  If **matherr** returns 0 (indicating that **matherr** was not able to resolve the error), **_matherr** sets *errno* and prints an error message.

  If **matherr** returns nonzero (indicating that **matherr** was able to resolve the error), **_matherr** is silent; it does not set *errno* or print any messages.
- It returns *e -> retval* to the original caller. Note that **matherr** might modify *e -> retval* to specify the value it wants propagated back to the original caller.

When **_matherr** sets *errno* (based on a 0 return from **matherr**), it maps the kind of error that occurred (from the *type* field in the **exception** structure) onto an *errno* value of either EDOM or ERANGE.

Return value **_matherr** returns the value *e* -> *retval*. This value is initially the value of the input parameter *retval* passed to **_matherr** and might be modified by **matherr**.

For math function results with a magnitude greater than MAXDOUBLE, *retval* defaults to the macro HUGE_VAL of appropriate sign before being passed to **_matherr**. For math function results with a magnitude less than MINDOUBLE, *retval* is set to 0, then passed to **_matherr**. In both of these extremes, if **matherr** does not modify *e* -> *retval*, **_matherr** sets *errno* to

ERANGE   Result out of range

See also **matherr**

matherr

Function User-modifiable math error handler.

Syntax

```
#include &lt;math.h&gt;
int matherr(struct exception *e);
```

Prototype in math.h

Remarks **matherr** is called by the **_matherr** routine to handle errors generated by the math library.

matherr serves as a user hook (a function that can be customized by the user) that you can replace by writing your own math error-handling routine—see the following example of a user-defined **matherr** implementation.

matherr is useful for trapping domain and range errors caused by the math functions. It does not trap floating-point exceptions such as division by zero. See **signal** for trapping such errors.

You can define your own **matherr** routine to be a custom error-handler (such as one that catches and resolves certain types of errors); this customized function will override the default version in the C library. The customized **matherr** should return 0 if it fails to resolve the error, or nonzero if the error is resolved. When

matherr returns nonzero, no error message is printed, and *errno* is not changed.

This is the **exception** structure (defined in math.h):

```
struct exception {
   int   type;
   char  *Function;
   double arg1, arg2, retval;
};
```

The members of the **exception** structure are shown in the following table.

Member	What It Is (or Represents)
type	The type of mathematical error that occurred; an **enum** type defined in the **typedef** *_mexcep* (see definition after this list).
name	A pointer to a null-terminated string holding the *name* of the math library function that resulted in an error.
arg1, *arg2*	The arguments (passed to the function *name* points to) that caused the error; if only one argument was passed to the function, it is stored in *arg1*.
retval	The default return value for **matherr**; you can modify this value.

The **typedef** *_mexcep*, also defined in math.h, enumerates the following symbolic constants representing possible mathematical errors:

Symbolic Constant	Mathematical Error
DOMAIN	Argument was not in domain of function (such as *log*(-1)).
SING	Argument would result in a singularity (such as *pow*(0, –2)).
OVERFLOW	Argument would produce a function result greater than MAXDOUBLE (such as *exp*(1000)).
UNDERFLOW	Argument would produce a function result less than MINDOUBLE (such as *exp*(-1000)).
TLOSS	Argument would produce function result with total loss of significant digits (such as *sin*(10*e*70)).

The symbolic constants MAXDOUBLE and MINDOUBLE are defined in values.h.

The source code to the default **matherr** is on the Turbo C distribution disks.

Note that **_matherr** is not meant to be modified. The **matherr** function is more widely found in C run-time libraries and thus is recommended for portable programming.

The UNIX-style **matherr** default behavior (printing a message and terminating) is not ANSI compatible. If you desire a UNIX-style version of **matherr**, use MATHERR.C provided on the Turbo C distribution disks.

Return value

The default return value for **matherr** is 1 if the error is UNDERFLOW or TLOSS, 0 otherwise. **matherr** can also modify *e -> retval*, which propagates through **_matherr** back to the original caller.

When **matherr** returns 0 (indicating that it was not able to resolve the error), **_matherr** sets *errno* and prints an error message. (See **_matherr** for details.)

When **matherr** returns nonzero (indicating that it was able to resolve the error), *errno* is not set and no messages are printed.

Portability

matherr is available on many C compilers, but it is not compatible with ANSI C. A UNIX-style **matherr** that prints a message and terminates is provided in MATHERR.C on the Turbo C distribution disks.

See also

_matherr

Example

```
/* This is a user-defined matherr function that
   catches negative arguments passed to sqrt and
   converts them to nonnegative values before sqrt
   processes them. */

#include<math.h>
#include<string.h>

int matherr(struct exception *a)
{
   if (a -> type == DOMAIN)
   {
      if(strcmp(a -> name, "sqrt") == 0)
      {
         a -> retval = sqrt (-(a -> arg1));
         return (1);
      }
   }
   return (0);
}
```

---

max

Function

Returns the larger of two values.

Syntax

#include <stdlib.h>
(type) max(*a*, *b*);

Prototype in

stdlib.h

Remarks

This macro compares two values and returns the larger of the two. Both arguments and the function declaration must be of the same type.

Return value

max returns the larger of two values.

Example

```
#include <stdlib.h>
main()
{
   int x = 5;
   int y = 6;
   int z;
   z = (int)max(x, y);
   printf("The larger number is %d\n", z);
}
```

Program output

```
The larger number is 6
```

memccpy

Function Copies a block of *n* bytes.

Syntax #include <mem.h>
void *memccpy(void **dest*, const void **src*,
int *c*, size_t *n*);

Prototype in string.h, mem.h

Remarks **memccpy** copies a block of *n* bytes from *src* to *dest*. The copying stops as soon as either of the following occurs:

- The character *c* is first copied into *dest*.
- *n* bytes have been copied into *dest*.

Return value **memccpy** returns a pointer to the byte in *dest* immediately following *c*, if *c* was copied; otherwise, **memccpy** returns null.

Portability **memccpy** is available on UNIX System V systems.

See also **memcpy, memmove, memset**

memchr

Function Searches *n* bytes for character *c*.

Syntax

```
#include <mem.h>
void *memchr(const void *s, int c, size_t n);
```

Prototype in string.h, mem.h

Remarks **memchr** searches the first *n* bytes of the block pointed to by *s* for character *c*.

Return value On success, **memchr** returns a pointer to the first occurrence of *c* in *s*; otherwise, it returns null.

Portability **memchr** is available on UNIX System V systems and is compatible with ANSI C.

memcmp

Function Compares two blocks for a length of exactly *n* bytes.

Syntax

```
#include <mem.h>
int memcmp(const void *s1,
           const void *s2, size_t n);
```

Prototype in string.h, mem.h

Remarks **memcmp** compares the first *n* bytes of the blocks *s1* and *s2*, as **unsigned chars**.

Return value **memcmp** returns a value

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

Since it compares bytes as **unsigned chars**, for example,

```
memcmp("\xFF", "\x7F", 1)
```

returns a value than 0.

Portability **memcmp** is available on UNIX System V systems and is compatible with ANSI C.

See also **memicmp**

memcpy

Function Copies a block of *n* bytes.

Syntax

```
#include &lt;mem.h&gt;
void *memcpy(void *dest,
             const void *src, size_t n);
```

Prototype in string.h, mem.h

Remarks **memcpy** copies a block of *n* bytes from *src* to *dest*. If *src* and *dest* overlap, the behavior of **memcpy** is undefined.

Return value **memcpy** returns *dest*.

Portability **memcpy** is available on UNIX System V systems and is compatible with ANSI C.

See also **memccpy, memmove, memset, movedata, movmem**

memicmp

Function Compares *n* bytes of two character arrays, ignoring case.

Syntax

```
#include <mem.h>
int memicmp(const void *s1,
            const void *s2, size_t n);
```

Prototype in string.h, mem.h

Remarks **memicmp** compares the first *n* bytes of the blocks *s1* and *s2*, ignoring character case (upper or lower).

Return value **memicmp** returns a value

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

Portability **memicmp** is available on UNIX System V systems.

See also **memcmp**

memmove

Function Copies a block of *n* bytes.

Syntax

```
#include <mem.h>
void *memmove(void *dest, const void *src,
              size_t n);
```

Prototype in string.h, mem.h

Remarks **memmove** copies a block of *n* bytes from *src* to *dest*. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly.

Return value **memmove** returns *dest*.

Portability **memmove** is available on UNIX System V systems and is compatible with ANSI C.

See also **memccpy, memcpy, movmem**

memset

Function Sets *n* bytes of block of memory to byte *c*.

Syntax

```
#include <mem.h>
void *memset(void *s, int c, size_t n);
```

Prototype in string.h, mem.h

Remarks **memset** sets the first *n* bytes of the array *s* to the character *c*.

Return value **memset** returns *s*.

Portability **memset** is available on UNIX System V systems and is compatible with ANSI C.

See also **memccpy, memcpy, setmem**

min

Function Returns the smaller of two values.

Syntax #include <stdlib.h>
(type) min(*a*, *b*);

Prototype in stdlib.h

Remarks This macro compares two values and returns the smaller of the two. Both arguments and the function declaration must be of the same type.

Return value **min** returns the smaller of two values.

See also **max**

Example

```
#include <stdlib.h>
main()
{
   int x = 5;
   int y = 6;
   int z;
   z = (int)min(x, y);
   printf("The smaller number is %d\n", z);
}
```

Program output

```
The smaller number is 5
```

mkdir

Function Creates a directory.

Syntax int mkdir(const char **path*);

Prototype in dir.h

Remarks **mkdir** creates a new directory from the given path name *path*.

Return value **mkdir** returns the value 0 if the new directory was created.

A return value of –1 indicates an error, and *errno* is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

See also **chdir, getcurdir, getcwd, rmdir**

MK_FP

Function Makes a far pointer.

Syntax
```
#include <dos.h>
void far * MK_FP(unsigned seg, unsigned ofs);
```

Prototype in dos.h

Remarks **MK_FP** is a macro that makes a far pointer from its component segment (*seg*) and offset (*ofs*) parts.

Return value **MK_FP** returns a far pointer.

See also **FP_OFF, FP_SEG, movedata, segread**

Example See **FP_OFF**

mktemp

Function Makes a unique file name.

Syntax
```
char *mktemp(char *template);
```

Prototype in dir.h

Remarks **mktemp** replaces the string pointed to by *template* with a unique file name and returns *template*.

The *template* should be a null-terminated string with six trailing *X*'s. These *X*'s are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.

Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same format.

Return value If *template* is well-formed, **mktemp** returns the address of the *template* string. Otherwise, it returns null.

Portability **mktemp** is available on UNIX systems.

modf

Function Splits into integer and fraction parts.

Syntax

```
#include <math.h>
double modf(double x, double *ipart);
```

Prototype in math.h

Remarks **modf** breaks the double *x* into two parts: the integer and the fraction. It stores the integer in *ipart* and returns the fraction.

Return value **modf** returns the fractional part of *x*.

See also **fmod, ldexp**

movedata

Function Copies *n* bytes.

Syntax

```
void movedata(unsigned srcseg,
              unsigned srcoff, unsigned dstseg,
              unsigned dstoff, size_t n);
```

Prototype in mem.h, string.h

Remarks **movedata** copies *n* bytes from the source address (*srcseg:srcoff*) to the destination address (*dstseg:dstoff*).

movedata is a means of moving blocks of data that is independent of memory model.

Return value None.

See also **FP_OFF, memcpy, MK_FP, movmem, segread**

Example

```
#include <mem.h>

#define MONO_BASE 0xB000

/* Saves the contents of the monochrome screen in buffer */
void save_mono_screen(char near *buffer)
{
   movedata(MONO_BASE, 0, _DS, (unsigned)buffer, 80*25*2);
```

```
}

main()
{
   char buf[80*25*2];
   save_mono_screen(buf);
}
```

moverel

Function Moves the current position (CP) a relative distance.

Syntax #include <graphics.h>
void far moverel(int *dx*, int *dy*);

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **moveto**

movetext

Function Copies text onscreen from one rectangle to another.

Syntax int movetext(int *left*, int *top*,
int *right*, int *bottom*,
int *destleft*, int *desttop*);

Prototype in conio.h

Remarks **movetext** copies the contents of the onscreen rectangle defined by *left*, *top*, *right*, and *bottom* to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (*destleft*, *desttop*).

All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly.

movetext is a text mode function performing direct video output.

Return value **movetext** returns nonzero if the operation succeeded. If the operation failed (for example, if you gave coordinates outside the range of the current screen mode), **movetext** returns 0.

Portability **movetext** can be used on IBM PCs and BIOS-compatible systems.

See also **gettext, puttext**

Example

```
/* Copy the contents of the old rectangle, whose
   upper left corner is (5, 15) and whose lower right
   corner is (20, 25), to a new rectangle whose upper
   left corner is (10, 20). */

movetext(5, 15, 20, 25, 10, 20);
```

---

moveto

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

Syntax #include <graphics.h>
void far moveto(int *x*, int *y*);

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **moverel**

---

movmem

Function Copies a block of *length* bytes.

Syntax void movmem(void *_src_, void *_dest_, unsigned _length_);

Prototype in mem.h

Remarks **movmem** copies a block of *length* bytes from *src* to *dest*. Even if the source and destination blocks overlap, the copy direction is chosen so that the data is always copied correctly.

Return value None.

See also **memcpy, memmove, movedata**

normvideo

Function Selects normal-intensity characters.

Syntax void normvideo(void);

Prototype in conio.h

Remarks **normvideo** selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

This function does not affect any characters currently on the screen, only those displayed by functions (such as **cprintf**) performing direct console output functions *after* **normvideo** is called.

Return value None.

Portability **normvideo** works with IBM PCs and compatibles only; a corresponding function exists in Turbo Pascal.

See also **highvideo, lowvideo, textattr, textcolor**

nosound

Function Turns PC speaker off.

Syntax void nosound(void);

Prototype in dos.h

Remarks Turns the speaker off after it has been turned on by a call to **sound**.

Return value None.

See also **delay, sound**

_open

Function Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>
int _open(const char *filename, int oflags);
```

Prototype in io.h

Remarks **_open** opens the file specified by *filename*, then prepares it for reading and/or writing as determined by the value of *oflags*. The file is opened in the mode specified by *_fmode*.

For **_open**, the value of *oflags* in DOS 2.x is limited to O_RDONLY, O_WRONLY, and O_RDWR. For DOS 3.x, the following additional values can also be used:

- O_NOINHERIT is included if the file is not to be passed to child programs.
- O_DENYALL allows only the current handle to have access to the file.
- O_DENYWRITE allows only reads from any other open to the file.
- O_DENYREAD allows only writes from any other open to the file.
- O_DENYNONE allows other shared opens to the file.

These O_... symbolic constants are defined in fcntl.h.

Only one of the O_DENY*xxx* values can be included in a single **_open** under DOS 3.x. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is a system configuration parameter.

Return value On successful completion, **_open** returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of

the file. On error, **_open** returns –1 and *errno* is set to one of the following:

ENOENT	Path or file not found
EMFILE	Too many open files
EACCES	Permission denied
EINVACC	Invalid access code

Portability **_open** is unique to DOS.

See also **open, _read, sopen**

open

Function Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>
#include<sys\stat.h>
int open(const char *path, int access
        [, unsigned mode]);
```

Prototype in io.h

Remarks **open** opens the file specified by *path*, then prepares it for reading and/or writing as determined by the value of *access*.

To create a file in a particular mode, you can either assign to *_fmode* or call **open** with the O_CREAT and O_TRUNC options ORed with the translation mode desired. For example, the call

```
open("xmp",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

will create a binary-mode, read-only file named XMP, truncating its length to 0 bytes if it already existed.

For **open**, *access* is constructed by bitwise ORing flags from the following two lists. Only one flag from the first list can be used; the remaining flags can be used in any logical combination.

List 1: Read/Write Flags

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

List 2: Other Access Flags

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits, as in **chmod**.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	This flag can be given to explicitly open the file in binary mode.
O_TEXT	This flag can be given to explicitly open the file in text mode.

These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing *access*, you need to supply the *mode* argument to **open**, from the following symbolic constants defined in sys\stat.h.

Value of *mode*	Access Permission
S_IWRITE	permission to write
S_IREAD	permission to read
S_IREAD | S_IWRITE	permission to read and write

Return value

On successful completion, **open** returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file. On error, **open** returns –1 and *errno* is set to one of the following:

ENOENT	No such file or directory
EMFILE	Too many open files
EACCES	Permission denied
EINVACC	Invalid access code

Portability **open** is available on UNIX systems. On UNIX version 7, the O_type mnemonics are not defined. UNIX System III uses all of the O_type mnemonics except O_BINARY.

See also **chmod, chsize, close, creat, creatnew, creattemp, dup, dup2, fdopen, filelength, fopen, freopen, getftime, lock, _open, read, sopen, _write**

---

outport

Function Outputs a word to a hardware port.

Syntax void outport(int *portid*, int *value*);

Prototype in dos.h

Remarks **outport** writes the word given by *value* to the output port specified by *portid*.

Return value None.

Portability **outport** is unique to the 8086 family.

See also **inport, inportb, outportb**

---

outportb

Function Outputs a byte to a hardware port.

Syntax

```
#include <dos.h>
void outportb(int portid,
              unsigned char value);
```

Prototype in dos.h

Remarks **outportb** is a macro that writes the byte given by *value* to the output port specified by *portid*.

If **outportb** is called when dos.h has been included, it will be treated as a macro that expands to inline code.

If you don't include dos.h, or if you do include dos.h and `#undef` the macro **outportb**, you will get the **outportb** function.

Return value None.

Portability **outportb** is unique to the 8086 family.

See also **inport, inportb, outport**

outtext

Function Displays a string in the viewport.

Syntax

```
#include <graphics.h>
void far outtext(char far *textstring);
```

Prototype in graphics.h

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

outtext outputs *textstring* at the 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 will be truncated.

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, outtextxy, settextjustify, textheight, textwidth**

outtextxy

Function Displays a string at a specified location.

Syntax

```
#include <graphics.h>
void far outtextxy(int x, int y,
                   char far *textstring);
```

Prototype in graphics.h

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 will be truncated.

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, outtext, textheight, textwidth**

parsfnm

Function Parses file name.

Syntax

```
#include <dos.h>
char *parsfnm(const char *cmdline,
              struct fcb *fcb, int opt);
```

Prototype in dos.h

Remarks **parsfnm** parses a string pointed to by *cmdline* for a file name. The string is normally a command line. The file name is placed in an FCB as a drive, file name, and extension. The FCB is pointed to by *fcb*.

The *opt* parameter is the value documented for AL in the DOS parse system call. See the MS-DOS *Programmer's Reference Manual* under system call 0x29 for a description of the parsing operations performed on the file name.

Return value On success, **parsfnm** returns a pointer to the next byte after the end of the file name. If there is any error in parsing the file name, **parsfnm** returns NULL.

Portability **parsfnm** is unique to DOS.

---

peek

Function Returns the word at memory location specified by *segment:offset*.

Syntax int peek(unsigned *segment*, unsigned *offset*);

Prototype in dos.h

Remarks **peek** returns the word at the memory location *segment:offset*.

If **peek** is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and `#undef` **peek**, you will get the function rather than the macro.

Return value **peek** returns the word of data stored at the memory location *segment:offset*.

Portability **peek** is unique to the 8086 family.

See also **harderr, peekb, poke**

---

peekb

Function Returns the byte of memory specified by *segment:offset*.

Syntax #include <dos.h>
char peekb(unsigned *segment*, unsigned *offset*);

Prototype in dos.h

Remarks **peekb** returns the byte at the memory location addressed by *segment:offset*.

If **peekb** is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and `#undef` **peekb**, you will get the function rather than the macro.

Return value **peekb** returns the byte of information stored at the memory location *segment:offset*.

Portability **peekb** is unique to the 8086 family.

See also **peek, pokeb**

---

perror

Function Prints a system error message.

Syntax void perror(const char *s);

Prototype in stdio.h

Remarks **perror** prints to the *stderr* stream (normally the console) the system error message for the last library routine that produced the error.

First the argument *s* is printed, then a colon, then the message corresponding to the current value of *errno*, and finally a newline. The convention is to pass the file name of the program as the argument string.

The array of error message strings is accessed through *sys_errlist*. *errno* can be used as an index into the array to find the string corresponding to the error number. None of the strings includes a newline character.

sys_nerr contains the number of entries in the array.

Refer to *errno*, *sys_errlist*, and *sys_nerr* in the "Global Variables" section of Chapter 1 for more information.

Return value None.

Portability **perror** is available on UNIX systems and is compatible with ANSI C.

See also **clearerr, eof, _strerror, strerror**

pieslice

Function Draws and fills in pie slice.

Syntax

```
#include <graphics.h>
void far pieslice(int x, int y, int stangle,
                  int endangle, int radius);
```

Prototype in graphics.h

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.

Note: If you are using a CGA or monochrome adapter, the examples in this book of how to use graphics functions may 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 example under **arc** on how to use the **pieslice** function.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

Examples See **arc**

poke

Function Stores an integer value at a memory location given by *segment:offset*.

Syntax void poke(unsigned *segment*, unsigned *offset*, int *value*);

Prototype in dos.h

Remarks **poke** stores the integer *value* at the memory location *segment:offset*.

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and `#undef` **poke**, you will get the function rather than the macro.

Return value None.

Portability **poke** is unique to the 8086 family.

See also **harderr, peek, pokeb**

pokeb

Function Stores a byte value at memory location *segment:offset*.

Syntax

```
#include <dos.h>
void pokeb(unsigned segment,
           unsigned offset, char value);
```

Prototype in dos.h

Remarks **pokeb** stores the byte *value* at the memory location *segment:offset*.

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and `#undef` **pokeb**, you will get the function rather than the macro.

Return value None.

Portability **pokeb** is unique to the 8086 family.

See also peekb, poke

poly

Function Generates a polynomial from arguments.

Syntax

```
#include <math.h>
double poly(double x, int degree,
            double coeffs[]);
```

Prototype in math.h

Remarks **poly** generates a polynomial in *x*, of degree *degree*, with coefficients *coeffs[0]*, *coeffs[1]*, ..., *coeffs[degree]*. For example, if *n* = 4, the generated polynomial is

$$coeffs[4]x^4 + coeffs[3]x^3 + coeffs[2]x^2 + coeffs[1]x + coeffs[0]$$

Return value **poly** returns the value of the polynomial as evaluated for the given *x*.

Portability **poly** is available on UNIX systems.

pow

Function Calculates *x* to the power of *y*.

Syntax

```
#include <math.h>
double pow(double x, double y);
```

Prototype in math.h

Remarks **pow** calculates x^y.

Return value On success, **pow** returns the value calculated, x^y.

Sometimes the arguments passed to **pow** produce results that overflow or are incalculable. When the correct value would overflow, **pow** returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to

ERANGE	Result out of range

errno is set to

EDOM   Domain error

if the argument *x* passed to **pow** is less than or equal to 0, and *y* is not a whole number. When this error occurs, **pow** returns the value negative HUGE_VAL.

If the arguments *x* and *y* passed to **pow** are both 0, **pow** returns 1.

Error-handling for **pow** can be modified through the function **matherr**.

Portability **pow** is available on UNIX systems and is compatible with ANSI C.

See also **exp, pow10, sqrt**

pow10

Function Calculates 10 to the power of *p*.

Syntax #include <math.h>
double pow10(int *p*);

Prototype in math.h

Remarks **pow10** computes 10^p.

Return value On success, **pow10** returns the value calculated, 10^p.

The result is actually calculated to **long double** accuracy. All arguments are valid, though some may cause an underflow or overflow.

Portablity Available on UNIX systems.

See also **exp, pow**

printf

Function Writes formatted output to *stdout*.

Syntax int printf(const char **format*[, *argument*, ...]);

Prototype in stdio.h

Remarks

printf accepts a series of arguments, applies to each a format specification contained in the format string given by *format*, and outputs the formatted data to *stdout*. There must be the same number of format specifications as arguments.

The Format String

The format string, present in each of the **...printf** function calls, controls how each function will convert, format, and print its arguments. There must be enough arguments for the format; if there are not, the results will be unpredictable and likely disastrous. Excess arguments (more than required by the format) are merely ignored.

The format string is a character string that contains two types of objects—*plain characters* and *conversion specifications*.

- The plain characters are simply copied verbatim to the output stream.
- The conversion specifications fetch arguments from the argument list and apply formatting to them.

Format Specifications

...printf format specifications have the following form:

```
% [flags] [width] [.prec] [F|N|h|l|L] type
```

Each conversion specification begins with the percent character (%). After the % come the following, in this order:

- an optional sequence of flag characters `[flags]`
- an optional width specifier `[width]`
- an optional precision specifier `[.prec]`
- an optional input-size modifier `[F|N|h|l|L]`
- the conversion type character `[type]`

Optional Format String Components

These are the general aspects of output formatting controlled by the optional characters, specifiers, and modifiers in the format string:

Character or Specifier	What It Controls or Specifies
flags	output justification, numeric signs, decimal points, trailing zeroes, octal and hex prefixes
width	minimum number of characters to print, padding with blanks or zeroes
precision	maximum number of characters to print; for integers, minimum number of digits to print
size	override default size of argument: **N** = near pointer **F** = far pointer **h** = **short int** **l** = **long** **L** = **long double**

...printf Conversion Type Characters

The following table lists the **...printf** conversion type characters, the type of input argument accepted by each, and in what format the output will appear.

The information in this table of type characters is based on the assumption that no flag characters, width specifiers, precision specifiers, or input-size modifiers were included in the format specification. To see how the addition of the optional characters and specifiers affects the **...printf** output, refer to the tables following this one.

Type Character	Input Argument	Format of Output
	Numerics	
d	integer	**signed** decimal **int**
i	integer	**signed** decimal **int**
o	integer	**unsigned** octal **int**
u	integer	**unsigned** decimal **int**
x	integer	**unsigned** hexadecimal **int** (with *a, b, c, d, e, f*)
X	integer	**unsigned** hexadecimal **int** (with *A, B, C, D, E, F*)
f	floating point	**signed** value of the form [-]*dddd.dddd*
e	floating point	**signed** value of the form [-]*d.dddd* *e* [+/–]*ddd*
g	floating point	**signed** value in either *e* or *f* form, based on given value and precision Trailing zeroes and the decimal point are printed only if necessary.
E	floating point	same as *e*, but with *E* for exponent
G	floating point	same as *g*, but with *E* for exponent if *e* format used

Type Character	Input Argument	Format of Output
		Characters
c	character	Single character.
s	string pointer	Prints characters until a null-terminator is hit or precision is reached.
%	none	The % character is printed.
		Pointers
n	pointer to **int**	Stores (in the location pointed to by the input argument) a count of the characters written so far.
p	pointer	Prints the input argument as a pointer: **far** pointers are printed as XXXX:YYYY **near** pointers are printed as YYYY (offset only)

Conventions

Certain conventions accompany some of these specifications, as summarized in the following table.

Characters	Conventions
e or **E**	The argument is converted to match the style [-] d.ddd...e[+/–]ddd where: • One digit precedes the decimal point. • The number of digits after the decimal point is equal to the precision. • The exponent always contains at least two digits.
f	The argument is converted to decimal notation in the style [-] *ddd.ddd...*, where the number of digits after the decimal point is equal to the precision (if a nonzero precision was given).
g or **G**	The argument is printed in style *e*, *E* or *f*, with the precision specifying the number of significant digits. Trailing zeroes are removed from the result, and a decimal point appears only if necessary. The argument is printed in style *e* or *f* (with some restraints) if *g* is the conversion character, and in style *E* if the character is *G*. Style *e* is used only if the exponent that results from the conversion is either (a) greater than the precision or (b) less than –4
x or **X**	For **x** conversions, the letters *a*, *b*, *c*, *d*, *e*, and *f* will appear in the output; for *X* conversions, the letters *A*, *B*, *C*, *D*, *E*, and *F* will appear.

Note: Infinite floating-point numbers are printed as +INF and –INF. An IEEE Not-a-Number is printed as +NAN or –NAN.

Flag Characters

The flag characters are minus (-), plus (+), sharp (#), and blank ( ). They can appear in any order and combination.

Flag	What It Specifies
–	Left-justifies the result, pads on the right with blanks. If not given, right-justifies result, pads on left with zeroes or blanks.
+	Signed conversion results always begin with a plus (+) or minus (-) sign.
blank	If value is nonnegative, the output begins with a blank instead of a plus; negative values still begin with a minus.
#	Specifies that arg is to be converted using an "alternate form." See the following table.

Note: Plus (+) takes precedence over blank ( ) if both are given.

Alternate Forms

If the # flag is used with a conversion character, it has the following effect on the argument (*arg*) being converted:

Conversion Character	How # Affects *arg*
c,s,d,i,u	No effect.
0	0 will be prepended to a nonzero *arg*.
x or X	0x (or 0X) will be prepended to *arg*.
e, E, or f	The result will always contain a decimal point even if no digits follow the point. Normally, a decimal point appears in these results only if a digit follows it.
g or G	Same as *e* and *E*, with the addition that trailing zeroes will not be removed.

Width Specifiers

The width specifier sets the minimum field width for an output value.

Width is specified in one of two ways: directly, through a decimal digit string, or indirectly, through an asterisk (*). If you use an asterisk for the width specifier, the next argument in the call (which must be an **int**) specifies the minimum output field width.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Width Specifier	How Output Width Is Affected
n	At least *n* characters are printed. If the output value has less than *n* characters, the output is padded with blanks (right-padded if - flag given, left-padded otherwise).
0n	At least *n* characters are printed. If the output value has less than *n* characters, it is filled on the left with zeroes.
*	The argument list supplies the width specifier, which must precede the actual argument being formatted.

Precision Specifiers

Precision specification always begins with a period (.), to separate it from any preceding width specifier. Then, like width, precision is specified either directly, through a decimal digit string, or indirectly, through an asterisk (*). If you use an asterisk for the precision specifier, the next argument in the call (treated as an **int**) specifies the precision.

If you use asterisks for the width or the precision, or for both, the width argument must immediately follow the specifiers, followed by the precision argument, then the argument for the data to be converted.

Precision Specifier	How Output Precision Is Affected
(none given)	Precision set to default: default = 1 for *d, i, o, u, x, X* types default = 6 for *e, E, f* types default = all significant digits for *g, G* types default = print to first null character for *s* types; no effect on *c* types
.0	For *d, i, o, u, x* types, precision set to default; for *e, E, f* types, no decimal point is printed.
.n	*n* characters or *n* decimal places are printed. If the output value has more than *n* characters, the output might be truncated or rounded. (Whether or not this happens depends on the type character.)
*	The argument list supplies the precision specifier, which must precede the actual argument being formatted.

Note: If an explicit precision of zero is specified, *and* the format specification for the field is one of the integer formats (that is, *d, i, o, u, x*), *and* the value to be printed is 0, no numeric characters will be output for that field (that is, the field will be blank).

Conversion Character	How Precision Specification (.n) Affects Conversion
d i o u x X	.*n* specifies that at least *n* digits will be printed. If the input argument has less than *n* digits, the output value is left-padded with zeroes. If the input argument has more than *n* digits, the output value is not truncated.
e E f	.*n* specifies that *n* characters will be be printed after the decimal point, and the last digit printed is rounded.
g G	.*n* specifies that at most *n* significant digits will be printed.
c	.*n* has no effect on the output.
s	.*n* specifies that no more than *n* characters will be printed.

Input-Size Modifier

The input-size modifier character (*F*, *N*, *h*, *l*, or *L*) gives the size of the subsequent input argument:

F = far pointer
N = near pointer
h = **short int**
l = **long**
L = **long double**

The input-size modifiers (*F*, *N*, *h*, *l*, and *L*) affect how the **...printf** functions interpret the data type of the corresponding input argument *arg*. *F* and *N* apply only to input *args* that are pointers (*%p*, *%s*, and *%n*). *h*, *L*, and *L* apply to input *args* that are numeric (integers and floating-point).

Both *F* and *N* reinterpret the input *arg*. Normally, the *arg* for a *%p*, *%s*, or *%n* conversion is a pointer of the default size for the memory model. *F* says "interpret *arg* as a far pointer." *N* says "interpret *arg* as a near pointer."

Both *h*, *l*, and *L* override the default size of the numeric data input args: *l* and *L* apply to integer (*d*, *i*, *o*, *u*, *x*, X) and floating-point (*e*, *E*, *f*, *g*, and *G*) types, while *h* applies to integer types only. Neither *h* nor *l* affect character (*c*, *s*) or pointer (*p*, *n*) types.

Input-Size Modifier	How *arg* Is Interpreted
F	*arg* is read as a **far** pointer.
N	*arg* is read as a **near** pointer. *N* cannot be used with any conversion in huge model.
h	*arg* is interpreted as a **short int** for *d*, *i*, *o*, *u*, *x*, or X.
l	*arg* is interpreted as a **long int** for *d*, *i*, *o*, *u*, *x*, or *X*; *arg* is interpreted as a **double** for *e*, *E*, *f*, *g*, or *G*.
L	*arg* is interpreted as a **long double** for *e*, *E*, *f*, *g*, or *G*.

Return value **printf** returns the number of bytes output. In the event of error, **printf** returns EOF.

Portability **printf** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **cprintf, ecvt, fprintf, fread, fscanf, putc, puts, putw, scanf, sprintf, vprintf, vsprintf**

Example

```
#define I 555
#define R 5.5

main()
{
   int i,j,k,l;
   char buf[7];
   char *prefix = buf;
   char tp[20];
   printf("prefix  6d      6o      8x         10.2e         "
          "10.2f\n");
   strcpy(prefix,"%");
   for (i=0;i<2;i++)
   {
      for (j=0;j<2;j++)
         for (k=0;k<2;k++)
            for (l=0;l<2;l++)
            {
               if (i==0)  strcat(prefix,"-");
               if (j==0)  strcat(prefix,"+");
               if (k==0)  strcat(prefix,"#");
               if (l==0)  strcat(prefix,"0");
               printf("%5s |",prefix);
               strcpy(tp,prefix);
               strcat(tp,"6d |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"6o |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"8x |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"10.2e |");
               printf(tp,R);
```

```
                strcpy(tp,prefix);
                strcat(tp,"10.2f |");
                printf(tp,R);
                printf("  \n");
                strcpy(prefix,"%");
            }
        }
}
```

Program output

```
prefix    6d      6o       8x          10.2e        10.2f
%-+#0 |+555    |01053  |0x22b     |+5.50e+00  |+5.50       |
 %-+# |+555    |01053  |0x22b     |+5.50e+00  |+5.50       |
 %-+0 |+555    |1053   |22b       |+5.50e+00  |+5.50       |
  %-+ |+555    |1053   |22b       |+5.50e+00  |+5.50       |
 %-#0 |555     |01053  |0x22b     |5.50e+00   |5.50        |
  %-# |555     |01053  |0x22b     |5.50e+00   |5.50        |
  %-0 |555     |1053   |22b       |5.50e+00   |5.50        |
   %- |555     |1053   |22b       |5.50e+00   |5.50        |
 %+#0 |+00555 |001053 |0x00022b |+05.50e+00 |+000005.50 |
  %+# |  +555 | 01053 |    0x22b |  +5.50e+00 |      +5.50 |
  %+0 |+00555 |001053 |0000022b |+05.50e+00 |+000005.50 |
   %+ |  +555 |  1053 |      22b |  +5.50e+00 |      +5.50 |
  %#0 |000555 |001053 |0x00022b |005.50e+00 |0000005.50 |
   %# |   555 | 01053 |    0x22b |   5.50e+00 |       5.50 |
   %0 |000555 |001053 |0000022b |005.50e+00 |0000005.50 |
    % |   555 |  1053 |      22b |   5.50e+00 |       5.50 |
```

putc

Function Outputs a character to a stream.

Syntax #include <stdio.h>
int putc(int *c*, FILE **stream*);

Prototype in stdio.h

Remarks **putc** is a macro that outputs the character *c* to the stream given by *stream*.

Return value On success, **putc** returns the character printed, *c*. On error, **putc** returns EOF.

Portability **putc** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also	**fprintf, fputc, fputch, getc, getchar, printf, putch, putchar**

putch

Function	Outputs character to screen.
Syntax	int putch(int *c*);
Prototype in	conio.h
Remarks	**putch** outputs the character *c* to the current text window. It is a text mode function performing direct video output to the console. **putch** does not translate line feed characters (`\n`) into hard return-linefeed pairs.
Return value	On success, **putch** returns the character printed, *c*. On error, it returns EOF.
Portability	**putch** works with IBM PCs and compatibles only.
See also	**cprintf, cputs, getch, getche, putc, putchar**

putchar

Function	Outputs character on *stdout*.
Syntax	#include <stdio.h> int putchar(int *c*);
Prototype in	stdio.h
Remarks	**putchar**(*c*) is a macro defined to be `putc(c, stdout)`.
Return value	On success, **putchar** returns the character *c*. On error, **putchar** returns EOF.
Portability	**putchar** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.
See also	**fputchar, getc, getchar, putc, putch, puts**

putenv

Function Adds string to current environment.

Syntax int putenv(const char **name*);

Prototype in stdlib.h

Remarks **putenv** accepts the string *name* and adds it to the environment of the *current* process. For example,

```
putenv("PATH=C:\FOO");
```

putenv can also be used to modify or delete an existing *name*. Delete an existing entry by making the variable value empty (for example, `MYVAR=`).

putenv can be used only to modify the current program's environment. Once the program ends, the old environment is restored.

Return value On success, **putenv** returns 0; on failure, –1.

Portability **putenv** is available on UNIX systems.

See also **getenv**

Example See **getenv**

putimage

Function Outputs a bit image onto the screen.

Syntax

```
#include <graphics.h>
void far putimage(int left, int top,
                  void far *bitmap, int op);
```

Prototype in graphics.h

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 on screen 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 will copy the source bitmap image onto the screen, XOR_PUT will XOR the source image with that already onscreen, OR_PUT will OR the source image with that onscreen, and so on.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getimage, imagesize, putpixel, setvisualpage**

Example See **getimage**

putpixel

Function Plots a pixel at a specified point.

Syntax #include <graphics.h>
void far putpixel(int *x*, int *y*, int *color*);

Prototype in graphics.h

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getpixel, putimage**

puts

Function Outputs a string to the *stdout* stream.

Syntax int puts(const char *s);

Prototype in stdio.h

Remarks **puts** copies the null-terminated string s to the standard output stream *stdout* and appends a newline character.

Return value On successful completion, **puts** returns a nonnegative value. Otherwise, it returns a value of EOF.

Portability **puts** is available on UNIX systems and is compatible with ANSI C.

See also **cputs, fputs, gets, printf, putchar**

puttext

Function Copies text from memory to text mode screen.

Syntax int puttext(int *left*, int *top*, int *right*,
int *bottom*, void **source*);

Prototype in conio.h

Remarks **puttext** writes the contents of the memory area pointed to by *source* out to the onscreen rectangle defined by *left*, *top*, *right*, and *bottom*.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

puttext places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom.

puttext is a text mode function performing direct video output.

Return value **puttext** returns a nonzero value if the operation succeeds; it returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

Portability **puttext** works only on IBM PCs and BIOS-compatible systems.

See also **gettext, movetext, window**

---

putw

Function Puts an integer on a stream.

Syntax

```
#include <stdio.h>
int putw(int w, FILE *stream);
```

Prototype in stdio.h

Remarks **putw** outputs the integer *w* to the given stream. **putw** neither expects nor causes special alignment in the file.

Return value On success, **putw** returns the integer *w*. On error, **putw** returns EOF.

Since EOF is a legitimate integer, **ferror** should be used to detect errors with **putw**.

Portability **putw** is available on UNIX systems.

See also **getw, printf**

---

qsort

Function Sorts using the quicksort algorithm.

Syntax

```
void qsort(void *base, size_t nelem,
           size_t width, int (*fcmp)
           (const void *, const void *) );
```

Prototype in stdlib.h

Remarks **qsort** is an implementation of the "median of three" variant of the quicksort algorithm. **qsort** sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by *fcmp*.

- *base* points to the base (0th element) of the table to be sorted.
- *nelem* is the number of entries in the table.

- *width* is the size of each entry in the table, in bytes.

fcmp*, the comparison function, accepts two arguments, *elem1* and *elem2*, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (elem1* and **elem2*), and returns an integer based on the result of the comparison.

If the item	*fcmp* returns
*elem1 < *elem2	an integer < 0
*elem1 = = *elem2	0
*elem1 > *elem2	an integer > 0

In the comparison, the less-than symbol (<) means that the left element should appear before the right element in the final, sorted sequence. Similarly, the greater-than (>) symbol means that the left element should appear after the right element in the final, sorted sequence.

Return value None.

Portability **qsort** is available on UNIX systems and is compatible with ANSI C.

See also **bsearch, lsearch**

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char list[5][4] = { "cat", "car", "cab", "cap", "can" };

main()
{
  int  x;

  qsort(&list, 5, sizeof(list[0]), strcmp);
  for (x = 0; x < 5; x++)
    printf("%s\n", list[x]);
}
```

Program output

```
cab
can
cap
car
cat
```

raise

Function Sends a software signal to the executing program.

Syntax

#include <signal.h>
int raise(int *sig*);

Prototype in signal.h

Remarks **raise** sends a signal of type *sig* to the program. If the program has installed a signal handler for the signal type specified by *sig*, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken.

The signal types currently defined in signal.h are

Signal	Meaning
SIGABRT	abnormal termination (*)
SIGFPE	bad floating point operation
SIGILL	illegal instruction (#)
SIGINT	control break interrupt
SIGSEGV	invalid access to storage (#)
SIGTERM	request for program termination (*)

Signal types marked with a (*) aren't generated by DOS or Turbo C during normal operation. However, they can be generated with **raise**. Signals marked by (#) *can't* be generated asynchronously on 8088 or 8086 processors but *can* be generated on some other processors (see **signal** for details).

Return value **raise** returns 0 if successful, nonzero otherwise.

Portability **raise** is available on UNIX systems, and is compatible with ANSI C.

See also **abort, signal**

Example

```
#include <signal.h>
main()
{
   int a, b, c;
```

```
   a = 10;
   b = 0;
   if (b == 0)
   /* Preempt divide by zero error */
      raise(SIGFPE);
   c = a / b;
}
```

rand

Function Random number generator.

Syntax int rand(void);

Prototype in stdlib.h

Remarks **rand** uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h; its value is $2^{15} \pm 1$.

Return value **rand** returns the generated pseudo-random number.

Portability **rand** is available on UNIX systems and is compatible with ANSI C.

See also **random, randomize, srand**

Example

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

main()
/* prints 5 random numbers from 0 to 32767 */
{
   int i;
   /* start at a random place */
   srand(time(NULL) % 37);
   for (i=0; i<5; i++)
      printf("%d\n", rand());
}
```

randbrd

Function Random block read.

Syntax

```
#include <dos.h>
int randbrd(struct fcb *fcb, int rcnt);
```

Prototype in dos.h

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 . 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 will be 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.

Portability **randbrd** is unique to DOS.

See also **getdta, randbwr, setdta**

randbwr

Function Random block write.

Syntax

```
#include <dos.h>
int randbwr(struct fcb *fcb, int rcnt);
```

Prototype in dos.h

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 DOS 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 will be advanced by the number of records actually written.

Return value

The following values are returned, depending upon 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).

Portability

randbwr is unique to DOS.

See also

randbrd

random

Function

Random number generator.

Syntax

```
#include <stdlib.h>
int random(int num);
```

Prototype in

stdlib.h

Remarks

random returns a random number between 0 and (*num*-1). **random**(*num*) is a macro defined as (**rand**() % (*num*)). Both *num* and the random number returned are integers.

Return value

random returns a number between 0 and (*num*-1).

Portability

A corresponding function exists in Turbo Pascal.

See also

rand, randomize, srand

Example

```
#include <stdlib.h>
#include <time.h>
```

```
/* prints a random number in the range 0-99 */
main()
{
   int n;
   randomize();
   /* selects a random number between 1 and 20 */
   n = random(20) + 1;
   while (n-- > 0)
      printf("%d ", random (100));
   printf("\n");
}
```

randomize

Function Initializes random number generator.

Syntax #include <stdlib.h>
#include <time.h>
void randomize(void);

Prototype in stdlib.h

Remarks **randomize** initializes the random number generator with a random value. Because **randomize** is implemented as a macro that calls the **time** function prototyped in time.h, we recommend that you also include time.h when you are using this routine.

Return value None.

Portability A corresponding function exists in Turbo Pascal.

See also **rand, random, srand**

_read

Function Reads from file.

Syntax int _read(int *handle*, void **buf*, unsigned *len*);

Prototype in io.h

Remarks **_read** attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*. **_read** is a direct call to the DOS read system call.

When a file is opened in text mode, **_read** does not remove carriage returns.

handle is a file handle obtained from a **creat, open, dup,** or **dup2** call.

On disk files, **_read** begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that **_read** can read is 65534, since 65535 (0xFFFF) is the same as –1, the error return indicator.

Return value On successful completion, **_read** returns a positive integer indicating the number of bytes placed in the buffer. On end-of-file, **_read** returns zero. On error, it returns –1, and *errno* is set to one of the following:

EACCES	Permission denied
EBADF	Bad file number

Portability **_read** is unique to DOS.

See also **_open, read, _write**

read

Function Reads from file.

Syntax int read(int *handle*, void **buf*, unsigned *len*);

Prototype in io.h

Remarks **read** attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*.

For a file opened in text mode, **read** removes carriage returns and reports end-of-file when it reaches the end of the file.

handle is a file handle obtained from a **creat, open, dup,** or **dup2** call.

On disk files, **read** begins reading at the current file pointer. When the reading is complete, it increments the

	file pointer by the number of bytes read. On devices, the bytes are read directly from the device.
	The maximum number of bytes that **read** can read is 65534, since 65535 (0xFFFF) is the same as –1, the error return indicator.
Return value	On successful completion, **read** returns an integer indicating the number of bytes placed in the buffer. If the file was opened in text mode, **read** does not count carriage returns or *Ctrl-Z* characters in the number of bytes read.
	On end-of-file, **read** returns 0. On error, **read** returns –1 and sets *errno* to one of the following: EACCES Permission denied EBADF Bad file number
Portability	**read** is available on UNIX systems.
See also	**open, _read, write**

realloc

Function	Reallocates main memory.
Syntax	#include <stdlib.h> void *realloc(void **block*, size_t *size*);
Prototype in	stdlib.h, alloc.h
Remarks	**realloc** attempts to shrink or expand the previously allocated block to *size* bytes. The *block* argument points to a memory block previously obtained by calling **malloc, calloc,** or **realloc.** If *block* is a null pointer, **realloc** works just like **malloc.**
	realloc adjusts the size of the allocated block to *size,* copying the contents to a new location if necessary.
Return value	**realloc** returns the address of the reallocated block, which may be different than the address of the original block. If the block cannot be reallocated, or *size* == 0, **realloc** returns NULL.

Portability realloc is available on UNIX systems and is compatible with ANSI C.

See also **calloc, farrealloc, free, malloc**

Example See **malloc**

rectangle

Function Draws a rectangle.

Syntax
#include <graphics.h>
void far rectangle(int *left*, int *top*,
int *right*, int *bottom*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

Example

```
int i;

for (i = 0; i < 10; i++)
   rectangle(20-2*i,20-2*i,10*(i+2),10*(i+2));
```

registerbgidriver

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

Syntax
#include <graphics.h>
int registerbgidriver(void (*driver)(void));

Prototype in graphics.h

Remarks

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

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 Appendix D.

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 drive number to be used.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

graphresult, initgraph, installuserdriver, registerbgifont

Example

```
/* Register the EGA/VGA driver */
if (registerbgidriver(EGAVGA_driver) < 0) exit(1);
```

---

registerbgifont

Function

Registers linked-in stroked font code.

Syntax

#include <graphics.h>
int registerbgifont(void (*font)(void));

Prototype in

graphics.h

Remarks

Calling **registerbigfont** 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 in Appendix D.

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **graphresult, initgraph, installuserdriver, registerbgidriver, settextstyle**

Example

```
/* Register the gothic font */
if (registerbgifont(gothic_font) != GOTHIC_FONT) exit(1);
```

remove

Function Removes a file.

Syntax #include <stdio.h>
int remove(const char **filename*);

Prototype in stdio.h

Remarks **remove** deletes the file specified by *filename*. It is a macro that simply translates its call to a call to **unlink**.

Return value On successful completion, **remove** returns 0. On error, it returns –1, and *errno* is set to one of the following:

ENOENT	No such file or directory
EACCES	Permission denied

Portability **remove** is available on UNIX systems and is compatible with ANSI C.

See also **unlink**

rename

Function Renames a file.

Syntax int rename(const char **oldname*, const char **newname*);

Prototype in stdio.h

Remarks **rename** changes the name of a file from *oldname* to *newname*. If a drive specifier is given in *newname*, the specifier must be the same as that given in *oldname*.

Directories in *oldname* and *newname* need not be the same, so **rename** can be used to move a file from one directory to another. Wildcards are not allowed.

Return value On successfully renaming the file, **rename** returns 0. In the event of error, –1 is returned, and *errno* is set to one of the following:

ENOENT	No such file or directory
EACCES	Permission denied
ENOTSAM	Not same device

Portability **rename** is compatible with ANSI C.

restorecrtmode

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

Syntax

```
#include <graphics.h>
void far restorecrtmode(void);
```

Prototype in graphics.h

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; it is used only when the screen is in text mode, to change to a different text mode.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getgraphmode, initgraph, setgraphmode**

rewind

Function Repositions a file pointer to the beginning of a stream.

Syntax

```
#include <stdio.h>
void rewind(FILE *stream);
```

Prototype in stdio.h

Remarks **rewind**(*stream*) is equivalent to **fseek**(*stream*, 0L, SEEK_SET), except that **rewind** clears the end-of-file and error indicators, while **fseek** only clears the end-of-file indicator.

After **rewind**, the next operation on an update file can be either input or output.

Return value None.

Portability **rewind** is available on all UNIX systems, and it is compatible with ANSI C.

See also **fopen, fseek, ftell**

Example See **fseek**

rmdir

Function Removes a DOS file directory.

Syntax int rmdir(const char **path*);

Prototype in dir.h

Remarks **rmdir** deletes the directory whose path is given by *path*. The directory named by *path*

- must be empty
- must not be the current working directory
- must not be the root directory

Return value **rmdir** returns 0 if the directory is successfully deleted. A return value of –1 indicates an error, and *errno* is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file function not found

See also **chdir, getcurdir, getcwd, mkdir**

_rotl

Function Bit-rotates an **unsigned** integer value to the left.

Syntax unsigned _rotl(unsigned *value*, int *count*);

Prototype in stdlib.h

Remarks **_rotl** rotates the given *value* to the left *count* bits. The value rotated is an **unsigned** integer.

Return value **_rotl** returns the value of *value* left-rotated *count* bits.

See also **_lrotl**

Example

```
#include <stdlib.h>

main()
{
   printf("rotate 0xABCD 4 bits left  = %04X\n",
          _rotl(0xABCD, 4));
   printf("rotate 0xABCD 4 bits right = %04X\n",
          _rotr(0xABCD, 4));
   printf("rotate 0x55555555 1 bit left  = %08lX\n",
          _lrotl(0x55555555L, 1));
   printf("rotate 0xAAAAAAAA 1 bit right = %08lX\n",
          _lrotr(0xAAAAAAAAL,1));
}
```

Program Output

```
rotate 0xABCD 4 bits left  = BCDA
rotate 0xABCD 4 bits right = DABC
rotate 0x55555555 1 bit left  = AAAAAAAA
rotate 0xAAAAAAAA 1 bit right = 55555555
```

_rotr

Function Bit-rotates an **unsigned** integer value to the right.

Syntax unsigned _rotr(unsigned *value*, int *count*);

Prototype in stdlib.h

Remarks **_rotr** rotates the given *value* to the right *count* bits. The value rotated is an **unsigned** integer.

Return value **_rotr** returns the value of *value* right-rotated *count* bits.

See also **_lrotr**

sbrk

Function Changes data segment space allocation.

Syntax void *sbrk(int *incr*);

Prototype in alloc.h

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 *errno* is set to

ENOMEM Not enough core

Portability **sbrk** is available on UNIX systems.

See also **brk**

scanf

Function Scans and formats input from the *stdin* stream.

Syntax int scanf(const char **format*[, address, ...]);

Prototype in stdio.h

Remarks **scanf** scans a series of input fields, one character at a time, reading from the *stdin* stream. Then each field is formatted according to a format specification passed to **scanf** in the format string pointed to by *format*. Finally, **scanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifications and addresses as there are input fields.

The Format String

The format string present in **scanf** and the related functions **cscanf, fscanf, sscanf, vscanf, vfscanf,** and **vsscanf** controls how each function will scan, convert, and store its input fields. There must be enough address arguments for the given format specifications; if not, the results are unpredictable, and likely disastrous. Excess address arguments (more than required by the format) are merely ignored.

The format string is a character string that contains three types of objects: whitespace characters, non-whitespace characters, and format specifications.

- The whitespace characters are blank ( ), tab (\t) or newline (\n). If a **...scanf** function encounters a whitespace character in the format string, it will read, but not store, all consecutive whitespace characters up to the next non-whitespace character in the input.
- The non-whitespace characters are all other ASCII characters except the percent sign (%). If a **...scanf** function encounters a non-whitespace character in the format string, it will read, but not store, a matching non-whitespace character.
- The format specifications direct the **...scanf** functions to read and convert characters from the input field into specific types of values, then store them in the locations given by the address arguments.

Trailing whitespace is left unread (including a newline), unless explicitly matched in the format string.

Format Specifications

...scanf format specifications have the following form:

```
% [*] [width] [F|N] [h|l|L] type_character
```

Each format specification begins with the percent character (%). After the % come the following, in this order:

- an optional assignment-suppression character [*]
- an optional width specifier [width]
- an optional pointer size modifier [F|N]
- an optional argument-type modifier [h|l|L]
- the type character

Optional Format String Components

These are the general aspects of input formatting controlled by the optional characters and specifiers in the **...scanf** format string:

Character or Specifier	What It Controls or Specifies
*	Suppresses assignment of the next input field.
width	Maximum number of characters to read; fewer characters might be read if the **...scanf** function encounters a whitespace or unconvertible character.
size	Overrides default size of address argument. *N* = near pointer *F* = far pointer
argument type	Overrides default type of address argument. *h* = **short int** *l* = **long int** (if the type character specifies an integer conversion) *l* = **double** (if the type character specifies a floating-point conversion) *L* = **long double** (valid only with floating-point conversions)

...scanf Type Characters

The following table lists the **...scanf** type characters, the type of input expected by each, and in what format the input will be stored.

The information in this table is based on the assumption that no optional characters, specifiers, or modifiers (*, width, or size) were included in the format specification. To see how the addition of the optional elements affects the **...scanf** input, refer to the tables following this one.

Type Character	Expected Input	Type of Argument
		Numerics
d	Decimal integer	Pointer to **int** (int **arg*)
D	Decimal integer	Pointer to **long** (long **arg*)
o	Octal integer	Pointer to **int** (int **arg*)
O	Octal integer	Pointer to **long** (long **arg*)
i	Decimal, octal, or hexadecimal integer	Pointer to **int** (int **arg*)
I	Decimal, octal, or hexadecimal integer	Pointer to **long** (long **arg*)
u	Unsigned decimal integer	Pointer to **unsigned int** (unsigned int **arg*)
U	Unsigned decimal integer	Pointer to **unsigned long** (unsigned long **arg*)
x	Hexadecimal integer	Pointer to **int** (int **arg*)
X	Hexadecimal integer	Pointer to **long** (long **arg*)
e	Floating	Pointer to **float** (float **arg*)
E	Floating	Pointer to **float** (float **arg*)
f	Floating	Pointer to **float** (float **arg*)
g	Floating	Pointer to **float** (float **arg*)
G	Floating	Pointer to **float** (float **arg*)

Type Character	Expected Input	Type of Argument
		Characters
s	Character string	Pointer to array of characters (char *arg[]*)
c	Character	Pointer to character (char **arg*) if a field width *W* is given along with the *c*-type character (such as %5c).
		Pointer to array of *W* characters (char *arg*[W])
%	% character	No conversion is done; the % character is stored.
		Pointers
n		Pointer to **int** (int **arg*). The number of characters read successfully, up to the *%n,* is stored in this **int**.
p	Hexadecimal form YYYY:ZZZZ or ZZZZ	Pointer to an object (*far** or *near**) %*p* conversions default to the pointer size native to the memory model.

Input Fields

Any one of the following is an input field:

- all characters up to (but not including) the next whitespace character
- all characters up to the first one that cannot be converted under the current format specification (such as an 8 or 9 under octal format)
- up to *n* characters, where *n* is the specified field width

Conventions

Certain conventions accompany some of these format specifications, as summarized here.

%c conversion
This specification reads the next character, including a whitespace character. To skip one whitespace character and read the next non-whitespace character, use %1s.

%Wc conversion (W = width specification)
The address argument is a pointer to an array of characters; the array consists of *W* elements (char *arg*[*W*]).

%s conversion
The address argument is a pointer to an array of characters (char *arg*[]).

The array size must be *at least* (*n*+1) bytes, where *n* equals the length of string *s* (in characters). A space or newline terminates the input field. A null-terminator is automatically appended to the string and stored as the last element in the array.

%[search_set] conversion
The set of characters surrounded by square brackets can be substituted for the *s*-type character. The address argument is a pointer to an array of characters (char *arg*[]).

These square brackets surround a set of characters that define a *search set* of possible characters making up the string (the input field).

If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the square brackets. (Normally, a caret will be included in the inverted search set unless explicitly listed somewhere after the first caret.)

The input field is a string not delimited by whitespace. The **...scanf** function reads the corresponding input field up to the first character it reaches that does not appear in the search set (or in the inverted search set). Two examples of this type of conversion are

`%[abcd]`	Searches for any of the characters *a*, *b*, *c*, and *d* in the input field.
`%[^abcd]`	Searches for any characters except *a*, *b*, *c*, and *d* in the input field.

You can also use a *range facility* shortcut to define a range of characters (numerics or letters) in the search set. For example, to catch all decimal digits, you could define the search set by using

```
%[0123456789]
```

or you could use the shortcut to define the same search set by using

```
%[0-9]
```

To catch alphanumerics, you could use the following shortcuts:

`%[A-Z]`	Catches all uppercase letters.
`%[0-9A-Za-z]`	Catches all decimal digits and all letters (uppercase and lowercase).
`%[A-FT-Z]`	Catches all uppercase letters from *A* through *F* and from *T* through *Z*.

The rules covering these search set ranges are straightforward:

- The character prior to the hyphen (-) must be lexically less than the one after it.
- The hyphen must not be the first nor the last character in the set. (If it is first or last, it is considered to just be the hyphen character, not a range definer.)
- The characters on either side of the hyphen must be the ends of the range and not part of some other range.

Here are some examples where the hyphen just means the hyphen character, not a range between two ends:

`%[-+*/]`	The four arithmetic operations
`%[z-a]`	The characters *z*, –, and *a*
`%[+0-9-A-Z]`	The characters + and –, and the ranges 0 through 9 and *A* through *Z*
`%[+0-9A-Z-]`	Also the characters + and –, and the ranges 0 through 9 and *A* through *Z*
`%[^-0-9+A-Z]`	All characters except + and –, and those in the ranges 0 through 9 and *A* through *Z*

%e, %E. %f, %g, and %G (floating-point) conversions

Floating-point numbers in the input field must conform to the following generic format:

```
[+/-] ddddddddd [.] dddd [E | e] [+/-] ddd
```

where [*item*] indicates that *item* is optional, and *ddd* represents decimal, octal, or hexadecimal digits.

In addition, +INF, -INF, +NAN, and –NAN are recognized as floating-point numbers. Note that the sign and capitalization are required.

%d, %i, %o, %x, %D, %I, %O, %X, %c, %n conversions

A pointer to **unsigned** character, **unsigned** integer, or **unsigned long** can be used in any conversion where a pointer to a character, integer, or **long** is allowed.

Assignment-Suppression Character

The assignment-suppression character is an asterisk (*); it is not to be confused with the C indirection (pointer) operator (also an asterisk).

If the asterisk follows the percent sign (%) in a format specification, the next input field will be scanned but will not be assigned to the next address argument. The suppressed input data is assumed to be of the type specified by the type character that follows the asterisk character.

The success of literal matches and suppressed assignments is not directly determinable.

Width Specifiers

The width specifier (*n*), a decimal integer, controls the maximum number of characters that will be read from the current input field.

If the input field contains fewer than *n* characters, the **...scanf** function reads all the characters in the field, then proceeds with the next field and format specification.

If a whitespace or nonconvertible character occurs before width characters are read, the characters up to that character are read, converted, and stored, then the function attends to the next format specification.

A nonconvertible character is one that cannot be converted according to the given format (such as an 8 or 9 when the format is octal, or a *J* or *K* when the format is hexadecimal or decimal).

Width Specifier	How Width of Stored Input Is Affected
n	Up to *n* characters will be read, converted, and stored in the current address argument.

Input-Size and Argument-Type Modifiers

The input-size modifiers (*N* and *F*) and argument-type modifiers (*h, l,* and *L*) affect how the **...scanf** functions interpret the corresponding address argument *arg[f]*.

F and *N* override the default or declared size of *arg.*

h, l, and *L* indicate which type (version) of the following input data is to be used (*h* = **short**, *l* = **long**, *L* = **long double**). The input data will be converted to the specified version, and the *arg* for that input data should point to an object of the corresponding size (**short** object for **%h**, **long** or **double** object for **%l**, and **long double** object for **%L**).

Modifier	How Conversion Is Affected
F	Overrides default or declared size; *arg* interpreted as far pointer.
N	Overrides default or declared size; *arg* interpreted as near pointer. Cannot be used with any conversion in huge model.
h	For *d, i, o, u, x* types: convert input to **short int,** store in **short** object.
	For *D, I, O, U, X* types: has no effect.
	For *e, f, c, s, n, p* types: has no effect.
l	For *d, i, o, u, x* types: convert input to **long int**, store in **long** object.
	For *e, f, g* types: convert input to **double,** store in **double** object.
	For *D, I, O, U, X* types: has no effect.
	For *c, s, n, p* types: has no effect.
L	For *e, f, g* types: convert input to a **long double**, store in **long double** object. **L** has no effect on other formats.

When scanf Stops Scanning

scanf may stop scanning a particular field before reaching the normal field-end character (whitespace), or may terminate entirely, for a variety of reasons.

scanf will stop scanning and storing the current field and proceed to the next input field if any of the following occurs:

- An assignment-suppression character (*) appears after the percent character in the format specification; the current input field is scanned but not stored.
- *width* characters have been read (*width* = width specification, a positive decimal integer in the format specification).
- The next character read cannot be converted under the current format (for example, an *A* when the format is decimal).
- The next character in the input field does not appear in the search set (or does appear in an inverted search set).

When **scanf** stops scanning the current input field for one of these reasons, the next character is assumed to be unread and to be the first character of the following input field, or the first character in a subsequent read operation on the input.

scanf will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next character in the input field is EOF.
- The format string has been exhausted.

If a character sequence that is not part of a format specification occurs in the format string, it must match the current sequence of characters in the input field; **scanf** will scan, but not store, the matched characters. When a conflicting character occurs, it remains in the input field as if it were never read.

Return value **scanf** returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields that were not stored.

If **scanf** attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

Portability **scanf** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **cscanf, fscanf, printf, sscanf, vfscanf, vscanf, vsscanf**

searchpath

Function Searches the DOS path for a file.

Syntax char *searchpath(const char *file*);

Prototype in dir.h

Remarks **searchpath** attempts to locate *file*, searching along the DOS path, which is the `PATH=...` string in the environment. A pointer to the complete path-name string is returned as the function value.

searchpath searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found or the path is exhausted.

When the file is located, a string is returned containing the full path name. This string can be used in a call to access the file (for example, with **fopen** or **exec...**).

The string returned is located in a static buffer and is overwritten on each subsequent call to **searchpath**.

Return value **searchpath** returns a pointer to a file name string if the file is successfully located; otherwise, **searchpath** returns null.

Portability **searchpath** is unique to DOS.

See also **exec..., spawn..., system**

Example

```
#include <stdio.h>
#include <dir.h>

main()
{
   char *p;
   p = searchpath("TLINK.EXE");
   printf("Search for TLINK.EXE : %s\n", p);
   p = searchpath("NOTEXIST.FIL");
   printf("Search for NOTEXIST.FIL : %s\n", p);
}
```

Program output

```
Search for TLINK.EXE : C:\BIN\TLINK.EXE
Search for NOTEXIST.FIL : (null)
```

sector

Function Draws and fills an elliptical pie slice.

Syntax #include <graphics.h>
void far sector(int *x*, int *y*,
int *stangle*, int *endangle*,
int *xradius*, int *yradius*);

Prototype in graphics.h

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** will return a value of –6 (grNoScanMem).

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

---

segread

Function Reads segment registers.

Syntax
```
#include <dos.h>
void segread(struct SREGS *segp);
```

Prototype in dos.h

Remarks **segread** places the current values of the segment registers into the structure pointed to by *segp*.

This call is intended for use with **intdosx** and **int86x.**

Return value None.

Portability **segread** is unique to the 8086 family of processors.

See also **FP_OFF, int86 intdos, MK_FP, movedata**

---

setactivepage

Function Sets active page for graphics output.

Syntax
```
#include <graphics.h>
void far setactivepage(int page);
```

Prototype in graphics.h

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

The active graphics page may or may 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **setvisualpage**

Example

```
cleardevice();
/* make page 0 (blank) visible */
setvisualpage(0);
/* use page 1 for output */
setactivepage(1);
/* draw a bar in page 1 */
bar(50, 50, 150, 150);
/* show page 1 (with bar) */
setvisualpage(1);
```

setallpalette

Function Changes all palette colors as specified.

Syntax

```
#include <graphics.h>
void far setallpalette(struct palettetype
                               far *palette);
```

Prototype in graphics.h

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 defined in graphics.h.

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 on the screen. Each time a palette color is changed, all occurrences of that color on the screen 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** will return -11 (grError), and the current palette remains unchanged.

Portability

This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also

getpalette, graphresult, setbkcolor, setcolor, setpalette

setaspectratio

Function Changes the default aspect ratio correction factor.

Syntax #include <graphics.h>
void far setaspectratio(int *xasp*, int *yasp*);

Prototype in graphics.h

Remarks **setaspectratio** is used to change the default aspect ratio of the graphics system. The aspect ratio is used by the graphics system to make sure that circles are drawn round. If circles appear elliptical, the monitor is not aligned properly. This can be corrected in the hardware by realigning the monitor, or it can be changed 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **circle, getaspectratio**

setbkcolor

Function Sets the current background color using the palette.

Syntax #include <graphics.h>
void far setbkcolor(int *color*);

Prototype in graphics.h

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

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

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

```
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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getbkcolor, setallpalette, setcolor, setpalette**

setblock

Function Modifies the size of a previously allocated block.

Syntax int setblock(unsigned *segx*, unsigned *newsize*);

Prototype in dos.h

Remarks **setblock** modifies the size of a memory segment. *segx* is the segment address returned by a previous call to **allocmem**. *newsize* is the new, requested size in paragraphs.

Return value **setblock** returns –1 on success. In the event of error, it returns the size of the largest possible block (in paragraphs), and *_doserrno* is set.

Portability **setblock** is unique to DOS.

See also **allocmem**

setbuf

Function Assigns buffering to a stream.

Syntax

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Prototype in stdio.h

Remarks **setbuf** causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after *stream* has been opened.

If *buf* is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h).

stdin and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered. **setbuf** can be used to change the buffering style being used.

Unbuffered means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

setbuf will produce unpredictable results unless it is called immediately after opening *stream* or after a call to **fseek**. Calling **setbuf** after *stream* has been unbuffered is legal and will not cause problems.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file

before returning from the function where the buffer was declared.

Return value None.

Portability **setbuf** is available on UNIX systems and is compatible with ANSI C.

See also **fflush, fopen, fseek, setvbuf**

Example See **setvbuf**

---

setcbrk

Function Sets control-break setting.

Syntax int setcbrk(int *cbrkvalue*);

Prototype in dos.h

Remarks **setcbrk** uses the DOS system call 0x33 to set control-break checking on or off.

value = 0 Turns checking off (check only during I/O to console, printer, or communications devices).

value = 1 Turns checking on (check at every system call).

Return value **setcbrk** returns *cbrkvalue*, the value passed.

Portability **setcbrk** is unique to DOS.

See also **getcbrk**

---

setcolor

Function Sets the current drawing color using the palette.

Syntax #include <graphics.h>
void far setcolor(int *color*);

Prototype in graphics.h

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, etc., 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

Numeric Value	*Symbolic Name*
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getcolor, getmaxcolor, setallpalette, setbkcolor, setpalette**

setdate

Function Sets DOS date.

Syntax #include <dos.h>
void setdate(struct date **datep*);

Prototype in dos.h

Remarks **setdate** sets the system date (month, day, and year) to that in the **date** structure pointed to by *datep*.

The **date** structure is defined as follows:

```
struct date {
   int da_year;                       /* current year */
   char da_day;                  /* day of the month */
   char da_mon;                   /* month (1 = Jan) */
};
```

Return value None.

Portability **setdate** is unique to DOS.

See also **getdate, gettime, settime**

Example See **getdate**

setdisk

Function Sets current disk drive.

Syntax int setdisk(int *drive*);

Prototype in dir.h

Remarks **setdisk** sets the current drive to the one associated with *drive*: 0 for A, 1 for B, 2 for C, and so on (equivalent to DOS call 0x0E).

Return value **setdisk** returns the total number of drives available.

Portability **setdisk** is unique to DOS.

See also **getdisk**

setdta

Function Sets disk transfer address.

Syntax void setdta(char far **dta*);

Prototype in dos.h

Remarks **setdta** changes the current setting of the DOS disk transfer address (DTA) to the value given by *dta*.

Return value None.

Portability **setdta** is unique to DOS.

See also **getdta**

setfillpattern

Function Selects a user-defined fill pattern.

Syntax #include <graphics.h>
void far setfillpattern(char far **upattern*, int *color*);

Prototype in graphics.h

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 will be plotted.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

setfillstyle

Function Sets the fill pattern and color.

Syntax

```
#include <graphics.h>
void far setfillstyle(int pattern, int color);
```

Prototype in graphics.h

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*, 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 cross-hatch 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 uses the current background color.

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

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

setftime

Function Sets file date and time.

Syntax

```
#include <io.h>
int setftime(int handle, struct ftime *ftimep);
```

Prototype in io.h

Remarks **setftime** sets the file date and time of the disk file associated with the open *handle* to the date and time in the **ftime** structure pointed to by *ftimep*.

The **ftime** structure is defined as follows:

```
struct ftime {
   unsigned ft_tsec: 5;                    /* two seconds */
   unsigned ft_min: 6;                         /* minutes */
   unsigned ft_hour: 5;                          /* hours */
   unsigned ft_day: 5;                            /* days */
   unsigned ft_month: 4;                        /* months */
   unsigned ft_year: 7;                     /* year - 1980*/
};
```

Return value **setftime** returns 0 on success.

In the event of an error, –1 is returned, and the global variable *errno* is set to one of the following:

EINVFNC	Invalid function number
EBADF	Bad file number

Portability **setftime** is unique to DOS.

See also **getftime**

Example See **getdate**

setgraphbufsize

Function Changes the size of the internal graphics buffer.

Syntax #include <graphics.h>
unsigned far setgraphbufsize(unsigned *bufsize*);

Prototype in graphics.h

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, which is allocated by **_graphgetmem**, is 4096 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **closegraph, _graphfreemem, _graphgetmem, initgraph, sector**

Example

```
int cbsize;
/* get current size */
cbsize = setgraphbufsize(1000);
/* restore size */
setgraphbufsize(cbsize);
printf("The graphics buffer is currently %u bytes.",
       cbsize);
```

setgraphmode

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

Syntax #include <graphics.h>
void far setgraphmode(int *mode*);

Prototype in graphics.h

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 (CP, 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** will return a value of –10 (grInvalidMode).

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **getgraphmode, getmoderange, graphresult, initgraph, restorecrtmode**

setjmp

Function Sets up for nonlocal goto.

Syntax #include <setjmp.h>
int setjmp(jmp_buf *jmpb*);

Prototype in setjmp.h

Remarks **setjmp** captures the complete *task state* in *jmpb* and returns 0.

A later call to **longjmp** with *jmpb* restores the captured task state and returns in such a way that **setjmp** appears to have returned with the value *val*.

A task state is

- all segment registers (CS, DS, ES, SS)
- register variables (SI, DI)
- stack pointer (SP)
- frame base pointer (BP)
- flags

A task state is complete enough that **setjmp** can be used to implement co-routines.

setjmp must be called before **longjmp**. The routine that calls **setjmp** and sets up *jmpb* must still be active and cannot have returned before the **longjmp** is called. If it has returned, the results are unpredictable.

setjmp is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

Return value **setjmp** returns 0 when it is initially called.

Portability **setjmp** is available on UNIX systems and is compatible with ANSI C.

See also **longjmp, signal**

Example See **longjmp**

---

setlinestyle

Function Sets the current line width and style.

Syntax #include <graphics.h>
void far setlinestyle(int *linestyle*, unsigned *upattern*,
int *thickness*);

Prototype in graphics.h

Remarks **setlinestyle** sets the style for all lines drawn by **line, lineto, rectangle, drawpoly,** etc.

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*, 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 it is ignored.

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

Return value If invalid input is passed to **setlinestyle, graphresult** will return –11, and the current line style will remain unchanged.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **bar3d, getlinesettings, graphresult, line, linerel, lineto, rectangle**

setmem

Function Assigns a value to a range of memory.

Syntax void setmem(void **dest*, unsigned *length*, char *value*);

Prototype in mem.h

Remarks **setmem** sets a block of *length* bytes, pointed to by *dest*, to the byte *value*.

Return value None.

Portability **setmem** is unique to the 8086 family.

See also **memset, strset**

setmode

Function Sets mode of open file.

Syntax #include <fcntl.h>
int setmode(int *handle*, int *amode*);

Prototype in io.h

Remarks **setmode** sets the mode of the open file associated with *handle* to either binary or text. The argument *amode* must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.)

Return value **setmode** returns 0 if successful. On error it returns –1 and sets *errno* to

EINVAL Invalid argument

Portability **setmode** is available on UNIX systems.

See also **_creat, creat,_open, open**

setpalette

Function Changes one palette color.

Syntax

```
#include <graphics.h>
void far setpalette(int colornum, int color);
```

Prototype in graphics.h

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 defined in graphics.h.

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 on the screen. Each time a palette color is changed, all occurrences of that color on the screen will 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** will return –11, and the current palette remains unchanged.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

setrgbpalette

Function Allows user to define colors for the IBM8514.

Syntax

```
#include <graphics.h>
void far setrgbpalette(int colornum, int red,
                                    int green, int blue);
```

Prototype in graphics.h

Remarks **setrgbpalette** can be used with the IBM8514 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 IBM8514 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 IBM8514 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **setpalette**

settextjustify

Function Sets text justification for graphics functions.

Syntax

```
#include <graphics.h>
void far settextjustify(int horiz, int vert);
```

Prototype in graphics.h

Remarks Text output after a call to **settextjustify** will be justified around the 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**.

Name	*Value*	*Description*
LEFT_TEXT	0	horiz
CENTER_TEXT	1	horiz and vert
RIGHT_TEXT	2	horiz
BOTTOM_TEXT	0	vert
TOP_TEXT	2	vert

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** will return –11, and the current text justification remains unchanged.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, graphresult, outtext, settextstyle**

settextstyle

Function Sets the current text characteristics for graphics output.

Syntax

```
#include <graphics.h>
void far settextstyle(int font, int direction,
                                    int charsize);
```

Prototype in graphics.h

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*, defined in graphics.h, provides names for these different font settings (see following table).

Name	*Value*	*Description*
DEFAULT_FONT	0	8×8 bit-mapped font
TRIPLEX_FONT	1	stroked triplex font
SMALL_FONT	2	stroked small font
SANSSERIF_FONT	3	stroked sans-serif font
GOTHIC_FONT	4	stroked gothic 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 Appendix D of this manual.

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** will display characters from the 8×8 bit-mapped font in an 8×8 pixel rectangle on the screen.
- If *charsize* equals 2, these output functions will 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

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

settime

Function Sets system time.

Syntax #include <dos.h>
void settime(struct time **timep*);

Prototype in dos.h

Remarks **settime** sets the system time to the values in the **time** structure pointed to by *timep*.

The **time** structure is defined as follows:

```
struct time {
   unsigned char ti_min;                        /* minutes */
   unsigned char ti_hour;                         /* hours */
   unsigned char ti_hund;       /* hundredths of seconds */
   unsigned char ti_sec;                        /* seconds */
};
```

Return value None.

Portability **settime** is unique to DOS.

See also **ctime, getdate, gettime, setdate, time**

setusercharsize

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

Syntax #include <graphics.h>
void far setusercharsize(int *multx*, int *divx*,
int *multy*, int *divy*);

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, graphresult, settextstyle**

Example

```
#include <graphics.h>
#include <conio.h>

main()
{
   /* will request autodetection */
   int graphdriver = DETECT, graphmode;
   char *title = "TEXT in a BOX";
   /* initialize graphics */
   initgraph(&graphdriver, &graphmode, "");

   /* Draw a rectangle and fit a text string inside */
   settextjustify(CENTER_TEXT, CENTER_TEXT);
   setusercharsize(1,1,1,1);
   settextstyle(TRIPLEX_FONT, HORIZ_DIR, USER_CHAR_SIZE);
   setusercharsize(200, textwidth(title), 100,
                   textheight(title));
   rectangle(0, 0, 200, 100);
   outtextxy(100, 50, title);
   getche();
   closegraph();
}
```

setvbuf

Function Assigns buffering to a stream.

Syntax

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type,
            size_t size);
```

Prototype in stdio.h

Remarks

setvbuf causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given stream is opened.

If *buf* is null, a buffer will be allocated using **malloc**; the buffer will use *size* as the amount allocated. The *size* parameter specifies the buffer size and must be greater than zero.

Note: The parameter *size* is limited to a maximum of 32767.

stdin and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered.

Unbuffered means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

The *type* parameter is one of the following:

_IOFBF The file is *fully buffered*. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.

_IOLBF The file is *line buffered*. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file.

_IONBF The file is *unbuffered*. The *buf* and *size* parameters are ignored. Each input

operation will read directly from the file, and each output operation will immediately write the data to the file.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return value **setvbuf** returns 0 on success. It returns nonzero if an invalid value is given for *type* or *size*, or if there is not enough space to allocate a buffer.

Portability **setvbuf** is available on UNIX systems and is compatible with ANSI C.

See also **fflush, fopen, setbuf**

Example

```
#include <stdio.h>

main ()
{
     FILE *input, *output;
     char bufr[512];
     input = fopen("file.in", "r");
     output = fopen("file.out", "w");

     /* Set up the input stream for minimal disk access,
        using our own character buffer */
     if (setvbuf(input, bufr, _IOFBF, 512) != 0)
        printf("failed to set up buffer for input file\n");
     else
        printf("buffer set up for input file\n");

     /* Set up the output stream for line buffering using
        space that will be obtained through an indirect
        call to malloc */
     if (setvbuf(output, NULL
, _IOLBF, 132) != 0)
        printf("failed to set up buffer for output file\n");
     else
        printf("buffer set up for output file\n");

     /* Perform file I/O here */

     /* Close files */
     fclose(input);
     fclose(output);
}]
```

setvect

Function Sets interrupt vector entry.

Syntax

```
void setvect(int interruptno,
             void interrupt (*isr) ( ));
```

Prototype in dos.h

Remarks Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

setvect sets the value of the interrupt vector named by *interruptno* to a new value, *isr*, which is a far pointer containing the address of a new interrupt function. The address of a C routine can only be passed to *isr* if that routine is declared to be an *interrupt routine.*

Note: If you use the prototypes declared in dos.h, you can simply pass the address of an interrupt function to **setvect** in any memory model.

Return value None.

Portability **setvect** is unique to the 8086 family of processors.

See also **getvect**

setverify

Function Sets the state of the verify flag in DOS.

Syntax void setverify(int *value*);

Prototype in dos.h

Remarks **setverify** sets the current state of the verify flag to *value.*

- A *value* of 0 = verify flag off.
- A *value* of 1 = verify flag on.

The verify flag controls output to the disk. When verify is off, writes are not verified; when verify is on, all disk writes are verified to ensure proper writing of the data.

Return value None.

Portability **setverify** is unique to DOS.

See also getverify

setviewport

Function Sets the current viewport for graphics output.

Syntax
```
#include <graphics.h>
void far setviewport(int left, int top, int right,
                     int bottom, int clip);
```

Prototype in graphics.h

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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **clearviewport, getviewsettings, graphresult**

setvisualpage

Function Sets the visual graphics page number.

Syntax
```
#include <graphics.h>
void far setvisualpage(int page);
```

Prototype in graphics.h

Remarks **setvisualpage** makes *page* the visual graphics page.

Return value None.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **graphresult, setactivepage**

Example See **setactivepage**

---

setwritemode

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

Syntax #include <graphics.h>
void far setwritemode(int *mode*);

Prototype in graphics.h

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 on the screen. 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **drawpoly, line, linerel, lineto, putimage**

signal

Function Specifies signal-handling actions.

Syntax

```
#include <signal.h>
void (*signal(int sig, void (*func)
        (int sig[, int subcode])))(int);
```

Prototype in signal.h

Remarks **signal** determines how receipt of signal number *sig* will subsequently be treated. You can install a user-specified handler routine or use one of the two predefined handlers in signal.h.

The two predefined handlers follow:

Function Pointer	Meaning
SIG_DFL	Terminates the program.
SIG_IGN	Ignore this type signal.

A third predefined handler, defined in signal.h, is SIG_ERR. It is used to indicate an error return from **signal**.

The signal types and their defaults are as follows:

Signal Type	Meaning
SIGABRT	Abnormal termination. Default action is equivalent to calling **_exit**(3).
SIGFPE	Arithmetic error caused by division by 0, invalid operation, and the like. Default action is equivalent to calling **_exit**(1).
SIGILL	Illegal operation. Default action is equivalent to calling **_exit**(1).
SIGINT	CTRL-C interrupt. Default action is to do an INT 23H.
SIGSEGV	Illegal storage access. Default action is equivalent to calling **_exit**(1).
SIGTERM	Request for program termination. Default action is equivalent to calling **_exit**(1).

signal.h defines a type called *sig_atomic_t,* the largest integer type the processor can load or store atomically in the presence of asynchronous interrupts (for the 8086 family, this is a 16-bit word; that is, a Turbo C integer).

When a signal is generated by the **raise** function or by an external event, the following happens:

1. If a user-specified handler has been installed for the signal, the action for that signal type is set to SIG_DFL.
2. The user-specified function is called with the signal type as the parameter.

User-specified handler functions can terminate by a return or by a call to **abort**, **_exit**, **exit**, or **longjmp**.

Turbo C implements an extension to ANSI C when the signal type is SIGFPE, SIGSEGV, or SIGILL. The user-specified handler function is called with one or two extra parameters. If SIGFPE, SIGSEGV, or SIGILL has

been raised as the result of an explicit call to the **raise** function, the user-specified handler is called with one extra parameter, an integer specifying that the handler is being explicitly invoked. The explicit activation values for SIGFPE, SIGSEGV and SIGILL are as follows (see declarations in float.h):

Signal Type	Activation Value
SIGFPE	FPE_EXPLICITGEN
SIGSEGV	SEGV_EXPLICITGEN
SIGILL	ILL_EXPLICITGEN

If SIGFPE is raised because of a floating-point exception, the user handler is called with one extra parameter that specifies the FPE_xxx type of the signal. If SIGSEGV, SIGILL or the integer-related variants of SIGFPE signals (FPE_INTOVFLOW or FPE_INTDIV0) are raised as the result of a processor exception, the user handler is called with two extra parameters:

1. The SIGFPE, SIGSEGV or SIGILL exception type (see float.h for all these types). This first parameter is the usual ANSI signal type.
2. An integer pointer into the stack of the interrupt handler that called the user-specified handler. This pointer points to a list of the processor registers saved when the exception occured. The registers are in the same order as the parameters to an interrupt function; that is, BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS, FLAGS. To have a register value changed when the handler returns, change one of the locations in this list. For example, to have a new SI value on return, do something like this:

   ```
   *((int*)list_pointer + 2) = new_SI_value;
   ```

   In this way the handler can examine and make any adjustments to the registers that you want. (See Example 2 for a demonstration.)

The following SIGFPE type signals can occur (or be generated). They correspond to the exceptions that the

80x87 is capable of detecting, as well as the "INTEGER DIVIDE BY ZERO" and the "INTERRUPT ON OVERFLOW" on the main CPU. The declarations for these are in float.h.

SIGFPE Signal	Meaning
FPE_INTOVFLOW	INTO executed with OF flag set.
FPE_INTDIV0	Integer divide by zero.
FPE_INVALID	Invalid operation.
FPE_ZERODIVIDE	Division by zero.
FPE_OVERFLOW	Numeric overflow.
FPE_UNDERFLOW	Numeric underflow.
FPE_INEXACT	Precision.
FPE_EXPLICITGEN	User program executed **raise**(SIGFPE).

Note: The FPE_INTOVFLOW and FPE_INTDIV0 signals are generated by integer operations, and the others are generated by floating-point operations. Whether the floating-point exceptions are generated depends on the coprocessor control word, which can be modified with **_control87**. Denormal exceptions are handled by Turbo C and not passed to a signal handler.

The following SIGSEGV type signals can occur:

SIGSEGV Signal	Meaning
SEGV_BOUND	Bound constraint exception.
SEGV_EXPLICITGEN	**raise**(SIGSEGV) was executed.

Note: The 8088 and 8086 processors *don't* have a bound instruction. The 186, 286, 386, and NEC V series processors *do* have this instruction. So, on the 8088 and 8086 processors, the SEGV_BOUND type of SIGSEGV signal won't occur. Turbo C doesn't generate bound instructions, but they can be used in inline code and seperately compiled assembler routines that are linked in.

The following SIGILL type signals can occur:

SIGILL Signal	Meaning
ILL_EXECUTION	Illegal operation attempted.
ILL_EXPLICITGEN	**raise**(SIGILL) was executed.

Note: The 8088, 8086, NEC V20, and NEC V30 processors *don't* have an illegal operation exception. The 186, 286, 386, NEC V40, and NEC V50 processors *do* have this exception type. So, on 8088, 8086, NEC V20, and NEC V30 processors the ILL_EXECUTION type of SIGILL won't occur.

Note: When the signal type is SIGFPE, SIGSEGV, or SIGILL, a return from a signal handler is generally not advisable because the state of the 8087 is corrupt, the results of an integer division are wrong, an operation that shouldn't have overflowed did, a bound instruction failed, or an illegal operation was attempted. The only time a return is reasonable is when the handler alters the registers so that a reasonable return context exists *or* the signal type indicates that the signal was generated explicitly (for example, FPE_EXPLICITGEN, SEGV_EXPLICITGEN, or ILL_EXPLICITGEN). Generally in this case you would print an error message and terminate the program via **_exit**, **exit**, or **abort**. If a return is executed under any other conditions, the program's action will probably be unpredictable upon resuming.

Return value If the call succeeds, **signal** returns a pointer to the previous handler routine for the specified signal type. If the call fails, **signal** returns SIG_ERR, and the external variable *errno* is set to EINVAL.

Portability **signal** is compatible with ANSI C.

See also **abort, _control87, ctrlbrk, exit, longjmp, raise, setjmp**

Example

```
/* This example installs a signal handler routine to be run
when a Ctrl-Break is hit. */

#include <stdio.h>
#include <signal.h>
```

```
void Catcher(int sig)
{
   printf("\nNow in break routine\n");
   exit(1);
}

main()
{
   signal(SIGINT, Catcher);
   for (;;)
      printf("\nIn main() program\n");
}
```

Example 2

```
/* This example installs a signal handler routine for SIGFPE,
   catches an integer overflow condition, makes an adjustment
   to AX register, and returns. */

#pragma inline
#include <stdio.h>
#include <signal.h>

void Catcher(int sig, int type, int *reglist)
{
   printf("Caught it!\n");
   *(reglist + 8) = 3; /* make return AX = 3 */
}

main()
{
   signal(SIGFPE, Catcher);
   asm     mov     ax,07FFFH          /* AX = 32767       */
   asm     inc     ax                 /* cause overflow   */
   asm     into                       /* activate handler */

/* The handler set AX to 3 on return. If that hadn't
   happened, there would have been another exception when
   the next 'into' was executed after the 'dec'
   instruction. */

   asm     dec     ax                 /* no overflow now  */
   asm     into                       /* doesn't activate */
}
```

sin

Function Calculates sine.

Syntax

```
#include <math.h>
double sin(double x);
```

Prototype in math.h

Remarks **sin** computes the sine of the input value. Angles are specified in radians.

Error-handling for this routine can be modified through the function **matherr**.

Return value **sin** returns the sine of the input value.

Portability **sin** is available on UNIX systems and is compatible with ANSI C.

See also **acos, asin, atan, atan2, cos, cosh, tan, tanh**

sinh

Function Calculates hyperbolic sine.

Syntax

```
#include <math.h>
double sinh(double x);
```

Prototype in math.h

Remarks **sinh** computes the hyperbolic sine for a real argument.

Error-handling for **sinh** can be modified through the function **matherr**.

Return value **sinh** returns the hyperbolic sine of *x*.

When the correct value would overflow, **sinh** returns the value HUGE_VAL of appropriate sign.

Portability **sinh** is available on UNIX systems and is compatible with ANSI C.

See also **acos, asin, atan, atan2, cos, cosh, sin, tan, tanh**

sleep

Function Suspends execution for an interval (seconds).

Syntax void sleep(unsigned *seconds*);

Prototype in dos.h

Remarks With a call to **sleep**, the current program is suspended from execution for the number of seconds specified by the argument *seconds*. The interval is only accurate to the nearest hundredth of a second or the accuracy of the DOS clock, whichever is less accurate.

Return value None.

Portability **sleep** is available on UNIX systems.

See also **delay**

sopen

Function Opens a shared file.

Syntax

```
#include <fcntl.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>
int sopen(char *path, int access,
      int shflag, int mode);
```

Prototype in io.h

Remarks **sopen** opens the file given by *path* and prepares it for shared reading and/or writing, as determined by *access*, *shflag*, and *mode*.

sopen is a macro defined as

```
open(path, (access) | (shflag), mode)
```

For **sopen**, *access* is constructed by ORing flags bitwise from the following two lists. Only one flag from the first list can be used; the remaining flags can be used in any logical combination.

List 1: Read/Write Flags

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

List 2: Other Access Flags

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits, as in **chmod**.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	This flag can be given to explicitly open the file in binary mode.
O_TEXT	This flag can be given to explicitly open the file in text mode.

These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing *access*, you need to supply the *mode* argument to **sopen**, from the following symbolic constants defined in sys\stat.h.

Value of *permiss*	**Access Permission**
S_IWRITE	permission to write
S_IREAD	permission to read
S_IREAD | S_IWRITE	permission to read/write

shflag specifies the type of file-sharing allowed on the file *path*. Symbolic constants for *shflag* are defined in share.h.

Value of *shflag*	What It Does
SH_COMPAT	sets compatibility mode
SH_DENYRW	denies read and write access
SH_DENYWR	denies write access
SH_DENYRD	denies read access
SH_DENYNONE	permits read/write access
SH_DENYNO	permits read/write access

Return value

On successful completion, **sopen** returns a non-negative integer (the file handle), and the file pointer (that marks the current position in the file) is set to the beginning of the file. On error, it returns –1, and *errno* is set to one of the following:

ENOENT	Path or file function not found
EMFILE	Too many open files
EACCES	Permission denied
EINVACC	Invalid access code

Portability

sopen is available on UNIX systems. On UNIX version 7, the O_type mnemonics are not defined. UNIX System III uses all the O_type mnemonics except O_BINARY.

See also

chmod, close, creat, lock, lseek, _open, open, unlock, unmask

sound

Function

Turns PC speaker on at specified frequency.

Syntax

void sound(unsigned *frequency*);

Prototype in

dos.h

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.**

Portability

sound works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also delay, nosound

Example

```
/* Emits a 7-Hz tone for 10 seconds.
   True story: 7 Hz is the resonant frequency of a chicken's
   skull cavity. This was determined empirically in Australia,
   where a new factory generating 7-Hz tones was located too
   close to a chicken ranch: When the factory started up,
   all the chickens died.

   Your PC may not be able to emit a 7-Hz tone. */

main()
{
   sound(7);
   delay(10000);
   nosound();
}
```

spawn...

Function Creates and runs child processes.

Syntax

```
#include <process.h>
#include <stdio.h>
int spawnl(int mode, char *path,
          char *arg0, arg1, ..., argn, NULL);

int spawnle(int mode, char *path,
          char *arg0, arg1, ..., argn, NULL, char
          *envp[]);

int spawnlp(int mode, char *path,
          char *arg0, arg1, ..., argn, NULL);

int spawnlpe(int mode, char *path,
          char *arg0, arg1, ..., argn, NULL, char
          *envp[]);

int spawnv(int mode, char *path, char *argv[]);
int spawnve(int mode, char *path,
          char *argv[], char *envp[]);

int spawnvp(int mode, char *path, char *argv[]);
int spawnvpe(int mode, char *path,
          char *argv[], char *envp[]);
```

Prototype in process.h

Remarks The functions in the **spawn...** family create and run (execute) other files, known as *child processes*. There must be sufficient memory available for loading and executing a child process.

The value of *mode* determines what action the calling function (the *parent process*) will take after the spawn call. The possible values of *mode* are

P_WAIT	Puts parent process "on hold" until child process completes execution.
P_NOWAIT	Continues to run parent process while child process runs.
P_OVERLAY	Overlays child process in memory location formerly occupied by parent. Same as an **exec...** call.

Note: P_NOWAIT is currently not available; using it will generate an error value.

path is the file name of the called child process. The **spawn...** function calls search for *path* using the standard DOS search algorithm:

- No extension or no period: Search for exact file name; if not successful, add .EXE and search again.
- Extension given: Search only for exact file name.
- Period given: Search only for file name with no extension.
- If *path* does not contain an explicit directory, **spawn...** functions that have the *p* suffix will search the current directory, then the directories set with the DOS PATH environment variable.

The suffixes *l*, *v*, *p*, and *e* added to the **spawn...** "family name" specify that the named function will operate with certain capabilities.

p The function will search for the file in those directories specified by the PATH environment variable. Without the *p* suffix, the function will search only the current working directory.

l The argument pointers *arg0*, *arg1*, ..., *argn* are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.

v The argument pointers *argv[0]*, ..., *arg[n]* are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.

e The argument *envp* can be passed to the child process, allowing you to alter the environment for the child

process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the **spawn...** family *must* have one of the two argument-specifying suffixes (either *l* or *v*). The *path search* and *environment inheritance* suffixes (*p* and *e*) are optional.

For example:

- **spawnl** takes separate arguments, searches only the current directory for the child, and passes on the parent's environment to the child.
- **spawnvpe** takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *envp* argument for altering the child's environment.

The **spawn...** functions must pass at least one argument to the child process (*arg0* or *argv[0]*): This argument is, by convention, a copy of *path*. (Using a different value for this 0th argument won't produce an error.)

Under DOS 3.x, *path* is available for the child process; under earlier versions, the child process cannot use the passed value of the 0th argument (*arg0* or *argv[0]*).

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1*, ..., *argn* point to character strings that form the new list of arguments. A mandatory null following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *envp*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

envvar = *value*

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *envp[]* is null. When *envp* is null, the child inherits the parents' environment settings.

The combined length of *arg0* + *arg1* + ... + *argn* (or of *argv[0]* + *argv[1]* + ... + *argv[n]*), including space char-

acters that separate the arguments, must be < 128 bytes. Null-terminators are not counted.

When a **spawn...** function call is made, any open files remain open in the child process.

Return value

On a successful execution, the **spawn...** functions return the child process's exit status (0 for a normal termination). If the child specifically calls **exit** with a non-zero argument, its exit status can be set to a non-zero value.

On error, the **spawn...** functions return –1, and *errno* is set to one of the following:

E2BIG	Arg list too long
EINVAL	Invalid argument
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough core

See also

abort, atexit, _exit, exit, exec..., _fpreset, searchpath, system

Example

```
/* This program is SPAWNFAM.C.

/* To run this example, you must first compile CHILD.C to an
   .EXE file. */

#include <stdio.h>
#include <process.h>

status(int val)
{
   if (val == -1)
      printf("failed to start child process\n");
   else
      if (val > 0)  printf("child terminated abnormally\n");

}

main()
{
   /* NOTE: These environment strings should be changed
      to work on your computer. */

   /* create an environment string */
   char *envp[] =  { "PATH=C:\\",
                     "DUMMY=YES",
                   };

   /* create a pathname */
   char *pathname = "C:\\CHILDREN\\CHILD.EXE";
```

```
   /* create an argument string */
   char *args[] =  { "CHILD.EXE",
                     "1st",
                     "2nd",
                      NULL
                   };

   printf("SPAWNL:\n");
   status(spawnl(P_WAIT, pathname,args[0], args[1], NULL));

   printf("\nSPAWNV:\n");
   status(spawnv(P_WAIT, pathname, args));

   printf("\nSPAWNLE:\n");
   status(spawnle(P_WAIT, pathname,
          args[0], args[1], NULL, envp));

   printf("\nSPAWNVPE:\n");
   status(spawnvpe(P_WAIT, pathname, args, envp));
}

/* This is CHILD.C -- the child process for SPAWNFAM.C. */

#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
   int i;
   char *path, *dummy;

   path = getenv("PATH");
   dummy = getenv("DUMMY");

   for (i = 0; i < argc; i++)
      printf("argv[%d] %s\n", i, argv[i]);

   if (path)
      printf("PATH = %s\n", path);

   if (dummy)
      printf("DUMMY = %s\n", dummy);

   /* return to parent with error code 0 */
   exit(0);
}
```

sprintf

Function Writes formatted output to a string.

Syntax

```
int sprintf(char *buffer,
    const char *format[, argument, ...]);
```

Prototype in stdio.h

Remarks **sprintf** accepts a series of arguments, applies to each a format specification contained in the format string pointed to by *format*, and outputs the formatted data to a string.

sprintf applies the first format specification to the first argument, the second to the second, and so on. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification.

Return value **sprintf** returns the number of bytes output. **sprintf** does not include the terminating null byte in the count. In the event of error, **sprintf** returns EOF.

Portability **sprintf** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **fprintf**, **printf**

Example See **printf**

sqrt

Function Calculates the positive square root of input value.

Syntax

```
#include <math.h>
double sqrt(double x);
```

Prototype in math.h

Remarks **sqrt** calculates the positive square root of the input value.

	Error-handling for **sqrt** can be modified through the function **matherr**.
Return value	On success, **sqrt** returns the value calculated, the positive square root of *x*.
	If *x* is negative, *errno* is set to
	EDOM Domain error
	and **sqrt** returns 0.
Portability	**sqrt** is available on UNIX systems and is compatible with ANSI C.
See also	**exp, log, pow**

srand

Function	Initializes random-number generator.
Syntax	void srand(unsigned *seed*);
Prototype in	stdlib.h
Remarks	The random-number generator is reinitialized by calling **srand** with an argument value of 1. It can be set to a new starting point by calling **srand** with a given *seed* number.
Return value	None.
Portability	**srand** is available on UNIX systems and is compatible with ANSI C.
See also	**rand, random, randomize**
Example	See **rand**

sscanf

Function	Scans and formats input from a string.
Syntax	int sscanf(const char **buffer*, const char **format*[, *address*, ...]);
Prototype in	stdio.h

Remarks

sscanf scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specification passed to **sscanf** in the format string pointed to by *format*. Finally, **sscanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

sscanf may stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value

sscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If **sscanf** attempts to read at end-of-string, the return value is EOF.

If no fields were stored, the return value is 0.

Portability

sscanf is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also

fscanf, scanf

---

stat

Function

Gets information about an open file.

Syntax

#include <sys\stat.h>
int stat(char **path*, struct stat **statbuf*)

Prototype in

sys\stat.h

Remarks

stat stores information about a given file or directory in the **stat** structure.

statbuf points to the **stat** structure (defined in sys\stat.h). That structure contains the following fields:

st_mode	bit mask giving information about the file's mode
st_dev	drive number of disk containing the file
st_rdev	same as *st_dev*
st_nlink	set to the integer constant 1
st_size	size of the file, in bytes
st_atime	most recent time the file was modified
st_mtime	same as *st_atime*
st_ctime	same as *st_atime*

The **stat** structure contains three additional fields not mentioned here; they contain values that are not meaningful under DOS.

The bit mask that gives information about the mode of the file includes the following bits.

One of the following bits will be set:

S_IFREG	Set if an ordinary file is specified by *path*.
S_IFDIR	Set if *path* specifies a directory.

One or both of the following bits will be set:

S_IWRITE	Set if user has permission to write to file.
S_IREAD	Set if user has permission to read to file.

The bit mask contains user-execute bits; these are set according to the open file's extension. The bit mask also includes the read/write bits; these are set according to the file's permission mode.

Return value

stat returns 0 if it successfully retrieves the information about the file. On error (failure to get the information), **stat** returns –1 and sets *errno* to

ENOENT	File or path not found

See also

access, chmod, fstat, stat

_status87

Function Gets floating-point status.

Syntax unsigned int _status87(void);

Prototype in float.h

Remarks **_status87** gets the floating-point status word, which is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler.

Return value The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by **_status87**.

See also **_clear87, _control87, _fpreset**

Example See **_control87**

stime

Function Sets system date and time.

Syntax #include <time.h>
int stime(time_t **tp*);

Prototype in time.h

Remarks **stime** sets the system time and date. *tp* points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970.

Return value **stime** returns a value of 0.

Portability **stime** is available on UNIX systems.

See also **asctime, ftime, gettime, gmtime, localtime, time, tzset**

stpcpy

Function Copies one string into another.

Syntax char *stpcpy(char *dest, const char *src);

Prototype in string.h

Remarks **stpcpy** copies the string *src* to *dest*, stopping after the terminating null character has been reached.

Return value **stpcpy** returns *dest* + **strlen**(*src*).

Portability **stpcpy** is available on UNIX systems.

strcat

Function Appends one string to another.

Syntax char *strcat(char *dest, const char *src);

Prototype in string.h

Remarks **strcat** appends a copy of *src* to the end of *dest*. The length of the resulting string is **strlen**(*dest*) + **strlen**(*src*).

Return value **strcat** returns a pointer to the concatenated strings.

Portability **strcat** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

strchr

Function Scans a string for the first occurrence of a given character.

Syntax char *strchr(const char *s, int c);

Prototype in string.h

Remarks **strchr** scans a string in the forward direction, looking for a specific character. **strchr** finds the *first* occurrence of the character *c* in the string *s*.

The null-terminator is considered to be part of the string, so that, for example,

```
strchr(strs,0)
```

returns a pointer to the terminating null character of the string *strs*.

Return value **strchr** returns a pointer to the first occurrence of the character *c* in *s*; if *c* does not occur in *s*, **strchr** returns null.

Portability **strchr** is available on UNIX systems and is compatible with ANSI C.

strcmp

Function Compares one string to another.

Syntax int strcmp(const char **s1*, const char **s2*);

Prototype in string.h

Remarks **strcmp** performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return value **strcmp** returns a value that is

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

Portability **strcmp** is available on UNIX systems and is compatible with ANSI C.

strcmpi

Function Compares one string to another, without case sensitivity.

Syntax #include <string.h>
int strcmpi(const char **s1*, const char **s2*);

Prototype in string.h

Remarks **strcmpi** performs an unsigned comparison of *s1* to *s2*, without case sensitivity (same as **stricmp**—implemented as a macro).

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routine **strcmpi** is the same, respectively, as **stricmp**. **strcmpi** is implemented via a macro in string.h and translates calls from **strcmpi** to **stricmp**. Therefore, in order to use **strcmpi**, you must `#include` the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return value **strcmpi** returns an **int** value that is

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

strcpy

Function Copies one string into another.

Syntax char* strcpy(char **dest*, const char **src*);

Prototype in string.h

Remarks copies string *src* to *dest*, stopping after the terminating null character has been moved.

Return value **strcpy** returns *dest*.

Portability **strcpy** is available on UNIX systems and is compatible with ANSI C.

strcspn

Function Scans a string for the initial segment not containing any subset of a given set of characters.

Syntax #include <string.h>
size_t strcspn(const char **s1*, const char **s2*);

Prototype in string.h

Return value **strcspn** returns the length of the initial segment of string *s1* that consists entirely of characters *not* from string *s2*.

Portability **strcspn** is available on UNIX systems and is compatible with ANSI C.

strdup

Function Copies a string into a newly-created location.

Syntax char *strdup(const char **s*);

Prototype in string.h

Remarks **strdup** makes a duplicate of string *s*, obtaining space with a call to **malloc**. The allocated space is (**strlen**(*s*) + 1) bytes long. The user is responsible for freeing the space allocated by **strdup** when it is no longer needed.

Return value **strdup** returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.

Portability **strdup** is available on UNIX systems.

See also **free**

_strerror

Function Returns a pointer to an error message string.

Syntax char *_strerror(const char **s*);

Prototype in string.h, stdio.h

Remarks

_strerror allows you to generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If *s* is null, the return value points to the most recently generated error message.
- If *s* is not null, the return value contains *s* (your customized error message), a colon, a space, the most-recently generated system error message, and a newline. *s* should be 94 characters or less.

_strerror is the same as **strerror** in version 1.0 of Turbo C.

Return value

_strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to **_strerror**.

See also

perror, strerror

strerror

Function

Returns a pointer to an error message string.

Syntax

char *strerror(int *errnum*);

Prototype in

stdio.h, string.h

Remarks

strerror takes an **int** parameter *errnum*, an error number, and returns a pointer to an error message string associated with *errnum*.

Return value

strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to **strerror**.

Portability

strerror is compatible with ANSI C.

See also

perror, _strerror

stricmp

Function Compares one string to another, without case sensitivity.

Syntax int stricmp(const char **s1*, const char **s2*);

Prototype in string.h

Remarks **stricmp** performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines **stricmp** and **strcmpi** are the same; **strcmpi** is implemented via a macro in string.h that translates calls from **strcmpi** to **stricmp**. Therefore, in order to use **strcmpi**, you must include the header file string.h for the macro to be available.

Return value **stricmp** returns an **int** value that is

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

strlen

Function Calculates the length of a string.

Syntax #include <string.h>
size_t strlen(const char **s*);

Prototype in string.h

Remarks **strlen** calculates the length of *s*.

Return value **strlen** returns the number of characters in *s*, not counting the null-terminating character.

Portability **strlen** is available on UNIX systems and is compatible with ANSI C.

strlwr

Function Converts uppercase letters in a string to lowercase.

Syntax char *strlwr(char *s);

Prototype in string.h

Remarks **strlwr** converts uppercase letters (*A*-*Z*) in string *s* to lowercase (*a*-*z*). No other characters are changed.

Return value **strlwr** returns a pointer to the string *s*.

See also **strupr**

strncat

Function Appends a portion of one string to another.

Syntax #include <string.h>
char* strncat(char *dest, const char *src,
size_t *maxlen*);

Prototype in string.h

Remarks **strncat** copies at most *maxlen* characters of *src* to the end of *dest* and then appends a null character. The maximum length of the resulting string is **strlen**(*dest*) + *maxlen*.

Return value **strncat** returns *dest*.

Portability **strncat** is available on UNIX systems and is compatible with ANSI C.

strncmp

Function Compares a portion of one string to a portion of another.

Syntax #include <string.h>
int strncmp(const char *s1, const char *s2,
size_t *maxlen*);

Prototype in string.h

Remarks

strncmp makes the same unsigned comparison as **strcmp**, but looks at no more than *maxlen* characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined *maxlen* characters.

Return value

strncmp returns a an **int** value based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

Portability

strncmp is available on UNIX systems and is compatible with ANSI C.

strncmpi

Function

Compares a portion of one string to a portion of another, without case sensitivity.

Syntax

```
#include <string.h>
int strncmpi(const char *s1,
             const char *s2, size_t n);
```

Prototype in

string.h

Remarks

strncmpi performs a signed comparison of *s1* to *s2*, for a maximum length of *n* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until *n* characters have been examined. The comparison is not case sensitive. (**strncmpi** is the same as **strnicmp**—implemented as a macro). It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines **strnicmp** and **strncmpi** are the same; **strncmpi** is implemented via a macro in string.h that translates calls from **strncmpi** to **strnicmp**. Therefore, in order to use **strncmpi**, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return value **strncmpi** returns an **int** value that is

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

strncpy

Function Copies a given number of bytes from one string into another, truncating or padding as necessary.

Syntax

```
#include <stdio.h>
char *strncpy(char *dest, const char *src,
              size_t maxlen);
```

Prototype in string.h

Remarks **strncpy** copies up to *maxlen* characters from *src* into *dest*, truncating or null-padding *dest*. The target string, *dest*, might not be null-terminated if the length of *src* is *maxlen* or more.

Return value **strncpy** returns *dest*.

Portability **strncpy** is available on UNIX systems and is compatible with ANSI C.

strnicmp

Function Compares a portion of one string to a portion of another, without case sensitivity.

Syntax

```
#include <string.h>
int strnicmp(const char *s1, const char *s2,
             size_t maxlen);
```

Prototype in string.h

Remarks **strnicmp** performs a signed comparison of *s1* to *s2*, for a maximum length of *maxlen* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

Return value **strnicmp** returns an **int** value that is

< 0	if *s1* is less than *s2*
== 0	if *s1* is the same as *s2*
> 0	if *s1* is greater than *s2*

---

strnset

Function Sets a specified number of characters in a string to a given character.

Syntax

```
#include <string.h>
char *strnset(char *s, int ch, size_t n);
```

Prototype in string.h

Remarks **strnset** copies the character *ch* into the first *n* bytes of the string *s* . If *n* > **strlen**(*s*), then **strlen**(*s*) replaces *n*. It stops when *n* characters have been set, or when a null character is found.

Return value **strnset** returns *s*.

---

strpbrk

Function Scans a string for the first occurrence of any character from a given set.

Syntax

```
char *strpbrk(const char *s1, const char *s2);
```

Prototype in string.h

Remarks **strpbrk** scans a string, *s1*, for the first occurrence of any character appearing in *s2*.

Return value **strpbrk** returns a pointer to the first occurrence of any of the characters in *s2*. If none of the *s2* characters occurs in *s1*, it returns null.

Portability **strpbrk** is available on UNIX systems and is compatible with ANSI C.

strrchr

Function Scans a string for the last occurrence of a given character.

Syntax char *strrchr(const char **s*, int *c*);

Prototype in string.h

Remarks **strrchr** scans a string in the reverse direction, looking for a specific character. **strrchr** finds the *last* occurrence of the character *c* in the string *s*. The null-terminator is considered to be part of the string.

Return value **strrchr** returns a pointer to the last occurrence of the character *c*. If *c* does not occur in *s*, **strrchr** returns null.

Portability **strrchr** is available on UNIX systems and is compatible with ANSI C.

strrev

Function Reverses a string.

Syntax char *strrev(char **s*);

Prototype in string.h

Remarks **strrev** changes all characters in a string to reverse order, except the terminating null character. (For example, it would change `string\0` to `gnirts\0`.)

Return value **strrev** returns a pointer to the reversed string. There is no error return.

strset

Function Sets all characters in a string to a given character.

Syntax char *strset(char **s*, int *ch*);

Prototype in string.h

Remarks **strset** sets all characters in the string *s* to the character *ch*. It quits when the terminating null character is found.

Return value **strset** returns *s*.

See also **setmem**

strspn

Function Scans a string for the first segment that is a subset of a given set of characters.

Syntax
```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Prototype in string.h

Remarks **strspn** finds the initial segment of string *s1* that consists entirely of characters from string *s2*.

Return value **strspn** returns the length of the initial segment of *s1* that consists entirely of characters from *s2*.

Portability **strspn** is available on UNIX systems and is compatible with ANSI C.

strstr

Function Scans a string for the occurrence of a given substring.

Syntax
```
char *strstr(const char *s1, const char *s2);
```

Prototype in string.h

Remarks **strstr** scans *s1* for the first occurrence of the substring *s2*.

Return value **strstr** returns a pointer to the element in *s1* where *s2* begins (points to *s2* in *s1*). If *s2* does not occur in *s1*, **strstr** returns null.

Portability **strstr** is available on UNIX systems and is compatible with ANSI C.

strtod

Function Converts a string to a **double** value.

Syntax

```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
```

Prototype in stdlib.h

Remarks **strtod** converts a character string, *s*, to a **double** value. *s* is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

where

[ws] = optional whitespace
[sn] = optional sign (+ or –)
[ddd] = optional digits
[fmt] = optional *e* or *E*
[.] = optional decimal point

strtod also recognizes +INF and –INF for plus and minus infinity, and +NAN and –NAN for Not-a-Number.

For example, here are some character strings that **strtod** can convert to **double**:

+ 1231.1981 *e*-1
502.85E2
+ 2010.952

strtod stops reading the string at the first character that cannot be interpreted as an appropriate part of a **double** value.

If *endptr* is not null, **strtod** sets **endptr* to point to the character that stopped the scan (**endptr* = *&stopper*).

Return value **strtod** returns the value of *s* as a **double**. In case of overflow, it returns HUGE_VAL.

Portability **strtod** is available on UNIX systems and is compatible with ANSI C.

See also **atof**

strtok

Function

Searches one string for tokens, which are separated by delimiters defined in a second string.

Syntax

char *strtok(char **s1*, const char **s2*);

Prototype in

string.h

Remarks

strtok considers the string *s1* to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string *s2*.

The first call to **strtok** returns a pointer to the first character of the first token in *s1* and writes a null character into *s1* immediately following the returned token. Subsequent calls with null for the first argument will work through the string *s1* in this way, until no tokens remain.

The separator string, *s2*, may be different from call to call.

Return value

strtok returns a pointer to the token found in *s1*. A null pointer is returned when there are no more tokens.

Portability

strtok is available on UNIX systems and is compatible with ANSI C.

Example

```
/*  strtok - This example demonstrates the use of strtok
   to parse dates. Note that in order to parse dates of
   varying formats (for example, 12/3/87; Dec.12,1987;
   January 15, 1988; 12-FEB-88, etc.), you must specify
   the delimiter string to contain either a period, space,
   comma, minus, or slash. Notice in the output that the
   delimiters are not returned. */

#include <stdio.h>
#include <string.h>

main ()
{
   char *ptr;
   ptr = strtok ("FEB.14,1988", ". ,-/" );
   while (ptr != NULL)
   {
      printf ("ptr = %s\n", ptr);
      ptr = strtok (NULL, ". ,-/" );
   }
```

```
}
```

Program output

```
ptr = FEB
ptr = 14
ptr = 1988
```

strtol

Function Converts a string to a **long** value.

Syntax long strtol(const char **s*, char ***endptr*, int *radix*);

Prototype in stdlib.h

Remarks **strtol** converts a character string, *s*, to a **long** integer value. *s* is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:

```
[ws] [sn] [0] [x] [ddd]
```

where

[ws] = optional whitespace
[sn] = optional sign (+ or –)
[0] = optional zero (0)
[x] = optional x or X
[ddd] = optional digits

strtol stops reading the string at the first character it doesn't recognize.

If *radix* is between 2 and 36, the long integer is expressed in base *radix*. If *radix* is 0, the first few characters of *s* determine the base of the value being converted.

First Character	Second Character	String Interpreted As
0	1 – 7	octal
0	x or X	hexadecimal
1 – 9		decimal

If *radix* is 1, it is considered to be an invalid value. If *radix* is less than 0 or greater than 36, it is considered to be an invalid value.

Any invalid value for *radix* causes the result to be 0 and sets the next character pointer **endptr* to the starting string pointer.

If the value in *s* is meant to be interpreted as octal, any character other than 0 to 7 will be unrecognized.

If the value in *s* is meant to be interpreted as decimal, any character other than 0 to 9 will be unrecognized.

If the value in *s* is meant to be interpreted as a number in any other base, then only the numerals and letters used to represent numbers in that base will be recognized. (For example, if *radix* equals 5, only 0 to 4 will be recognized; if *radix* equals 20, only 0 to 9 and *A* to *J* will be recognized.)

If *endptr* is not null, **strtol** sets **endptr* to point to the character that stopped the scan (**endptr* = &*stopper*).

Return value **strtol** returns the value of the converted string, or 0 on error.

Portability **strtol** is available on UNIX systems and is compatible with ANSI C.

See also **atoi, atol, strtoul**

strtoul

Function Converts a string to an **unsigned long** in the given radix.

Syntax unsigned long strtoul(const char **s*,
char ***endptr*, int *radix*);

Prototype in stdlib.h

Remarks **strtoul** operates the same as **strtol**, except that it converts a string, *str*, to an **unsigned long** value (whereas **strtol** converts to a **long**). Refer to the entry for **strtol** for more information.

Return value **strtoul** returns the converted value, an **unsigned long**, or 0 on error.

Portability **strtoul** is compatible with ANSI C.

See also **atol, strtol**

strupr

Function Converts lowercase letters in a string to uppercase.

Syntax char *strupr(char **s*);

Prototype in string.h

Remarks **strupr** converts lowercase letters (*a*-*z*) in string *s* to uppercase (*A*-*Z*). No other characters are changed.

Return value **strupr** returns *s*.

See also **strlwr**

swab

Function Swaps bytes.

Syntax void swab(char **from*, char **to*, int *nbytes*);

Prototype in stdlib.h

Remarks **swab** copies *nbytes* bytes from the *from* string to the *to* string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. *nbytes* should be even.

Return value None.

Portability **swab** is available on UNIX systems.

system

Function Issues a DOS command.

Syntax int system(const char **command*);

Prototype in stdlib.h, process.h

Remarks **system** invokes the DOS COMMAND.COM file to execute a DOS command, batch file, or other program named by the string *command*, from inside an executing C program.

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

The COMSPEC environment variable is used to find the COMMAND.COM file, so that file does not need to be in the current directory.

Return value **system** returns 0 on success, –1 on failure.

Portability **system** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

See also **exec..., _fpreset, searchpath, spawn...**

tan

Function Calculates the tangent.

Syntax

```
#include <math.h>
double tan(double x);
```

Prototype in math.h

Remarks **tan** calculates the tangent. Angles are specified in radians.

Error-handling for this routine can be modified through the function **matherr**.

Return value **tan** returns the tangent of *x*, any value for valid angles. For angles close to *pi*/2 or –*pi*/2, **tan** returns 0 and *errno* is set to

ERANGE	Result out of range

Portability **tan** is available on UNIX systems and is compatible with ANSI C.

See also **acos, asin, atan, atan2, cos, cosh, sin, sinh, tanh**

tanh

Function Calculates the hyperbolic tangent.

Syntax

```
#include <math.h>
double tanh(double x);
```

Prototype in math.h

Remarks **tanh** computes the hyperbolic tangent for real arguments.

Error-handling for **tanh** can be modified through the function **matherr**.

Return value **tanh** returns the hyperbolic tangent of *x*.

Portability **tanh** is available on UNIX systems and is compatible with ANSI C.

See also **acos, asin, atan, atan2, cos, cosh, sin, sinh, tan**

tell

Function Gets current position of file pointer.

Syntax

```
long tell(int handle);
```

Prototype in io.h

Remarks **tell** gets the current position of the file pointer associated with *handle* and expresses it as the number of bytes from the beginning of the file.

Return value **tell** returns the current file pointer position. A return of –1 (long) indicates an error, and *errno* is set to

EBADF	Bad file number

Portability **tell** is available on all UNIX systems.

See also **fgetpos, fseek, ftell, lseek**

textattr

Function Sets text attributes.

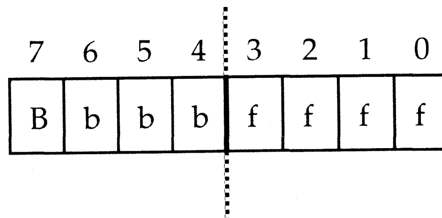
Syntax void textattr(int *newattr*);

Prototype in conio.h

Remarks **textattr** lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with **textcolor** and **textbackground**.)

This function does not affect any characters currently on the screen; it only affects those displayed by functions (such as **cprintf**) performing text mode, direct video output *after* this function is called.

The color information is encoded in the *newattr* parameter as follows:



In this 8-bit *newattr* parameter,

`ffff`	is the 4-bit foreground color (0 to 15).
`bbb`	is the 3-bit background color (0 to 7).
`B`	is the blink-enable bit.

If the blink-enable bit is on, the character will blink. This can be accomplished by adding the constant BLINK to the attribute.

If you use the symbolic color constants defined in conio.h for creating text attributes with **textattr**, note the following limitations on the color you select for the background:

- You can only select one of the first eight colors for the background.
- You must shift the selected background color left by 4 bits to move it into the correct bit positions.

These symbolic constants are listed in the table below:

Symbolic constant	*Numeric value*	*Foreground or background?*
BLACK	0	both
BLUE	1	both
GREEN	2	both
CYAN	3	both
RED	4	both
MAGENTA	5	both
BROWN	6	both
LIGHTGRAY	7	both
DARKGRAY	8	foreground only
LIGHTBLUE	9	foreground only
LIGHTGREEN	10	foreground only
LIGHTCYAN	11	foreground only
LIGHTRED	12	foreground only
LIGHTMAGENTA	13	foreground only
YELLOW	14	foreground only
WHITE	15	foreground only
BLINK	128	foreground only

Return value None.

Portability **textattr** works only on IBM PCs and compatible systems.

See also **gettextinfo, highvideo, lowvideo, normvideo, textbackground, textcolor**

Example

```
/* Select blinking yellow characters on a blue background */
textattr(YELLOW + (BLUE<<4) + BLINK);
cputs("Hello, world");
```

textbackground

Function Selects new text background color.

Syntax void textbackground(int *newcolor*);

Prototype in conio.h

Remarks **textbackground** selects the background text color.
The background color of all characters subsequently written by functions performing text mode, direct video output will be in the color given by *newcolor*, an integer from 0 to 7. You can give the color using one of the symbolic constants defined in conio.h. If you use these constants, you must include conio.h.

This function does not affect any characters currently on the screen, but only those displayed using direct console output (such as **cprintf**) after **textbackground** has been called.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

Symbolic Constant	*Numeric Value*
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7

Return value None.

Portability **textbackground** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **gettextinfo, textattr, textcolor**

Example

```
/* makes a magenta background */
```

```
textbackground(MAGENTA);
```

textcolor

Function Selects new character color in text mode.

Syntax #include <conio.h>
void textcolor(int *newcolor*);

Prototype in conio.h

Remarks **textcolor** selects the foreground character color. The foreground color of all characters subsequently written by the console output functions will be the color given by *newcolor*. You can give the color using a symbolic constant defined in conio.h. If you use these constants, you must include conio.h.

This function does not affect any characters currently on the screen, but only those displayed using direct console output (such as **cprintf)** after **textcolor** has been called.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

Symbolic Constant	*Numeric Value*
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

You can make the characters blink by adding 128 to the foreground color. The predefined constant BLINK exists for this purpose. For example,

```
textcolor(CYAN + BLINK);
```

Note: Some monitors do not recognize the intensity signal used to create the eight "light" colors (8-15). On such monitors, the light colors will be displayed as their "dark" equivalents (0-7). Also, systems that do not display in color may treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, and so on). Exactly what you'll see on such systems depends upon your own hardware.

Return value None.

Portability **textcolor** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **gettextinfo, highvideo, lowvideo, normvideo, textattr, textbackground**

textheight

Function Returns the height of a string in pixels.

Syntax
```
#include <graphics.h>
int far textheight(char far *textstring);
```

Prototype in graphics.h

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.

It is important to 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, outtext, outtextxy, setteststyle, textwidth**

textmode

Function Puts screen in text mode.

Syntax
```
void textmode(int newmode);
```

Prototype in conio.h

Remarks **textmode** selects a specific text mode.

You can give the text mode (the argument *newmode*) by using a symbolic constant from the enumeration type

text_modes (defined in conio.h). If you use these constants, you must include conio.h.

The *text_modes* type constants, their numeric values, and the modes they specify are given in the following table.

Symbolic constant	Numeric value	Text mode
LASTMODE	–1	previous text mode
BW40	0	black & white, 40 columns
C40	1	color, 40 columns
BW80	2	black & white, 80 columns
C80	3	color, 80 columns
MONO	7	monochrome, 80 columns

When **textmode** is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to **normvideo**.

Specifying LASTMODE to **textmode** causes the most recently selected text mode to be reselected. This feature is really only useful when you want to return to text mode after using a graphics mode.

textmode should be used only when the screen is in the text mode (presumably to change to a different text mode). This is the only context in which **textmode** should be used. When the screen is in graphics mode, you should use **restorecrtmode** instead to escape temporarily to text mode.

Return value None.

Portability **textmode** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **gettextinfo, window**

textwidth

Function Returns the width of a string in pixels.

Syntax
```
#include <graphics.h>
int far textwidth(char far *textstring);
```

Prototype in graphics.h

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.

It is important to 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.

Portability This function works only with IBM PCs and compatibles equipped with supported graphics display adapters.

See also **gettextsettings, outtext, outtextxy, settextstyle, textheight**

time

Function Gets time of day.

Syntax
```
#include <time.h>
time_t time(time_t *timer);
```

Prototype in time.h

Remarks **time** gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by *timer*, provided that *timer* is not a null pointer.

Return value **time** returns the elapsed time in seconds, as described.

Portability **time** is available on UNIX systems and is compatible with ANSI C.

See also **asctime, ctime, difftime, ftime, gettime, gmtime, localtime, settime, stime, tzset**

tmpfile

Function Opens a "scratch" file in binary mode.

Syntax
```
#include <stdio.h>
FILE *tmpfile(void);
```

Prototype in stdio.h

Remarks **tmpfile** creates a temporary binary file and opens it for update (*w* + *b*). The file is automatically removed when it's closed or when your program terminates.

Return value **tmpfile** returns a pointer to the stream of the temporary file created. If the file can't be created, **tmpfile** returns null.

Portability **tmpfile** is available on UNIX systems and is compatible with ANSI C.

tmpnam

Function Creates a unique file name.

Syntax
```
char *tmpnam(char *s);
```

Prototype in stdio.h

Remarks **tmpnam** creates a unique file name, which can safely be used as the name of a temporary file. **tmpnam** generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65535.

The parameter to **tmpnam**, *s*, is either null or a pointer to an array of at least *L_tmpnam* characters. *L_tmpnam* is defined in stdio.h. If *s* is null, **tmpnam** leaves the generated temporary file name in an internal static object and returns a pointer to that object. If *s* is not null,

tmpnam places its result in the pointed-to array, which must be at least *L_tmpnam* characters long, and returns *s*.

Note: If you do create such a temporary file with **tmpnam**, it is your responsibility to delete the file name (for example, with a call to **remove**). It is not deleted automatically.

Return value If *s* is null, **tmpnam** returns a pointer to an internal static object. Otherwise, **tmpnam** returns *s*.

Portability **tmpnam** is available on UNIX systems and is compatible with ANSI C.

toascii

Function Translates characters to ASCII format.

Syntax int toascii(int *c*);

Prototype in ctype.h

Remarks **toascii** is a macro that converts the integer *c* to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127.

Return value **toascii** returns the converted value of *c*.

Portability **toascii** is available on UNIX systems.

_tolower

Function Translates characters to lowercase.

Syntax

```
#include <ctype.h>
int _tolower(int ch);
```

Prototype in ctype.h

Remarks **_tolower** is a macro that does the same conversion as **tolower**, except that it should be used only when *ch* is known to be uppercase (*A-Z*).

To use **_tolower**, you must include ctype.h.

Return value **_tolower** returns the converted value of *ch* if it is uppercase; otherwise, the result is undefined.

Portability **_tolower** is available on UNIX systems.

tolower

Function Translates characters to lowercase.

Syntax int tolower(int *ch*);

Prototype in ctype.h

Remarks **tolower** is a function that converts an integer *ch* (in the range EOF to 255) to its lowercase (*a-z*) value (*if* it was uppercase (*A-Z*); all others are left unchanged.

Return value **tolower** returns the converted value of *ch* if it is uppercase; all others it returns unchanged.

Portability **tolower** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

_toupper

Function Translates characters to uppercase.

Syntax #include <ctype.h>
int _toupper(int *ch*);

Prototype in ctype.h

Remarks **_toupper** is a macro that does the same conversion as **toupper**, except that it should be used only when *ch* is known to be lowercase (*a-z*).

To use **_toupper**, you must include ctype.h.

Return value **_toupper** returns the converted value of *ch* if it is lowercase; otherwise, the result is undefined.

Portability **_toupper** is available on UNIX systems.

toupper

Function Translates characters to uppercase.

Syntax int toupper(int *ch*);

Prototype in ctype.h

Remarks **toupper** is a function that converts an integer *ch* (in the range EOF to 255) to its uppercase value (*A*-*Z*) (*if* it was lowercase (*a*-*z*); all others are left unchanged.

Return value **toupper** returns the converted value of *ch* if it is lowercase; it returns all others unchanged.

Portability **toupper** is available on UNIX systems and is compatible with ANSI C. It is defined in Kernighan and Ritchie.

tzset

Function Sets value of global variables *daylight*, *timezone*, and *tzname*.

Syntax #include <time.h>
void tzset(void)

Prototype in time.h

Remarks *tzset* sets the *daylight*, *timezone*, and *tzname* global variables based on the environment variable *TZ*. The library functions **ftime** and **localtime** use these global variables to correct Greenwich Mean Time (GMT) to whatever the local time zone is. The format of the *TZ* environment string is as follows :

```
TZ = zzz[+/-]d[d][lll]
```

zzz is a three-character string representing the name of the current time zone. All three characters are required. For example, the string "PST" could be used to represent Pacific Standard Time.

[+/−]*d*[*d*] is a required field containing an optionally signed number with 1 or more digits. This number is the local time zone's difference from GMT in hours. Positive

numbers adjust westward from GMT. Negative numbers adjust eastward from GMT. For example, the number 5 = EST, +8 = PST, and –1 = Continental Europe. This number is used in the calculation of the global variable *timezone*. *timezone* is the difference in seconds between GMT and the local time zone.

lll is an optional three-character field that represents the local time zone daylight savings time. For example, the string "PDT" could be used to represent Pacific Daylight Savings Time. If this field is present, it will cause the *daylight* global variable to be set nonzero. If this field is absent, *daylight* will be set to zero.

If the *TZ* environment string isn't present or isn't in the above form, a default *TZ* = "EST5EDT" is presumed for the purposes of assigning values to the global variables *daylight*, *timezone*, and *tzname*.

The global variable *tzname*[0] points to a three-character string with the value of the time zone name from the *TZ* environment string. The global variable *tzname*[1] points to a three-character string with the value of the daylight savings time zone name from the *TZ* environment string. If no daylight savings name is present, *tzname*[1] points to a null string.

Return value None.

Portability **tzset** is available on UNIX and XENIX systems.

See also **asctime, ctime, ftime, gmtime, localtime, stime, time**

Example

```
#include <time.h>
#include <stdlib.h>

main()
{
   time_t td;
   /* Pacific daylight savings */
   putenv("TZ=PST8PDT");
   tzset();
   /* get current time / date  */
   time(&td);
   printf("Current time = %s\n", asctime(localtime(&td)));
}
```

ultoa

Function Converts an **unsigned long** to a string.

Syntax

```
char *ultoa(unsigned long value,
            char *string, int radix);
```

Prototype in stdlib.h

Remarks **ultoa** converts *value* to a null-terminated string and stores the result in *string*. *value* is an **unsigned long**.

radix specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. **ultoa** performs no overflow-checking, and if *value* is negative and *radix* equals 10, it does not set the minus sign.

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). **ultoa** can return up to 33 bytes.

Return value **ultoa** returns *string*. There is no error return.

See also **itoa, ltoa**

ungetc

Function Pushes a character back into input stream.

Syntax

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Prototype in stdio.h

Remarks **ungetc** pushes the character *c* back onto the named input *stream*, which must be open for reading. This character will be returned on the next call to **getc** or **fread** for that *stream*. One character may be pushed back in all situations. A second call to **ungetc** without a call to **getc** will force the previous character to be forgotten. A call to **fflush, fseek, fsetpos,** or **rewind** erases all memory of any pushed-back characters.

Return value On success, **ungetc** returns the character pushed back; it returns EOF if the operation fails.

Portability **ungetc** is available on UNIX systems and is compatible with ANSI C.

See also **fgetc, getc, getchar**

ungetch

Function Pushes a character back to the keyboard buffer.

Syntax int ungetch(int *ch*);

Prototype in conio.h

Remarks **ungetch** pushes the character *ch* back to the console, causing *ch* to be the next character read. The **ungetch** function fails if it is called more than once before the next read.

Return value **ungetch** returns the character *ch* if it is successful. A return value of EOF indicates an error.

Portability **ungetch** is available on UNIX systems.

See also **getch, getche**

unixtodos

Function Converts date and time to DOS format.

Syntax

```
#include <dos.h>
void unixtodos(long time, struct date *d,
               struct time *t);
```

Prototype in dos.h

Remarks **unixtodos** converts the UNIX-format time given in *time* to DOS format and fills in the **date** and **time** structures pointed to by *d* and *t*.

Return value None.

Portability **unixtodos** is unique to DOS.

See also **dostounix**

unlink

Function Deletes a file.

Syntax int unlink(const char **filename*);

Prototype in dos.h, io.h, stdio.h

Remarks **unlink** deletes a file specified by *filename*. Any DOS drive, path, and file name can be used as a *filename*. Wildcards are not allowed.

Read-only files cannot be deleted by this call. To remove read-only files, first use **chmod** or **_chmod** to change the read-only attribute.

Return value On successful completion, **unlink** returns 0. On error, it returns –1, and *errno* is set to one of the following values:

ENOENT	Path or file name not found
EACCES	Permission denied

Portability **unlink** is available on UNIX systems.

See also **chmod, remove**

unlock

Function Releases file-sharing locks.

Syntax int unlock(int *handle*, long *offset*, long *length*);

Prototype in io.h

Remarks **unlock** provides an interface to the DOS 3.x file-sharing mechanism.

unlock removes a lock previously placed with a call to **lock**. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing.

Return value **unlock** returns 0 on success, –1 on error.

Portability **unlock** is unique to DOS 3.x. Older versions of DOS do not support this call.

See also **lock, sopen**

va_...

Function Implement a variable argument list.

Syntax

```
#include &lt;stdarg.h&gt;
void va_start(va_list param, lastfix);
type va_arg(va_list param, type);
void va_end(va_list param);
```

Prototype in stdarg.h

Remarks Some C functions, such as **vfprintf** and **vprintf,** take variable argument lists in addition to taking a number of fixed (known) parameters. The **va_...** macros provide a portable way to access these argument lists. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed.

The header file stdarg.h declares one type (*va_list*), and three macros (**va_start, va_arg,** and **va_end**).

va_list

This array holds information needed by **va_arg** and **va_end**. When a called function takes a variable argument list, it declares a variable *param* of type *va_list*.

va_start

This routine (implemented as a macro) sets *param* to point to the first of the variable arguments being passed to the function. **va_start** must be used before the first call to **va_arg** or **va_end**.

va_start takes two parameters: *param* and *lastfix*. (*param* is explained under *va_list* in the preceding paragraph; *lastfix* is the name of the last fixed parameter being passed to the called function.)

va_arg

This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable *param* to **va_arg** should be the same *param* that **va_start** initialized.

The first time **va_arg** is used, it returns the first argument in the list. Each successive time **va_arg** is used, it returns the next argument in the list. It does this by first de-referencing *param*, and then incrementing *param* to point to the following item. **va_arg** uses the *type* to both perform the de-reference and to locate the following item. Each successive time **va_arg** is invoked, it modifies *param* to point to the next argument in the list.

va_end

This macro helps the called function perform a normal return. **va_end** might modify *param* in such a way that it cannot be used unless **va_start** is re-called. **va_end** should be called after **va_arg** has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return value — **va_start** and **va_end** return no values; **va_arg** returns the current argument in the list (the one that *param* is pointing to).

Portability — **va_arg**, **va_start**, and **va_end** are available on UNIX systems.

See also — **v...printf, v...scanf**

Example

```
#include <stdio.h>
#include <stdarg.h>

/* calculate sum of a 0 terminated list */

void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;

   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0)
   {
      total += arg;
```

```
    }
    printf(msg, total);
}

main()
{
    sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
}
```

Program output

```
The total of 1+2+3+4 is 10
```

Example 2

```
#include <stdio.h>
#include <stdarg.h>

void error(char *format,...)
{
    va_list argptr;

    printf("error: ");
    va_start(argptr, format);
    vprintf(format, argptr);
    va_end(argptr);
}

main()
{
    int value = -1;

    error("this is just an error message\n");
    error("invalid value %d encountered\n", value);
}
```

Program output

```
error: this is just an error message
error: invalid value -1 encountered
```

vfprintf

Function Writes formatted output to a stream.

Syntax #include <stdio.h>
int vfprintf(FILE **stream*, const char **format*,
va_list arglist);

Prototype in stdio.h

Remarks

The **v...printf** functions are known as *alternate entry points* for the **...printf** functions. They behave exactly like their **...printf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vfprintf accepts a pointer to a series of arguments, applies to each argument a format specification contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification.

Return value

vfprintf returns the number of bytes output. In the event of error, **vfprintf** returns EOF.

Portability

vfprintf is available on UNIX System V, and it is compatible with ANSI C.

See also

printf, va_...

Example

See **printf**

---

vfscanf

Function

Scans and formats input from a stream.

Syntax

#include <stdio.h>
int vfscanf(FILE **stream*, const char **format*,
    va_list arglist);

Prototype in

stdio.h

Remarks

The **v...scanf** functions are known as *alternate entry points* for the **...scanf** functions. They behave exactly like their **...scanf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vfscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specification passed to **vfscanf** in the format string pointed to by *format*. Finally, **vfscanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the

same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

vfscanf may stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value — **vfscanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If **vfscanf** attempts to read at end-of-file, the return value is EOF.

Portability — **vfscanf** is available on UNIX system V.

See also — **fscanf, scanf, va_...**

vprintf

Function — Writes formatted output to *stdout*.

Syntax

```
#include <stdarg.h>
int vprintf(const char *format, va_list arglist);
```

Prototype in — stdio.h

Remarks — The **v...printf** functions are known as *alternate entry points* for the **...printf** functions. They behave exactly like their **...printf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vprintf accepts a pointer to a series of arguments, applies to each a format specification contained in the format string pointed to by *format*, and outputs the formatted data to *stdout*. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification.

Return value — **vprint** returns the number of bytes output. In the event of error, **vprint** returns EOF.

Portability — **vprintf** is available on UNIX System V and is compatible with ANSI C.

See also — **printf, va_...**

Example — See **printf**

vscanf

Function — Scans and formats input from *stdin*.

Syntax

```
#include <stdarg.h>
int vscanf(const char *format, va_list arglist);
```

Prototype in — stdio.h

Remarks — The **v...scanf** functions are known as *alternate entry points* for the **...scanf** functions. They behave exactly like their **...scanf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vscanf scans a series of input fields, one character at a time, reading from *stdin*. Then each field is formatted according to a format specification passed to **vscanf** in the format string pointed to by *format*. Finally, **vscanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

vscanf may stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value — **vscanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If **vscanf** attempts to read at end-of-file, the return value is EOF.

Portability **vscanf** is available on UNIX system V.

See also **fscanf, scanf, va_...**

vsprintf

Function Writes formatted output to a string.

Syntax

```
#include <stdarg.h>
int vsprintf(char *buffer, const char *format,
             va_list arglist);
```

Prototype in stdio.h

Remarks The **v...printf** functions are known as *alternate entry points* for the **...printf** functions. They behave exactly like their **...printf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vsprintf accepts a pointer to a series of arguments, applies to each a format specification contained in the format string pointed to by *format*, and outputs the formatted data to a string. There must be the same number of format specifications as arguments.

See **printf** for a description of the information included in a format specification.

Return value **vsprintf** returns the number of bytes output. In the event of error, **vsprintf** returns EOF.

Portability **vsprintf** is available on UNIX System V and is compatible with ANSI C.

See also **printf, va_...**

Example See **printf**

vsscanf

Function Scans and formats input from a stream.

Syntax

```
#include <stdarg.h>
int vsscanf(const char *buffer, const char *format,
            va_list arglist);
```

Prototype in stdio.h

Remarks The **v...scanf** functions are known as *alternate entry points* for the **...scanf** functions. They behave exactly like their **...scanf** counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vsscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specification passed to **vsscanf** in the format string pointed to by *format*. Finally, **vsscanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifications and addresses as there are input fields.

See **scanf** for a description of the information included in a format specification.

vsscanf may stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it may terminate entirely, for a number of reasons. See **scanf** for a discussion of possible causes.

Return value **vsscanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If **vsscanf** attempts to read at end-of-string, the return value is EOF.

Portability **vsscanf** is available on UNIX system V.

See also **fscanf, scanf, va_...**

wherex

Function Gives horizontal cursor position within window.

Syntax int wherex(void);

Prototype in conio.h

Remarks **wherex** returns the *x*-coordinate of the current cursor position (within the current text window).

Return value **wherex** returns an integer in the range 1 to 80.

Portability **wherex** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **gettextinfo, gotoxy, wherey**

Example

```
printf("The cursor is at (%d,%d)\n", wherex(),wherey());
```

wherey

Function Gives vertical cursor position within window.

Syntax int wherey(void);

Prototype in conio.h

Remarks **wherey** returns the *y*-coordinate of the current cursor position (within the current text window).

Return value **wherey** returns an integer in the range 1 to 25.

Portability **wherey** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **gettextinfo, gotoxy, wherex**

Example See **wherex**

window

Function Defines active text mode window.

Syntax void window(int *left*, int *top*, int *right*, int *bottom*);

Prototype in conio.h

Remarks **window** defines a text window on the screen. If the coordinates are in any way invalid, the call to **window** is ignored.

left and *top* are the screen coordinates of the upper left corner of the window.

right and *bottom* are the screen coordinates of the lower right corner.

The minimum size of the text window is 1 column by 1 line. The default window is full screen, with these coordinates:

80-column mode: 1, 1, 80, 25
40-column mode: 1, 1, 40, 25

Return value None.

Portability **window** works with IBM PCs and compatibles only. A corresponding function exists in Turbo Pascal.

See also **clreol, clrscr, delline, gettextinfo, gotoxy, insline, puttext, textmode**

_write

Function Writes to a file.

Syntax int _write(int *handle*, void **buf*, unsigned *len*);

Prototype in io.h

Remarks This function attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle.*

The maximum number of bytes that **_write** can write is 65534, since 65535 (0xFFFF) is the same as –1, which is the error return indicator for **_write**.

_write does not translate a linefeed character (LF) to a CR/LF pair, since all its files are binary files.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk.

For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

For files opened with the O_APPEND option, the file pointer is not positioned to EOF by **_write** before writing the data.

Return value **_write** returns the number of bytes written. In case of error, **_write** returns –1 and sets the global variable *errno* to one of the following:

EACCES	Permission denied
EBADF	Bad file number

Portability **_write** is unique to DOS.

See also **lseek, _read, write**

write

Function Writes to a file.

Syntax int write(int *handle*, void **buf*, unsigned *len*);

Prototype in io.h

Remarks **write** writes a buffer of data to the file or device named by the given *handle*. *handle* is a file handle obtained from a **creat, open, dup,** or **dup2** call.

This function attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle*. Except when **write** is used to write to a text file, the number of bytes written to the file will be no more than the number requested.

The maximum number of bytes that **write** can write is 65534, since 65535 (0xFFFF) is the same as –1, which is the error return indicator for **write**.

On text files, when **write** sees a linefeed (LF) character, it outputs a CR/LF pair.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk.

For disks or disk files, writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device.

For files opened with the O_APPEND option, the file pointer is positioned to EOF by **write** before writing the data.

Return value **write** returns the number of bytes written. A **write** to a text file does not count generated carriage returns. In case of error, **write** returns –1 and sets the global variable *errno* to one of the following:

EACCES	Permission denied
EBADF	Bad file number

Portability **write** is available on UNIX systems.

See also **creat, lseek, open, read, _write**

A P P E N D I X

A

The Turbo C Interactive Editor

Introduction

Turbo C's built-in editor is specifically designed for creating program source text. If you are familiar with the Turbo Pascal or SideKick editor or MicroPro's WordStar program, you already know how to use the Turbo C editor, since its commands are almost identical to those of these editors. A section at the end of this appendix summarizes the few differences between Turbo C's editor commands and WordStar's commands.

The TC Editor has expanded memory support on systems running EMM (Extended Memory Manager) drivers conforming to the 3.2 (and above) Lotus/Intel/Microsoft Expanded Memory Specification. At startup, Turbo C determines whether it can use EMS memory; if it can, it automatically places the Editor's buffer in expanded memory. This frees about 64K of RAM memory for compiling and running your programs. If you happen not to want Turbo C to use available EMS memory that it finds, you can disable this feature at the TCINST Options/Environment/Options for Editor menu.

Turbo In, Turbo Out

To invoke the editor, choose Edit from Turbo C's main menu. The Edit window becomes the active window; the Edit window's title is highlighted and the cursor, a flashing underbar that marks the point at which text will be entered, is positioned in the Edit window.

To enter text, type as though you were using a typewriter. To end a line, press the *Enter* key.

To invoke the main menu from within the editor, press *F10* (the data in the Edit window remains onscreen).

The Edit Window Status Line

The status line in the top bar of the Edit window gives you information about the file you are editing, where in the file the cursor is located, and which editing modes are activated:

```
Line  Col  Insert  Indent  Tab  Fill  Unindent  *  X:FILENAME.EXT
```

`Line`	Shows which file line number contains the cursor.
`Col`	Shows which file column number contains the cursor.
`Insert`	Tells you that the editor is in Insert mode; characters entered on the keyboard are inserted at the cursor position, and text in front of the cursor moves to the right. Use the *Ins* key or *Ctrl-V* to toggle the editor between Insert mode and Overwrite mode. In Overwrite mode, text entered at the keyboard overwrites characters under the cursor, instead of inserting them before existing text.
`Indent`	Indicates the autoindent feature is On. You toggle it Off and On with the command *Ctrl-O I*.
`Tab`	Indicates whether or not you can insert tabs. Use *Ctrl-O T* to toggle this On or Off.
`Fill`	When tab mode is On, Optimal fill mode will cause the editor to fill the beginning of each line optimally with tabs and spaces. This option is toggled with *Ctrl-O F*.
`Unindent`	When Unindent mode is On, the backspace key will outdent one level whenever the cursor is on the first nonblank character of a line or on a blank line. This option is toggled with *Ctrl-O U*.
`*`	The asterisk appears before the file name whenever the file has been modified and has not yet been saved.
`X:FILENAME.EXT`	Indicates the drive (X:), name (FILENAME), and extension (.EXT) of the file you are editing. If you have not specified a file name yet, the file name and extension displayed is NONAME.C, Turbo C's default file name.

Editor Commands

The editor provides approximately 50 commands to move the cursor around, page through the text, find and replace strings, and so on. These commands can be grouped into five main categories:

- basic cursor movement commands
- quick cursor movement commands[quick cursor movement commands (TC editor)]
- insert and delete commands
- block commands
- miscellaneous commands

Table A.1 summarizes the commands. Each entry in the table consists of a command definition, followed by the default keystrokes used to activate the command. In the pages after Table A.1, we further explain the actions of each editor command.

Table A.1: Summary of Editor Commands

Command	Keystrokes
Basic Cursor Movement Commands	
Character left	*Ctrl-S* or *Left*
Character right	*Ctrl-D* or *Right*
Word left	*Ctrl-A*
Word right	*Ctrl-F*
Line up	*Ctrl-E* or *Up*
Line down	*Ctrl-X* or *Down*
Scroll up	*Ctrl-W*
Scroll down	*Ctrl-Z*
Page up	*Ctrl-R* or *PgUp*
Page down	*Ctrl-C* or *PgDn*
Quick Cursor Movement Commands	
Beginning of line	*Ctrl-Q S* or *Home*
End of line	*Ctrl-Q D* or *End*
Top of window	*Ctrl-Q E*
Bottom of window	*Ctrl-Q X*
Beginning of file	*Ctrl-Q R* or *Ctrl-PgUp*
End of file	*Ctrl-Q C* or *Ctrl-PgDn*
Beginning of block	*Ctrl-Q B*
End of block	*Ctrl-Q K*
Last cursor position	*Ctrl-Q P*
Insert and Delete Commands	
Insert mode On/Off	*Ctrl-V* or *Ins*
Insert line	*Ctrl-N*
Delete line	*Ctrl-Y*
Delete to end of line	*Ctrl-Q Y*
Delete character left of cursor	*Ctrl-H* or *Backspace*
Delete character under cursor	*Ctrl-G* or *Del*
Delete word right of cursor	*Ctrl-T*
Block Commands	
Mark block-begin	*Ctrl-K B*
Mark block-end	*Ctrl-K K*
Mark single word	*Ctrl-K T*

Table A.1: Summary of Editor Commands (continued)

Copy block	*Ctrl-K C*
Delete block	*Ctrl-K Y*
Hide/display block	*Ctrl-K H*
Move block	*Ctrl-K V*
Read block from disk	*Ctrl-K R*
Write block to disk	*Ctrl-K W*
Indent block	*Ctrl-K I*
Outdent block	*Ctrl-K U*
Miscellaneous Commands	
Abort operation	*Ctrl-U*
Autoindent On/Off	*Ctrl-O I*
Control character prefix	*Ctrl-P*
Fill mode	*Ctrl-O F*
Find	*Ctrl-Q F*
Find and replace	*Ctrl-Q A*
Find place marker	*Ctrl-Q 0, Ctrl-Q 1, Ctrl-Q 2,* etc.
Toggle menus/active window	*F10*
Load file	*F3*
Optimal fill mode	*Ctrl-O F*
Pair matching	*Ctrl-Q [, Ctrl-Q]*
Print file	*Ctrl-K P*
Quit edit, no save	*Ctrl-K D* or *Ctrl-K Q*
Repeat last find	*Ctrl-L*
Restore line	*Ctrl-Q L*
Save and edit	*Ctrl-K S* or *F2*
Set place marker	*Ctrl-K 0, Ctrl-K 1, Ctrl-K 2,* etc.
Tab	*Ctrl-I* or *Tab*
Tab mode	*Ctrl-O T*
Unindent mode	*Ctrl-O U*

Basic Cursor Movement Commands

The editor uses control-key commands to move the cursor up, down, back, and forth on the screen. To control cursor movement in the part of your file currently onscreen, use the following sequences:

Keystroke	Action
Ctrl-A	Moves to first letter in word to left of cursor
Ctrl-S	Moves to first position to left of cursor
Ctrl-D	Moves to first position to right of cursor
Ctrl-F	Moves to first letter in word to right of cursor
Ctrl-E	Moves up one line
Ctrl-R or *PgUp*	Scrolls screen and cursor up one full screen
Ctrl-X	Moves down one line
Ctrl-C or *PgDn*	Scrolls screen and cursor down one full screen
Ctrl-W	Scrolls screen down one line; cursor stays in line
Ctrl-Z	Scrolls screen up one line; cursor stays in line

Quick Cursor Movement Commands

The editor also provides eight commands to move the cursor quickly to the extreme ends of lines, to the beginning and end of the file, and to the last cursor position.

Keystroke	Action
Ctrl-Q S or *Home*	Moves to first column of the current line
Ctrl-Q D or *End*	Moves to the end of the current line
Ctrl-Q E	Moves to the top of the screen
Ctrl-Q X	Moves to the bottom of the screen
Ctrl-Q R or *Ctrl-PgUp*	Moves to the first character in the file
Ctrl-Q C or *Ctrl-PgDn*	Moves to the last character in the file

The *Ctrl-Q* prefix with a *B*, *K*, or *P* character allows you to jump to certain special points in a document.

Ctrl-Q B Moves the cursor to the block-begin marker set with *Ctrl-K B*. The command works even if the block is not displayed (see "Hide/display block" under "Block Commands") or if the block-end marker is not set.

Ctrl-Q K Moves the cursor to the block-end marker set with *Ctrl-K K*. The command works even if the block is not displayed (see "Hide/display block") or the block-begin marker is not set.

Ctrl-Q P Moves to the last position of the cursor before the last command. This command is particularly useful after a Find or Find/Replace operation has been executed, and you'd like to return to the last position before its execution.

Insert and Delete Commands

To write a program, you need to know more than just how to move the cursor around. You also need to be able to insert and delete text. The following commands insert and delete characters, words, and lines.

Insert mode On/ff *Ctrl-V* or *Ins*
When entering text, you can choose between two basic entry modes: *Insert mode* and *Overwrite mode*. You can switch between these modes with the Insert mode toggle, *Ctrl-V* or *Ins*. The current mode is displayed in the status line at the top of the screen.

Insert mode is the Turbo C editor's default; this lets you insert new characters into old text. Text to the right of the cursor simply moves to the right as you enter new text.

Use Overwrite mode to replace old text with new; any characters entered replace existing characters under the cursor.

Delete character left of cursor/Unindent *Ctrl-H* or *Backspace*
Moves one character to the left and deletes the character positioned there. Any characters to the right of the cursor move one position to the left. You can use this command to remove line breaks.

If the cursor is on the first nonblank character of a line or on a blank line, and if Unindent mode is toggled on (*Ctrl-O U*), this command will move the cursor and any characters to the right of it out one level of indentation.

Delete character under cursor *Ctrl-G* or *Del*
Deletes the character under the cursor and moves any characters to the right of the cursor one position to the left.

Delete word right of cursor *Ctrl-T*
Deletes the word to the right of the cursor. A word is defined as a sequence of characters delimited by one of the following characters:

space < > , ; . ( ) [ ] ^ ' * + – / $

This command works across line breaks, and may be used to remove them.

Insert line *Ctrl-N*
Inserts a line break at the cursor position.

Delete line *Ctrl-Y*
Deletes the line containing the cursor and moves any lines below one line up. There's no way to restore a deleted line, so use this command with care.

Delete to end of line *Ctrl-Q Y*

Deletes all text from the cursor position to the end of the line.

Block Commands

The block commands also require a control-character command sequence. A block of text is any amount of text, from a single character to hundreds of lines, that has been surrounded with special block-marker characters. There can be only one block in a document at a time.

You mark a block by placing a block-begin marker on the first character and a block-end marker on the last character of the desired portion of text. Once marked, you can copy, move, or delete the block, or write it to a file.

Mark block-begin *Ctrl-K B*
Marks the beginning of a block. The marker itself is not visible, and the block itself only becomes visible if a block-end marker is set. Marked text (a block) is displayed in a different intensity, background, or color, depending on the kind of adapter you have.

Mark block-end *Ctrl-K K*
Marks the end of a block. The marker itself is invisible, and the block itself becomes visible only if a block-begin marker is also set.

Mark single word *Ctrl-K T*
Marks a single word as a block, replacing the block-begin/block-end sequence. If the cursor is placed within a word, then the word will be marked. If it is not within a word, then the word to the left of the cursor will be marked.

Copy block *Ctrl-K C*
Copies a previously marked block to the current cursor position. The original block is unchanged, and the markers are placed around the new copy of the block. If no block is marked or the cursor is within the marked block, nothing happens.

Delete block *Ctrl-K Y*
Deletes a previously marked block. There is no provision to restore a deleted block, so be careful with this command.

Hide/display block *Ctrl-K H*
Causes the visual marking of a block to be alternately switched off and on. The block manipulation commands (copy, move, delete, and write to a file) work only when the block is displayed. Block-related cursor movements (jump to beginning/end of block) work whether the block is hidden or displayed.

Move block *Ctrl-K V*
Moves a previously marked block from its original position to the cursor position. The block disappears from its original position, and the markers remain around the block at its new position. If no block is marked, nothing happens.

Read block from disk *Ctrl-K R*
Reads a disk file into the current text at the cursor position, exactly as if it were a block. The text read is then marked as a block.

When you issue this command, Turbo C's editor prompts you for the name of the file to read. You can use DOS wildcards to select a file to read; a directory appears in a small window onscreen. The file specified may be any legal file name. If you specify no file type (.C, .TXT, .BAK, etc.) the editor assumes you meant .C. To read a file that lacks an extension, append a period to the file name.

Write block to disk *Ctrl-K W*
Writes a previously marked block to a file. The block is left unchanged in the current file, and the markers remain in place. If no block is marked, nothing happens.

When you issue this command, Turbo C's editor prompts you for the name of the file to write to. To select a file to overwrite, use DOS wildcards; a directory appears in a small window onscreen. If the file specified already exists, the editor issues a warning and prompts for verification before overwriting the existing file. You can give the file any legal name (the default extension is .C). To write a file that lacks an extension, append a period to the file name.

Miscellaneous Editing Commands

This section describes commands that do not fall into any of the categories already covered. These commands are listed in alphabetical order.

Abort operation *Ctrl-U*
Lets you abort any command in process whenever it pauses for input, such as when Find/Replace asks `Replace Y/N?`, or when you are entering a search string.

Autoindent On/Off *Ctrl-O I*
Provides automatic indenting of successive lines. When Autoindent is active, the cursor does not return to column one when you press *Enter*; instead, it returns to the starting column of the line you just terminated.

When you want to change the indentation, use the space bar or *Tab*, and *Left arrow* or *Backspace* to select the new column. When Autoindent is On, the message `Indent` shows up in the status line; when Off, the message disappears. Autoindent is On by default.

Control character prefix *Ctrl-P*

Allows you to enter control characters into the file by prefixing the desired control character with a *Ctrl-P*; that is, first press *Ctrl-P*, then press the desired control character. Control characters will appear as low-intensity capital letters on the screen (or inverse, depending on your screen setup).

Find *Ctrl-Q F*

Lets you search for a string of up to 30 characters. When you enter this command, the status line is cleared, and the editor prompts you for a search string. Enter the string you are looking for and then press *Enter*.
The search string may contain any characters, including control characters. You enter control characters into the search string with the *Ctrl-P* prefix. For example, enter a *Ctrl-T* by holding down the *Ctrl* key as you press *P*, and then press *T*. You may include a line break in a search string by specifying *Ctrl-M J* (hard return/linefeed). Note that *Ctrl-A* has special meaning: It matches any character and may be used as a wildcard in search strings.

You may edit search strings with the character left, character right, word left, and word right commands. Word right recalls the previous search string, which you may then edit. To abort (quit) the search operation, use the abort command (*Ctrl-U*).

When you specify the search string, Turbo C's editor asks for search options. The following options are available:

B Searches backward from the current cursor position toward the beginning of the text.

L Performs a local search of the marked block.

n Where *n* equals a number, finds the *n*th occurrence of the search string, counted from the current cursor position.

U Ignores uppercase/lowercase distinctions.

W Searches for whole words only, skipping matching patterns embedded in other words.

Examples of Find Options:

W Searches for whole words only. The search string *term* will match *term*, for example, but not *terminal*.

BU Searches backward and ignores uppercase/lowercase differences. *Block* matches both *blockhead* and *BLOCKADE*, and so on.

125 Finds the 125th occurrence of the search string.

You can end the list of find options (if any) by pressing *Enter*; the search starts. If the text contains a target matching the search string, the editor positions the cursor on the target. The search operation may be repeated by the Repeat last find command (*Ctrl-L*).

Find and replace *Ctrl-Q A*

This operation works identically to the Find command, except that you can replace the "found" string with any other string of up to 30 characters. Note that *Ctrl-A* only functions as a wildcard in the Find string; it has no special meaning in the Replace string.

When you specify the search string, the editor asks you to enter the string that will replace the search string. Enter up to 30 characters; control character entry and editing is performed as with the Find command. If you just press *Enter*, the editor replaces the target with nothing, effectively deleting it.

Your choice of options are the same as those in the Find command with the addition of the following:

N Replaces without asking; does not ask for confirmation of each occurrence of the search string.

n Replaces the next *n* cases of the search string. If the *G* option is used, the search starts at the top of the file; otherwise it starts at the current cursor position.

Examples of Find and Replace Options:

N10 Finds the next ten occurrences of the search string and replaces each without asking.

GW Finds and replaces whole words in the entire text, ignoring uppercase/lowercase. It prompts for a replacement string.

GNU Finds (throughout the file) uppercase and lowercase small, antelope-like creatures and replaces them without asking.

Again, you can end the option list (if any) by pressing *Enter*; the Find/Replace operation starts. When the editor finds the item (and if the *N*

option is not specified), it then positions the cursor at one end of the item, and asks `Replace (Y/N)?` in the prompt line at the top of the screen. You may abort the Find/Replace operation at this point with the Abort command (*Ctrl-U*). You can repeat the Find/Replace operation with the Repeat last find command (*Ctrl-L*).

Find place marker — *Ctrl-Q 0, Ctrl-Q 1, Ctrl-Q 2, Ctrl-Q 3*
Finds up to four place markers (0-3), one at a time, in text. Move the cursor to any previously set marker by pressing *Ctrl-Q* and the marker number, *n*.

Load file — *F3*
Lets you edit an existing file or create a new file.

Optimal fill On/Off — *Ctrl-O F*
When Tab mode is On, Optimal fill mode will cause the editor to fill the beginning of each line optimally with tabs and spaces.

Pair matching — *Ctrl-Q [* or *Ctrl-Q]*
Locates the mate to a paired delimiter marked by the cursor. *Ctrl-Q [* searches forward from a left delimiter, *Ctrl-Q]* searches backward from a right delimiter. The delimiters recognized by these two commands are

```
{ }   < >   ( )   [ ]   /* */   " "   ' '
```

Print file — *Ctrl-K P*
Expands tabs (replaces tabs with the appropriate number of spaces), then prints the marked block. If no block is marked, prints the whole file.

Quit edit, no save — *Ctrl-K D* or *Ctrl-K Q*
Quits the editor and returns you to the main menu. You can save the edited file on disk either explicitly with the main menu's Save option under the Files command or manually while in the editor (*Ctrl-K S* or *F2*).

Repeat last find — *Ctrl-L*
Repeats the latest Find or Find/Replace operation as if all information had been reentered.

Restore line — *Ctrl-Q L*
Lets you undo changes made to a line, as long as you have not left the line. The line is restored to its original state regardless of any changes you have made.

Save file — *Ctrl-K S* or *F2*
Saves the file and remains in the editor.

Set place marker — *Ctrl-K 0, Ctrl-K 1, Ctrl-K 2, Ctrl-K 3*
You can mark up to four places in text; press *Ctrl-K*, followed by a single digit *n* (0-3). After marking your location, you can work elsewhere in the

file and then easily return to the marked location by using the *Ctrl-Q n* command.

Tab *Ctrl-I* or *Tab*

Inserts a tab or spaces, depending on the Tab mode setting. Default setting for tabs is eight columns apart in the Turbo C editor, but this can be changed using the Options/Environment/Tab size menu option or TCINST.

Tab On/Off *Ctrl-O T*

With Tab mode On, a tab is placed in the text using a fixed tab stop which defaults to 8. (To change tab size, use the Options/ Environment/Tab size menu setting.) If you toggle Tab mode Off, it spaces to the beginning of the first letter of each word in the previous line.

Unindent On/Off *Ctrl-O U*

When Unindent mode is toggled On, the backspace key will outdent one level whenever the cursor is on the first nonblank character of a line or on a blank line.

The Turbo C Editor Vs. WordStar

A few of the Turbo C editor's commands are slightly different from WordStar. Also, although the Turbo C editor contains only a subset of WordStar's commands, several features not found in WordStar have been added to enhance program source-code editing. These differences are discussed here, in alphabetical order.

Autoindent:

The Turbo C editor's *Ctrl-O I* command toggles Autoindent mode On and Off.

Cursor movement:

Turbo C's cursor movement controls—*Ctrl-S, Ctrl-D, Ctrl-E,* and *Ctrl-X*—move freely around on the screen without jumping to column one on empty lines. This does not mean that the screen is full of blanks; on the contrary, all trailing blanks are automatically removed. This way of moving the cursor is especially useful for program editing, for example, when matching indented statements.

Delete to left:

The WordStar sequence *Ctrl-Q Del,* delete from cursor position to beginning of line, is not supported.

Mark word as block:
Turbo C allows you to mark a single word as a block using *Ctrl-K T*. This is more convenient than WordStar's two-step process of separately marking the beginning and the end of the word.

Movement across line breaks:
Ctrl-S and *Ctrl-D* do not work across line breaks. To move from one line to another you must use *Ctrl-E, Ctrl-X, Ctrl-A,* or *Ctrl-F*.

Quit edit:
Turbo C's *Ctrl-K Q* does not resemble WordStar's *Ctrl-K Q* (quit edit) command. In Turbo C, the changed text is not abandoned—it is left in memory, ready to be compiled and saved.

Undo:
Turbo C's *Ctrl-Q L* command restores a line to its pre-edit contents as long as the cursor has not left the line.

Updating disk file:
Since editing in Turbo C is done entirely in memory, the *Ctrl-K D* command does not change the file on disk as it does in WordStar. You must explicitly update the disk file with the Save option within the File menu or by using *Ctrl-K S* or *F2* within the editor.

A P P E N D I X

B

Compiler Error Messages

The Turbo C compiler diagnostic messages fall into three classes: fatals, errors, and warnings.

Fatal errors are rare and probably indicate an internal compiler error. When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart compilation.

Errors indicate program syntax errors, disk or memory access errors, and command line errors. The compiler will complete the current phase of the compilation and then stop. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing and code-generating).

Warnings do not prevent the compilation from finishing. They indicate conditions which are suspicious, but which are legitimate as part of the language. The compiler will also produce warnings if you use machine-dependent constructs in your source files.

The compiler prints messages with the message class first, then the source file name and line number where the compiler detected the condition, and finally the text of the message itself.

In the following lists, messages are presented alphabetically within message class. With each message, a probable cause and remedy are provided.

You should be aware of one detail about line numbers in error messages: the compiler only generates messages as they are detected. Because C does not force any restrictions on placing statements on a line of text, the true

cause of the error may be one or more lines before the line number mentioned. In the following message list, we have indicated those messages which often appear (to the compiler) to be on lines after the real cause.

Fatal Errors

Bad call of inline function
You have used an inline function taken from a macro definition, but have called it incorrectly. An inline function is one that begins and ends with a double underscore (_ _).

Irreducible expression tree
This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. Whatever the offending expression is, it should be avoided. You should notify Borland International if the compiler ever encounters this error.

Register allocation failure
This is a sign of some form of compiler error. Some expression on the indicated line of the source file was so complicated that the code generator could not generate code for it. You should simplify the offending expression, and if this fails to solve the problem, the expression should be avoided. Notify Borland International if the compiler encounters this error.

Errors

#operator not followed by macro argument name
In a macro definition, the # may be used to indicate stringizing a macro argument. The # must be followed by a macro argument name.

'XXXXXXXX' not an argument
Your source file declared the named identifier as a function argument but the identifier was not in the function argument list.

Ambiguous symbol 'XXXXXXXX'
The named structure field occurs in more than one structure with different offsets, types, or both. The variable or expression used to refer to the field is not a structure containing the field. Cast the structure to the correct type, or correct the field name if it is wrong.

Argument # missing name

A parameter name has been left out in a function prototype used to define a function. If the function is defined with a prototype, the prototype must include the parameter names.

Argument list syntax error

Arguments to a function call must be separated by spaces and closed with a right parenthesis. Your source file contained an argument followed by a character other than comma or right parenthesis.

Array bounds missing]

Your source file declared an array in which the array bounds were not terminated by a right bracket.

Array size too large

The declared array would be too large to fit in the available memory of the processor.

Assembler statement too long

Inline assembly statements may not be longer than 480 bytes.

Bad configuration file

The TURBOC.CFG file contains uncommented text that is not a proper command option. Configuration file command options must begin with a dash (-).

Bad file name format in include directive

Include file names must be surrounded by quotes ("*filename.h*") or angle brackets (<*filename.h*>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.

Bad ifdef directive syntax

An `#ifdef` directive must contain a single identifier (and nothing else) as the body of the directive.

Bad ifndef directive syntax

An `#ifndef` directive must contain a single identifier (and nothing else) as the body of the directive.

Bad undef directive syntax

An `#undef` directive must contain a single identifier (and nothing else) as the body of the directive.

Bit field size syntax

A bitfield must be defined by a constant expression between 1 and 16 bits in width.

Call of non-function

The function being called is declared as a non-function. This is commonly caused by incorrectly declaring the function or misspelling the function name.

Cannot modify a const object

This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.

Case outside of switch

The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched curly braces.

Case statement missing :

A **case** statement must have a constant expression followed by a colon. The expression in the **case** statement either was missing a colon or had some extra symbol before the colon.

Cast syntax error

A cast contains some incorrect symbol.

Character constant too long

Character constants may only be one or two characters long.

Compound statement missing }

The compiler reached the end of the source file and found no closing brace. This is most commonly caused by mismatched braces.

Conflicting type modifiers

This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) may be given on a function.

Constant expression required

Arrays must be declared with constant size. This error is commonly caused by misspelling a `#define` constant.

Could not find file 'XXXXXXXX.XXX'

The compiler is unable to find the file supplied on the command line.

Declaration missing ;

Your source file contained a **struct** or **union** field declaration that was not followed by a semicolon.

Declaration needs type or storage class

A declaration must include at least a type or a storage class. This means a statement like the following is not legal:

```
i,j;
```

Declaration syntax error

Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.

Default outside of switch

The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched curly braces.

Define directive needs an identifier

The first non-whitespace character after a `#define` must be an identifier. The compiler found some other character.

Division by zero

Your source file contained a divide or remainder in a constant expression with a zero divisor.

Do statement must have while

Your source file contained a **do** statement that was missing the closing **while** keyword.

Do-while statement missing (

In a **do** statement, the compiler found no left parenthesis after the **while** keyword.

Do-while statement missing)

In a **do** statement, the compiler found no right parenthesis after the test expression.

Do-while statement missing ;

In a **do** statement test expression, the compiler found no semicolon after the right parenthesis.

Duplicate case

Each **case** of a **switch** statement must have a unique constant expression value.

Enum syntax error

An **enum** declaration did not contain a properly formed list of identifiers.

Enumeration constant syntax error

The expression given for an **enum** value was not a constant.

Error Directive: XXXX

This message is issued when an `#error` directive is processed in the source file. The text of the directive is displayed in the message.

Error writing output file
This error most often occurs when the work disk is full. It could also indicate a faulty disk. If the disk is full, try deleting unneeded files and restarting the compilation.

Expression syntax
This is a catch-all error message when the compiler parses an expression and encounters some serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

Extra parameter in call
A call to a function, via a pointer defined with a prototype, had too many arguments given.

Extra parameter in call to XXXXXXXX
A call to the named function (which was defined with a prototype) had too many arguments given in the call.

File name too long
The file name given in an `#include` directive was too long for the compiler to process. File names in DOS must be no more than 64 characters long.

For statement missing (
In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

For statement missing)
In a **for** statement, the compiler found no right parenthesis after the control expressions.

For statement missing ;
In a **for** statement, the compiler found no semicolon after one of the expressions.

Function call missing)
The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

Function definition out of place
A function definition may not be placed inside another function. Any declaration inside a function that looks like the beginning of a function with an argument list is considered a function definition.

Function doesn't take a variable number of arguments
Your source file used the **va_start** macro inside a function that does not accept a variable number of arguments.

Goto statement missing label
The **goto** keyword must be followed by an identifier.

If statement missing (
In an **if** statement, the compiler found no left parenthesis after the **if** keyword.

If statement missing)
In an **if** statement, the compiler found no right parenthesis after the test expression.

Illegal character 'C' (0xXX)
The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed.

Illegal initialization
Initializations must be either constant expressions, or else the address of a global **extern** or **static** variable plus or minus a constant.

Illegal octal digit
The compiler found an octal constant containing a non-octal digit (8 or 9).

Illegal pointer subtraction
This is caused by attempting to subtract a pointer from a non-pointer.

Illegal structure operation
Structures may only be used with dot (.), address-of (&) or assignment (=) operators, or be passed to or from a function as parameters. The compiler encountered a structure being used with some other operator.

Illegal use of floating point
Floating point operands are not allowed in shift, bitwise boolean, conditional (? :), indirection (*), or certain other operators. The compiler found a floating-point operand with one of these prohibited operators.

Illegal use of pointer
Pointers can only be used with addition, subtraction, assignment, comparison, indirection (*) or arrow (->). Your source file used a pointer with some other operator.

Improper use of a typedef symbol
Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.

Inline assembly not allowed
Your source file contains inline assembly language statements and you are compiling it from within the Integrated Environment. You must use the TCC command to compile this source file.

Incompatible storage class
Your source file used the **extern** keyword on a function definition. Only **static** (or no storage class at all) is allowed.

Incompatible type conversion
Your source file attempted to convert one type to another, but the two types were not convertible. This includes converting a function to or from a non-function, converting a structure or array to or from a scalar type, or converting a floating-point value to or from pointer type.

Incorrect command-line argument: XXXXXXXX
The compiler did not recognize the command-line parameter as legal.

Incorrect configuration file argument: XXXXXXXX
The compiler did not recognize the configuration file parameter as legal; check for a preceding dash ("-").

Incorrect number format
The compiler encountered a decimal point in a hexadecimal number.

Incorrect use of default
The compiler found no colon after the **default** keyword.

Initializer syntax error
An initializer has a missing or extra operator, mismatched parentheses, or is otherwise malformed.

Invalid indirection
The indirection operator (*) requires a non-**void** pointer as the operand.

Invalid macro argument separator
In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.

Invalid pointer addition
Your source file attempted to add two pointers together.

Invalid use of arrow
An identifier must immediately follow an arrow operator (->).

Invalid use of dot
An identifier must immediately follow a period operator (.).

Lvalue required

The left hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.

Macro argument syntax error

An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.

Macro expansion too long

A macro may not expand to more than 4096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

May compile only one file when an output file name is given

You have supplied an `-o` command-line option, which allows only one output file name. The first file is compiled but the other files are ignored.

Mismatched number of parameters in definition

The parameters in a definition do not match the information supplied in the function prototype.

Misplaced break

The compiler encountered a **break** statement outside a **switch** or looping construct.

Misplaced continue

The compiler encountered a **continue** statement outside a looping construct.

Misplaced decimal point

The compiler encountered a decimal point in a floating point constant as part of the exponent.

Misplaced else

The compiler encountered an **else** statement without a matching **if** statement. An extra **else** statement could cause this message, but it could also be caused by an extra semicolon, missing curly braces, or some syntax error in a previous **if** statement.

Misplaced elif directive

The compiler encountered an `#elif` directive without any matching `#if`, `#ifdef` or `#ifndef` directive.

Misplaced else directive

The compiler encountered an `#else` directive without any matching `#if`, `#ifdef` or `#ifndef` directive.

Misplaced endif directive

The compiler encountered an `#endif` directive without any matching `#if`, `#ifdef` or `#ifndef` directive.

Must be addressable

An ampersand (&) has been applied to an object that is not addressable, such as a register variable.

Must take address of memory location

Your source file used the address-of operator (&) with an expression which cannot be used that way; for example, a register variable.

No file name ending

The file name in an `#include` statement was missing the correct closing quote or angle bracket.

No file names given

The Turbo C compile command (TCC) contained no file names. You have to specify a source file name.

Non-portable pointer assignment

Your source file assigned a pointer to a non-pointer, or vice versa. Assigning a constant zero to a pointer is allowed as a special case. You should use a cast to suppress this error message if the comparison is proper.

Non-portable pointer comparison

Your source file made a comparison between a pointer and a non-pointer other than the constant zero. You should use a cast to suppress this error message if the comparison is proper.

Non-portable return type conversion

The expression in a return statement was not the same type as the function declaration. With one exception, this is only triggered if the function or the return expression is a pointer. The exception to this is that a function returning a pointer may return a constant zero. The zero will be converted to an appropriate pointer value.

Not an allowed type

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

Out of memory

The total working storage is exhausted. Compile the file on a machine with more memory. If you already have 640K, you may have to simplify the source file.

Pointer required on left side of →
Nothing but a pointer is allowed on the left side of the arrow (→).

Redeclaration of 'XXXXXXXX'
The named identifier was previously declared.

Size of structure or array not known
Some expression (such as a **sizeof** or storage declaration) occurred with an undefined structure or an array of empty length. Structures may be referenced before they are defined as long as their size is not needed. Arrays may be declared with empty length if the declaration does not reserve storage or if the declaration is followed by an initializer giving the length.

Statement missing ;
The compiler encountered an expression statement without a semicolon following it.

Structure or union syntax error
The compiler encountered the **struct** or **union** keyword without an identifier or opening curly brace following it.

Structure size too large
Your source file declared a structure which reserved too much storage to fit in the memory available.

Subscripting missing]
The compiler encountered a subscripting expression which was missing its closing bracket. This could be caused by a missing or extra operator, or mismatched parentheses.

Switch statement missing (
In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.

Switch statement missing)
In a **switch** statement, the compiler found no right parenthesis after the test expression.

Too few parameters in call
A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.

Too few parameters in call to 'XXXXXXXX'

A call to the named function (declared using a prototype) had too few arguments.

Too many cases
A switch statement is limited to 257 cases.

Too many decimal points
The compiler encountered a floating point constant with more than one decimal point.

Too many default cases
The compiler encountered more than one **default** statement in a single **switch**.

Too many exponents
The compiler encountered more than one exponent in a floating point constant.

Too many initializers
The compiler encountered more initializers than were allowed by the declaration being initialized.

Too many storage classes in declaration
A declaration may never have more than one storage class.

Too many types in declaration
A declaration may never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum** or **typedef**-*name*.

Too much auto memory in function
The current function declared more automatic storage than there is room for in the available memory.

Too much code defined in file
The combined size of the functions in the current source file exceeds 64K bytes. You may have to remove unneeded code, or split up the source file.

Too much global data defined in file
The sum of the global data declarations exceeds 64K bytes. Check the declarations for any array that may be too large. Also consider reorganizing the program if all the declarations are needed.

Two consecutive dots
Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), there is no way two dots can legally occur in a C program.

Type mismatch in parameter #

The function called, via a function pointer, was declared with a prototype; the given parameter #N (counting left-to-right from 1) could not be converted to the declared parameter type.

Type mismatch in parameter # in call to 'XXXXXXXX'

Your source file declared the named function with a prototype, and the given parameter #N (counting left-to-right from 1) could not be converted to the declared parameter type.

Type mismatch in parameter 'XXXXXXXX'

Your source file declared the function called via a function pointer with a prototype, and the named parameter could not be converted to the declared parameter type.

Type mismatch in parameter 'XXXXXXXX' in call to 'YYYYYYYY'

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type.

Type mismatch in redeclaration of 'XXX'

Your source file redeclared a variable with a different type than was originally declared for the variable. This can occur if a function is called and subsequently declared to return something other than an integer. If this has happened, you must declare the function before the first call to it.

Unable to create output file 'XXXXXXXX.XXX'

This error occurs if the work disk is full or write protected. If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write protected, move the source files to a writable disk and restart the compilation.

Unable to create turboc.lnk

The compiler cannot create the temporary file TURBOC.$LN because it cannot access the disk or the disk is full.

Unable to execute command 'XXXXXXXX'

TLINK or TASM cannot be found, or possibly the disk is bad.

Unable to open include file 'XXXXXXXX.XXX'

The compiler could not find the named file. This could also be caused if an `#include` file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try `FILES=20`). Check whether the named file exists.

Unable to open input file 'XXXXXXXX.XXX'

This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is on the proper disk or directory.

Undefined label 'XXXXXXXX'

The named label has a **goto** in the function, but no label definition.

Undefined structure 'XXXXXXXX'

Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.

Undefined symbol 'XXXXXXXX'

The named identifier has no declaration. This could be caused by a misspelling either at this point or at the declaration. This could also be caused if there was an error in the declaration of the identifier.

Unexpected end of file in comment started on line #

The source file ended in the middle of a comment. This is normally caused by a missing close of comment (*/).

Unexpected end of file in conditional started on line #

The source file ended before the compiler encountered `#endif`. The `#endif` either was missing or misspelled.

Unknown preprocessor directive: 'XXX'

The compiler encountered a # character at the beginning of a line, and the directive name following was not one of these: `define`, `undef`, `line`, `if`, `ifdef`, `ifndef`, `include`, `else` or `endif`.

Unterminated character constant

The compiler encountered an unmatched apostrophe.

Unterminated string

The compiler encountered an unmatched quote character.

Unterminated string or character constant

The compiler found no terminating quote after the beginning of a string or character constant.

User break

You typed a *Ctrl-Break* while compiling or linking in the Integrated Environment. (This is not an error, just a confirmation.)

While statement missing (

In a **while** statement, the compiler found no left parenthesis after the **while** keyword.

While statement missing)

In a **while** statement, the compiler found no right parenthesis after the test expression.

Wrong number of arguments in call of 'XXXXXXXX'

Your source file called the named macro with an incorrect number of arguments.

Warnings

'XXXXXXXX' declared but never used

Your source file declared the named variable as part of the block just ending, but the variable was never used. The warning is indicated when the compiler encounters the closing curly brace of the compound statement or function. The declaration of the variable occurs at the beginning of the compound statement or function.

'XXXXXXXX' is assigned a value which is never used

The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing curly brace.

'XXXXXXXX' not part of structure

The named field was not part of the structure on the left hand side of the dot (.) or arrow (→), or else the left hand side was not a structure (for a dot) or pointer to structure (for an arrow).

Ambiguous operators need parentheses

This warning is displayed whenever two shift, relational or bitwise-boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears unparenthesized with a shift operator will produce this warning. Programmers frequently confuse the precedence of these operators, since the precedence assigned to them is somewhat counter-intuitive.

Both return and return of a value used

This warning is issued when the compiler encounters a **return** statement that disagrees with some previous **return** statement in the function. It is almost certainly an error for a function to return a value in only some of the **return** statements.

Call to function with no prototype

This message is given if the "Prototypes required" warning is enabled and you call a function without first giving a prototype for that function.

Call to function '*XXXX*' with no prototype

This message is given if the "Prototypes required" warning is enabled and you call function *XXXX* without first giving a prototype for that function.

Code has no effect

This warning is issued when the compiler encounters a statement with some operators which have no effect. For example the statement

```
a + b;
```

has no effect on either variable. The operation is unnecessary and probably indicates a bug.

Constant is long

The compiler encountered either a decimal constant greater than 32767 or an octal (or hexadecimal) constant greater than 65535 without a letter *l* or *L* following it. The constant is treated as a **long**.

Constant out of range in comparison

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an **unsigned** quantity to –1 makes no sense. To get an **unsigned** constant greater than 32767 (in decimal), you should either cast the constant to **unsigned** (for example, (**unsigned**)65535) or append a letter *u* or *U* to the constant (for example, 65535u).

Conversion may lose significant digits

For an assignment operator or some other circumstance, your source file requires a conversion from **long** or **unsigned long** to **int** or **unsigned int** type. On some machines, since **int** type and **long** type variables have the same size, this kind of conversion may alter the behavior of a program being ported.

Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a **char** expression to 4000, the code will still perform the test. This also means that comparing an **unsigned** expression to –1 will do something useful, since an **unsigned** can have the same bit pattern as a –1 on the 8086.

Function should return a value

Your source file declared the current function to return some type other than **int** or **void**, but the compiler encountered a return with no value. This is usually some sort of error. **int** functions are exempt, since in old

versions of C there was no **void** type to indicate functions which return nothing.

Hexadecimal or octal constant too large
In a string literal or character constant, you used a hexadecimal or octal escape sequence with a value exceeding 255, for example, `\777` or `\x1234`.

Mixing pointers to signed and unsigned char
You converted a **char** pointer to an **unsigned char** pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but on the 8086, it is often harmless.)

No declaration for function 'XXXXXXXX'
This message is given if the "Declaration required" warning is enabled and you call a function without first declaring that function. The declaration can be either classic or modern (prototype) style.

Non-portable pointer assignment
Your source file assigned a pointer to a non-pointer, or vice versa. Assigning a constant zero to a pointer is allowed as a special case. You should use a cast to suppress this warning if the comparison is proper.

Non-portable pointer comparison
Your source file compared a pointer to a non-pointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.

Non-portable return type conversion
The expression in a **return** statement was not the same type as the function declaration. With one exception, this is only triggered if the function or the return expression is a pointer. The exception to this is that a function returning a pointer may return a constant zero. The zero will be converted to an appropriate pointer value.

Parameter 'XXXXXXXX' is never used
The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

Possible use of 'XXXXXXXX' before definition

Your source file used the named variable in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the program uses it.

Possibly incorrect assignment

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (i.e. part of an **if, while** or **do-while** statement). More often than not, this is a typographical error for the equality operator. If you wish to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. Thus,

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

should be rewritten as

```
if ((a = b) != 0) ...
```

Redefinition of 'XXXXXXXX' is not identical

Your source file redefined the named macro using text that was not exactly the same as the first definition of the macro. The new text replaces the old.

Restarting compile using assembly

The compiler encountered an **asm** with no accompanying `-B` command line option or `#pragma inline` statement. The compile restarts using assembly language capabilities.

Structure passed by value

If "Structure passed by value" warning is enabled, this warning is generated anytime a structure is passed by value as an argument. It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.

Superfluous & with function or array

An address-of operator (&) is not needed with an array name or function name; any such operators are discarded.

Suspicious pointer conversion

The compiler encountered some conversion of a pointer which caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.

Undefined structure 'XXXXXXXX'

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

Unknown assembler instruction

The compiler encountered an inline assembly statement with a disallowed opcode. Check the spelling of the opcode. Also check the list of allowed opcodes to see if the instruction is acceptable.

Unreachable Code

A **break, continue, goto** or **return** statement was not followed by a label or the end of a loop or function. The compiler checks **while, do** and **for** loops with a constant test condition, and attempts to recognize loops which cannot fall through.

Void functions may not return a value

Your source file declared the current function as returning **void**, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.

Zero length structure

Your source file declared a structure whose total size was zero. Any use of this structure would be an error.

A P P E N D I X

C

TCC Command-Line Options

This appendix lists each of the Turbo C compile-time, command-line options in alphabetical order under option type, and describes what each option does. The options are broken down into three general types:

- compiler options
- linker options
- environment options

Within the compiler options, there are several categories of options; these specify

- memory model
- #defines (macro definitions)
- code generation options
- optimization options
- source code options
- error-reporting options
- segment-naming control

To see an on-screen listing of all the TCC (command-line Turbo C) options, type `tcc` *Enter* at the DOS prompt (when you're in the TURBOC directory). Most of the command-line options have counterparts in the Turbo C Integrated Development Environment (TC) **O**ptions menus (and a few other menus). See Table C.1 for a correlation of the TC menu selections and the TCC command-line options.

Table C.1: Correlation of Command-Line Options and Menu Selections

Command-Line Switch		Menu Selection
–A		O/C/Source/ANSI keywords only...On
–a		O/C/Code generation/Alignment...Word
–a-	**	O/C/Code generation/Alignment...Byte
–B		(Not available)
–C		O/C/Source/Nested comments...On
–c		Compile/Compile to OBJ
–D*name*		O/C/Defines
–D*name=string*		O/C/Defines
–d		O/C/Code generation/Merge duplicate strings...On
–d-	**	O/C/Code generation/Merge duplicate strings...Off
–E*filename*		(Not available)
–e*filename*		(Not available)
–f	**	O/C/Code generation/Floating point...Emulation
–f-		O/C/Code generation/Floating point...None
–f87		O/C/Code generation/Floating point...8087
–G		O/C/Optimization/Optimize for...Speed
–g#		O/C/Errors/Warnings: stop after...#
–I*pathname*		O/D/Include directories
–i#		O/C/S/Identifier length...#
–j#		O/C/Errors/Errors: stop after...#
–K		O/C/Code generation/Default char type...Unsigned
–K-	**	O/C/Code generation/Default char type...Signed
–k	**	O/C/Code generation/Standard stack frame...On
–L*pathname*		O/D/Library directory
–l*x*		(Not available)
–M		O/L/Map file
–mc		O/C/Model...Compact
–mh		O/C/Model...Huge
–ml		O/C/Model...Large
–mm		O/C/Model...Medium
–ms	**	O/C/Model...Small
–mt		O/C/Model...Tiny
–N		O/C/Code generation/Test stack overflow...On
–n*pathname*		O/D/Output directory
–O		O/C/Optimization/Jump optimization...On
–o*filename*		(Not available)
–p		O/C/Code generation/Calling convention...Pascal
–p-	**	O/C/Code generation/Calling convention...C
–r	**	O/C/Optimization/Use register variables...On
–S		(Not available)
–U*name*		(Not available)
–u	**	O/C/Code generation/Generate underbars...On
–v		Debug/Source debugging...On
–w		O/C/Errors/Display warnings...On
–w-		O/C/Errors/Display warnings...Off
–w*xxx*		O/C/Errors/Portability warnings, ANSI violations, Common errors, or Less common errors...On
–w-*xxx*		O/C/Errors/Portability warnings, ANSI violations, Common errors, or Less common errors...Off
–y		O/C/Code generation/Line numbers...On
–Z		O/C/Optimization/Register optimization...On
–zA*name*		O/C/Names/Code/Class

-zB*name*		O/C/Names/BSS/Class
-zC*name*		O/C/Names/Code/Segment
-zD*name*		O/C/Names/BSS/Segment
-zG*name*		O/C/Names/BSS/Group
-zP*name*		O/C/Names/Code/Group
-zR*name*		O/C/Names/Data/Segment
-zS*name*		O/C/Names/Data/Group
-zT*name*		O/C/Names/Data/Class
-1		O/C/Code generation...80186/80286
-1-	**	O/C/Code generation...8088/8086

O/ = Options   C/ = Compiler   E/ = Environment   ** = Default

Turning Options On and Off

You select command-line options by entering a dash (-) immediately followed by the option letter (like this, `-I`). To turn an option Off, add a second dash after the option letter. For example, `-A` turns the ANSI keywords option On and `-A-` turns the option Off.

This feature is useful for disabling or enabling individual switches on the command line, thereby overriding the corresponding settings in the configuration file.

Syntax

You select Turbo C compiler options on the DOS command line, with the following syntax:

```
tcc [option option ...] filename filename ...
```

Turbo C compiles files according to the following set of rules:

`filename.asm`	Invoke TASM to assemble to .OBJ
`filename.obj`	Include as object at link time
`filename.lib`	Include as library at link time
`filename`	Compile filename.c
`filename.c`	Compile filename.c
`filename.xyz`	Compile filename.xyz

For example, given the following command line

```
tcc -a -f -C -O -Z -emyexe oldfile1.c oldfile2 nextfile.c
```

TCC will compile OLDFILE1.C, OLDFILE2.C, and NEXTFILE.C to .OBJ, producing an executable program file named MYEXE.EXE with the word alignment (-a), floating-point emulation (-f), nested comments (-C), jump optimization (-O), and register optimization (-Z) options selected.

TCC will invoke TASM if you give it an .ASM file on the command line or if a .C file contains inline assembly. The switches TCC gives to TASM are

```
/mx /D__mdl__
```

where *mdl* is one of: tiny, small, medium, compact, large, or huge. The /mx switch tells TASM to assemble with case-sensitivity on.

Compiler Options

Turbo C's command-line compiler options can be broken down into eight logical groups. These groups, and the ties that bind them, are as follows:

- ***Memory model options*** allow you to specify under which memory model Turbo C will compile your program. (The models are tiny, small, medium, compact, large, and huge.)
- ***#defines (macro definitions)*** allow you to define macros (also known as *manifest* or *symbolic* constants) on the command line. The default definition is the single space character. A numeric value, or a string may also be specified; these options also allow you to undefine previously defined macros.
- ***Code generation options*** govern characteristics of the generated code to be used at run time, such as the floating-point mode, calling convention, char type, or CPU instructions.
- ***Optimization options*** allow you to specify how the object code is to be optimized; for size or speed, with or without the use of register variables, and with or without redundant load operations.
- ***Source code options*** cause the compiler to recognize (or ignore) certain features of the source code; implementation-specific (non-ANSI) keywords, nested comments, and identifier lengths.
- ***Error-reporting options*** allow you to tailor which warning messages the compiler will report, and the maximum number of warnings (and errors) that can occur before the compilation stops.
- ***Segment-naming control options*** allows you to rename segments and to reassign their groups and classes.
- ***Compilation control options*** allow you to direct the compiler to

- compile to assembly code (rather than to an object module)
- compile a source file that contains inline assembly
- compile without linking

Memory Model

–mc Compile using compact memory model.

–mh Compile using huge memory model.

–ml Compile using large memory model.

–mm Compile using medium memory model.

–ms Compile using small memory model (the default).

–mt Compile using tiny memory model. Generates almost the same code as the small memory model, but uses C0T.OBJ in any link performed to produce a tiny model program.

For details about the Turbo C memory models, refer to Chapter 12 in the *Turbo C User's Guide.*

#defines

–D*xxx* Defines the named identifier *xxx* to the string consisting of the single space character.

–D*xxx*=*string* Defines the named identifier *xxx* to the string *string* after the equal sign. *string* cannot contain any spaces or tabs.

–U*xxx* Undefines any previous definitions of the named identifier *xxx*.

Turbo C allows you to make multiple `#define` entries on the command line in any of the following ways:

- You can include multiple entries after a single `-D` option, separating entries with a semicolon (this is known as "ganging" options):

```
tcc -Dxxx;yyy=1;zzz=NO myfile.c
```

- You can place more than one `-D` option on the command line:

```
tcc -Dxxx -Dyyy=1 -Dzzz=NO myfile.c
```

- You can mix ganged and multiple `-D` listings:

```
tcc -Dxxx -Dyyy=1;zzz=NO myfile.c
```

Code Generation Options

–1 Causes Turbo C to generate extended 80186 instructions. This option is also used to generate 80286 programs running in the real mode, such as with the IBM PC/AT under DOS.

–a Forces integer size items to be aligned on a machine-word boundary. Extra bytes will be inserted in a structure to ensure member alignment. Automatic and global variables will be aligned properly. **char** and **unsigned char** variables and fields may be placed at any address; all others must be placed at an even numbered address. (Off by default, allowing bytewise alignment.)

–d Merges literal strings when one string matches another; this produces smaller programs. (Off by default.)

–f87 Generates floating-point operations using inline 8087 instructions rather than using calls to 8087 emulation library routines. Specifies that a floating-point processor will be available at run time, so programs compiled with this option will not run on a machine that does not have a floating-point chip. (As currently implemented, this switch affects only which libraries are linked.)

–f Emulates 8087 calls at run time if the run-time system does not have an 8087; if it does have one, calls the 8087 for floating-point calculations (the default).

–f- Specifies that the program contains no floating-point calculations, so no floating-point libraries will be linked at the link step.

–K Causes the compiler to treat all **char** declarations as if they were **unsigned char** type. This allows for compatibility with other compilers that treat **char** declarations as **unsigned**. By default, **char** declarations are **signed**.

–k Generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines. The default is On.

–N Generates stack overflow logic at the entry of each function: This will cause a stack overflow message to appear when a stack overflow is detected. This is costly in both program size and speed but is provided as an option because stack overflows can be very difficult to detect. If an overflow is detected, the message "Stack overflow!" is printed and the program exits with an exit code of 1.

–p Forces the compiler to generate all subroutine calls and all functions using the Pascal parameter-passing sequence. The resulting function calls are smaller and faster. Functions must pass the correct number and type of arguments, unlike normal C usage which permits a variable number of function arguments. You can use the **cdecl** statement to override this option and specifically declare functions to be C-type.

–u With -u selected, when you declare an identifier, Turbo C automatically sticks an underscore (_) on the front before saving that identifier in the object module.

Turbo C treats Pascal-type identifiers (those modified by the **pascal** keyword) differently—they are uppercase and are *not* prefixed with an underscore.

Underscores for C identifiers are optional, but On by default. You can turn them Off with -u-. However, if you are using the standard Turbo C libraries, you will then encounter problems unless you rebuild the libraries. (To do this, you will need the Turbo C run-time library source code; refer to Chapter1 of this manual and contact Borland International for more information.)

See Chapter 12, "Advanced Programming in Turbo C" in the *Turbo C User's Guide* for details about underscores.

Note: Unless you are an expert, don't use -u-.

–y Includes line numbers in the object file for use by a symbolic debugger. This increases the size of the object file but will not affect size or speed of the executable program.

This option is only useful in concert with a symbolic debugger that can use the information.

–v Tells the compiler to include debug information in the .OBJ file so that the file(s) being compiled can be debugged with either the TC integrated debugger or the standalone debugger. The compiler also passes this switch on to the linker so it can include the debug information in the .EXE file.

Optimization Options

–G Causes the compiler to bias its optimization in favor of speed over size.

–O Optimizes by eliminating redundant jumps, and reorganizing loops and switch statements.

–r- Suppresses the use of register variables.

When you are using the `-r-` option or the **O/C/Optimization/Use** register variables Off, the compiler will not use register variables, and it also will not preserve and respect register variables (SI,DI) from any caller. For that reason, you should not have code that uses register variables call code which has been compiled with `-r-`.

On the other hand, if you are interfacing with existing assembly-language code that does not preserve SI,DI, the `-r-` option will allow you to call that code from Turbo C.

Note: Unless you are an expert, don't use `-r-`.

–r Enables the use of register variables (the default).

–Z Suppresses redundant load operations by remembering the contents of registers and reusing them as often as possible.

Note: You should exercise caution when using this option, because the compiler cannot detect if a register has been invalidated indirectly by a pointer.

For example, if a variable *A* is loaded into register DX, it is retained. If *A* is later assigned a value, the value of DX is reset to indicate that its contents are no longer current. Unfortunately, if the value of *A* is modified indirectly (by assigning through a pointer that points to *A*), Turbo C will not catch this and will continue to remember that DX contains the (now obsolete) value of *A*.

The `-Z` optimization is designed to suppress register loads when the value being loaded is already in a register. This can eliminate whole instructions and also convert instructions from referring to memory locations to using registers instead.

The following artificial sequence illustrates both the benefits and the drawbacks of this optimization, and demonstrates why you need to exercise caution when using `-Z`.

C Code	Optimized Assembler
`func()`	
`{`	
`  int   A, *P, B;`	
`  A = 4;`	`MOV   A,4`
`  ...`	
`  B = A;`	`MOV   AX,A`
	`MOV   B,AX`
`  P = &A;`	`LEA   BX,A`
	`MOV   P,BX`
`  *P = B + 5;`	`MOV   DX,AX`
	`ADD   DX,5`
	`MOV   [BX],DX`
`  printf("%d\n", A);`	`PUSH  AX`
`}`	

Note first that on the statement `*P = B + 5`, the code generated uses a move from AX to DX first. Without the `-Z` optimization, the move would be from *B*, generating a longer and slower instruction.

Second, the assignment into **P* recognizes that *P* is already in BX, so a move from *P* to BX after the `add` instruction has been eliminated. These improvements are harmless and generally useful.

The call to **printf**, however, is not correct. Turbo C sees that AX contains the value of *A*, and so pushes the contents of the register rather than the contents of the memory location. The **printf** will then display a value of 4 rather than the correct value of 9. The indirect assignment through *P* has hidden the change to *A*.

If the statement `*P = B + 5` had been written as `A = B + 5`, Turbo C would recognize a change in value.

The contents of registers are forgotten whenever a function call is made or when a point is reached where a jump could go (such as a label, a case statement, or the beginning or end of a loop). Because of this limit and the small number of registers in the 8086 family of processors, most programs using this optimization will never behave incorrectly.

Source Code Options

–A Compiles ANSI-compatible code: Any of the Turbo C extension keywords are ignored and may be used as normal identifiers. These keywords include:

near	**far**	**huge**	**cdecl**
asm	**pascal**	**interrupt**	
_es	**_ds**	**_cs**	**_ss**

and the register pseudo-variables, such as _AX, _BX, _SI, etc.

–C Allows nesting of comments. Comments may not normally be nested.

–i# Causes the compiler to recognize only the first # characters of identifiers. All identifiers, whether variables, preprocessor macro names, or structure member names, are treated as distinct only if their first # characters are distinct.

By default, Turbo C uses 32 characters per identifier. Other systems, including UNIX, ignore characters beyond the first 8. If you are porting to these other environments, you may wish to compile your code with a smaller number of significant characters. Compiling in this manner will help you see if there are any name conflicts in long identifiers when they are truncated to a shorter significant length.

Error-Reporting Options

–g# Stops compiling after # messages (warning and error messages combined).

–j# Stops compiling after # error messages.

–w*xxx* Enables the warning message indicated by *xxx*. The option –w-*xxx* suppresses the warning message indicated by *xxx*. See Appendix B of this manual for a detailed explanation of these warning messages. The possible values for –w*xxx* are as follows:

ANSI Violations

–wbig*	Hexadecimal or octal constant too large.
–wdup*	Redefinition of 'XXXXXXXX' is not identical.
–wret*	Both return and return of a value used.
–wstr*	'XXXXXXXX' not part of structure.
–wstu*	Undefined structure 'XXXXXXXX'.
–wsus*	Suspicious pointer conversion.
–wvoi*	Void functions may not return a value.
–wzst*	Zero length structure.

Common Errors

–waus*	'XXXXXXXX' is assigned a value that is never used.
–wdef*	Possible use of 'XXXXXXXX' before definition.
–weff*	Code has no effect.
–wpar*	Parameter 'XXXXXXXX' is never used.
–wpia*	Possibly incorrect assignment.
–wrch*	Unreachable code.
–wrvl	Function should return a value.

Less Common Errors

–wamb	Ambiguous operators need parentheses.
–wamp	Superfluous & with function or array.
–wnod	No declaration for function 'XXXXXXXX'.
–wpro	Call to function with no prototype.
–wstv	Structure passed by value.
–wuse	'XXXXXXXX' declared but never used.

Portability Warnings

–wapt*	Non-portable pointer assignment.
–wcln	Constant is long.
–wcpt*	Non-portable pointer comparison.
–wrng*	Constant out of range in comparison.
–wrpt*	Non-portable return type conversion.
–wsig	Conversion may lose significant digits.
–wucp	Mixing pointers to **signed** and **unsigned char**.

* On by default. All others are off by default.

Segment-Naming Control

–zA*name*	Changes the name of the code segment class to *name*. By default, the code segment is assigned to class CODE.
–zB*name*	Changes the name of the uninitialized data segment class to *name*. By default, the uninitialized data segments are assigned to class BSS.
–zC*name*	Changes the name of the code segment to *name*. By default, the code segment is named _TEXT, except for the medium, large and huge models, where the name is *filename_TEXT*. (*filename* here is the source file name.)
–zD*name*	Changes the name of the uninitialized data segment to *name*. By default, the uninitialized data segment is named _BSS, except in the huge model, where no uninitialized data segment is generated.
–zG*name*	Changes the name of the uninitialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
–zP*name*	Causes any output files to be generated with a code group for the code segment named *name*.
–zR*name*	Sets the name of the initialized data segment]] to *name*. By default, the initialized data segment is named _DATA except in the huge model, where the segment is named *filename*_DATA.
–zS*name*	Changes the name of the initialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
–zT*name*	Sets the name of the initialized data segment class to *name*. By default the initialized data segment class is named DATA.
–zX*	Uses the default name for X: for example, –zA* assigns the default class name CODE to the code segment.

Note: Do not use these switches unless you have a good understanding of segmentation on the 8086 processor. Under normal circumstances, you will not need to specify segment names.

Compilation Control Options

–B Compiles and calls the assembler to process inline assembly code.

Note that this option is not available in the Integrated Environment (TC.EXE).

–c Compiles and assembles the named .C and .ASM files, but does not execute a link command.

–o*filename* Compiles the named file to the specified *filename.OBJ*.

–S Compiles the named source files and produces assembly language output files (.ASM), but does not assemble.

Note that this option is not available in the Integrated Environment (TC.EXE).

–E*filename* Uses *filename* as the name of the assembler to use. By default, TASM is used.

Linker Options

–e*filename* Derives the executable program's name from *filename* by adding the file extension .EXE (the program name will then be FILENAME.EXE). *filename* must immediately follow the `-e`, with no intervening whitespace. Without this option, the linker derives the .EXE file's name from the name of the first source or object file in the file name list.

–M Forces the linker to produce a full link map. The default is to produce no link map.

–l*x* Passes option *x* to the linker. The switch `-l-`*x* suppresses option *x*. More than one option can appear after the `-l`. See the section on TLINK in Appendix D for a list of options.

Environment Options

–I*directory* Searches *directory*, the drive specifier or path name of a subdirectory, for include files (in addition to searching the standard places). A drive specifier is a single letter, either uppercase or lowercase, followed by a colon (:). A directory is

any valid path name of a directory file. Multiple -I directory options can be given.

–L*directory* Forces the linker to get the C0x.OBJ start-up object file and the Turbo C library files (Cx.LIB, MATHx.LIB, EMU.LIB, and FP87.LIB) from the named directory. By default, the linker looks for them in the current directory.

–n*xxx* Places any .OBJ or .ASM files created by the compiler in the directory or drive named by the path *xxx*.

Turbo C is able to search multiple directories for include and library files. This means that the syntax for the library directories (-L) and include directories (-I) command-line options, like that of the #define option (-D), allows multiple listings of a given option.

Here is the syntax for these options:

Library directories: `-Ldirname[;dirname;...]`

Include directories: `-Idirname[;dirname;...]`

The parameter *dirname* used with -L and -I can be any directory path name.

You can enter these multiple directories on the command line in the following ways:

- You can "gang" multiple entries with a single -L or -I option, separating ganged entries with a semicolon, like this:

```
tcc -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
tcc -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix ganged and multiple listings, like this:

```
tcc -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfile.c
```

If you list multiple -L or -I options on the command line, the result is cumulative: The compiler will search all the directories listed, or define the specified constants, in order from left to right.

Note: The integrated environment (TC.EXE) also supports multiple library directories (under the **O**ptions/**D**irectories/**I**nclude directories and the **O**ptions/**D**irectories/**L**ibrary directories menu items), using the "ganged entry" syntax.

Implicit vs. User-specified Library Files

Turbo C recognizes two types of library files: *implicit* and *user-specified* (also known as *explicit* library files).

- Implicit library files are the ones Turbo C automatically links in. These are the C*x*.LIB files, EMU.LIB or FP87.LIB, MATH*x*.LIB, and the start-up object files (C0*x*.OBJ).
- User-specified library files are the ones you explicitly list on the command line or in a project file; these are file names with an .LIB extension.

The Include and Library File-Search Algorithms

The Turbo C include file search algorithms search for the `#include` files listed in your source code in the following way:

- If you put an `#include <somefile.h>` statement in your source code, Turbo C will search for SOMEFILE.H only in the specified include directories.
- If, on the other hand, you put an `#include "somefile.h"` statement in your code, Turbo C will search for SOMEFILE.H first in the current directory; if it does not find the header file there, it will then search in the include directories specified in the command line.

The library file search algorithms are similar to those for include files:

- ***Implicit libraries:*** Turbo C searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for `#include <somefile.h>`.
- ***Explicit libraries:*** Where Turbo C searches for explicit (user-specified) libraries depends in part on how you list the library file name.
  - If you list an explicit library file name with no drive or directory (like this: `mylib.lib`), Turbo C will search for that library in the current directory first. Then (if the first search was unsuccessful), it will look in the specified library directories; this is similar to the search algorithm for `#include "somefile.h"`.
  - If you list a user-specified library with drive and/or directory information (like this: `c:mystuff\mylib1.lib`), Turbo C will search *only* in the location you explicitly listed as part of the library path name, and not in the specified library directories.

The library-search algorithms in Turbo C version 2.0 are upwardly compatible with those of earlier versions, which means that your code written under earlier versions will work without problems in version 2.0.

Using –L and –I in Configuration Files

If you do not understand how to use TURBOC.CFG (the command-line configuration file) with TCC.EXE, refer to the section "The TURBOC.CFG File" in Chapter 3 of the *Turbo C User's Guide*.

The `-L` and `-I` options you list on the command line take priority over those in the configuration file. The section on "The TURBOC.CFG File" in Chapter 3 of the *Turbo C User's Guide* describes how this works.

An Example With Notes

Here is an example of using a TCC command line that incorporates multiple library directories (`-L`) and include directories (`-I`) options.

1. Your current drive is C: and your current directory is C:\TURBOC, where TCC.EXE resides. Your A drive's current position is A:\ASTROLIB.
2. Your include files (.H or "header" files) are located in C:\TURBOC\INCLUDE.
3. Your startup files (C0T.OBJ, C0S.OBJ, ... , C0H.OBJ) are in C:\TURBOC.
4. Your standard Turbo C library files (CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ... , EMU.LIB, FP87.LIB, etc.) are in C:\TURBOC\LIB.
5. Your custom library files for star systems (which you created and manage with TLIB) are in C:\TURBOC\STARLIB. One of these libraries is PARX.LIB.
6. Your third-party-generated library files for quasars are in the A drive in \ASTROLIB. One of these libraries is WARP.LIB.

Under this configuration you enter the following TCC command line:

```
tcc -mm -Llib;starlib -Iinclude orion umaj parx.lib a:\astrolib\warp.lib
```

TCC will compile ORION.C and UMAJ.C to .OBJ files, then link them with the medium model start-up code (C0M.OBJ), the medium model libraries (CM.LIB, MATHM.LIB), the standard floating-point emulation library

(EMU.LIB), and the user-specified libraries (PARX.LIB and WARP.LIB), producing an executable file named ORION.EXE.

The compiler will search C:\TURBOC\INCLUDE for the include files in your source code.

It will search for the startup code in C:\TURBOC (then stop because they're there); it will search for the standard libraries in C:\TURBOC\LIB (search ends because they're there).

When it searches for the user-specified library PARX.LIB, the compiler first looks in the current directory, C:\TURBOC. Not finding the library there, the compiler then searches the library directories in order: first C:\TURBOC\LIB, then C:\TURBOC\STARLIB (where it locates PARX.LIB).

For the library WARP.LIB, an explicit path is given (A:\ASTROLIB\WARP.LIB), so the compiler only looks there.

A P P E N D I X

D

Turbo C Utilities

Your Turbo C package supplies much more than just two versions of the fastest C compiler available. It also provides seven powerful standalone utilities. You can use these standalone utilities with your Turbo C files as well as with your other modules.

These highly useful adjuncts to Turbo C are

- CPP (the Turbo C Preprocessor)
- MAKE (including the TOUCH utility; the standalone program manager
- TLINK (the Turbo Linker)
- TLIB (the Turbo Librarian)
- GREP (a file-search utility)
- BGIOBJ (a conversion utility for graphics drivers and fonts)
- OBJXREF (an object module cross-referencer)

This appendix explains what each utility is and illustrates, with code and command-line examples, how to use them.

CPP: The Turbo C Preprocessor Utility

The CPP preprocessor produces a listing file of a C source program in which include files and define macros have been expanded. It is not needed for normal compilations of C programs at all.

Often, when the compiler reports an error inside a macro or an include file, you can get more information about what the error is if you can see the results of the macro expansions or the include files. In many multi-pass compilers, a separate pass performs this work, and the results of the pass can be examined.

Since Turbo C uses an integrated single-pass compiler, CPP supplies the first-pass functionality found in other compilers. In addition, you can use CPP as a macro preprocessor.

You use CPP just as you would use TCC, the standalone compiler. CPP reads the same TURBOC.CFG file for default options, and accepts the same command-line options as TCC.

The TCC options that don't pertain to CPP are simply ignored by CPP. To see the list of arguments handled by CPP, type

```
cpp
```

at the DOS prompt.

With one exception, the file names listed on the CPP command line are treated like they are in TCC, with wildcards allowed. The exception to this is that all files are treated as C source files. There is no special treatment for .OBJ, .LIB, or .ASM files.

For each file processed by CPP, the output is written to a file in the current directory (or the output directory named by the `-n` option) with the same name as the source name but with an extension of .I.

This output file is a text file containing each line of the source file and any include files. Any preprocessing directive lines have been removed, along with any conditional text lines excluded from the compile. Unless you use a command-line option to specify otherwise, text lines are prefixed with the file name and line number of the source or include file the line came from. Within a text line, any macros are replaced with their expansion text.

The resulting output of CPP cannot be compiled because of the file name and line number prefix attached to each source line.

CPP as a Macro Preprocessor

The `-P` option to CPP tells it to prefix each line with the source file name and line number. If `-P-` is given, however, CPP omits this line number information. With this option turned off, CPP can be used as a macro preprocessor; the resulting .I file can then be compiled with TC or TCC.

An Example

The following simple program illustrates how CPP preprocesses a file, first with -P selected, then with -P-.

Source file: HELLOJOE.C

```
/* This is an example of the output of CPP */
#define NAME "Joe Smith"
#define BEGIN {
#define END   }

main()
BEGIN
   printf("%s\n", NAME);
END
```

Command Line Used to Invoke CPP as a Preprocessor:

```
cpp hellojoe.c
```

Output:

```
hellojoe.c 2:
hellojoe.c 3:
hellojoe.c 4:
hellojoe.c 6: main()
hellojoe.c 7: {
hellojoe.c 8:    printf("%s\n","Joe Smith");
hellojoe.c 9: }
```

Command Line Used to Invoke CPP as a Macro Preprocessor:

```
cpp -P- hellojoe.c
```

Output:

```
main()
{
   printf("%s\n","Joe Smith");
}
```

The Standalone MAKE Utility

Turbo C's MAKE utility is an intelligent program manager that—given the proper instructions—does all the work necessary to keep your programs up-to-date. When you run MAKE, it performs the following tasks for you:

- Reads a special MAKEFILE that you have created. This MAKEFILE tells it which .OBJ and library files have to be linked to create your executable (.EXE) file, and which source and header files have to be compiled to create each .OBJ file.
- Checks the time and date of each .OBJ file with the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the .OBJ file must be recompiled.
- Calls TCC to recompile the .OBJ file.
- Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of your executable file.
- If any of the .OBJ files is later than the .EXE file, calls TLINK, the Turbo Linker, to recreate the .EXE file.

In fact, MAKE can do far more than keep your programs current. It can make backups, pull files out of different subdirectories, and even automatically run your programs should the data files that they use be modified. As you use MAKE more and more, you'll see new and different ways it can help you to manage your program development.

MAKE is a standalone utility; it is different from Project-Make, which is part of the Integrated Environment.

In this section we describe how to use standalone MAKE with TCC and TLINK.

A Quick Example

Let's start off with an example to illustrate MAKE's usefulness. Suppose you're writing some programs to help you display information about nearby star systems. You have one program—GETSTARS—that reads in a text file listing star systems, does some processing on it, then produces a binary data file with the resulting information in it.

GETSTARS uses certain definitions, stored in STARDEFS.H, and certain routines, stored in STARLIB.C (and declared in STARLIB.H). In addition, the program GETSTARS itself is broken up into three files:

- GSPARSE.C
- GSCOMP.C
- GETSTARS.C

The first two files, GSPARSE and GSCOMP, have corresponding header files (GSPARSE.H and GSCOMP.H). The third file, GETSTARS.C has the main body of the program. Of the three files, only GSCOMP.C and GETSTARS.C make use of the STARLIB routines.

Here are the custom header files (other than the Turbo C headers that declare standard run-time library routines) needed by each .C file:

.C File	Custom Header File(s)
STARLIB.C	None
GSPARSE.C	STARDEFS.H
GSCOMP.C	STARDEFS.H,STARLIB.H
GETSTARS.C	STARDEFS.H,STARLIB.H,GSPARSE.H,GSCOMP.H

To produce GETSTARS.EXE (assuming a medium data model), you would enter the following command lines:

```
tcc -c -mm -f starlib
tcc -c -mm -f gsparse
tcc -c -mm -f gscomp
tcc -c -mm -f getstars
tlink lib\c0m starlib gsparse gscomp getstars,
   getstars, getstars, lib\emu lib\mathm lib\cm
```

Note: DOS requires that the TLINK command line all fit on one line: we show it here as two lines simply because the margins aren't wide enough to fit it all in one line.

Looking at the preceding information, you can see some *file dependencies*.

- GSPARSE, GSCOMP, and GETSTARS all depend on STARDEFS.H; in other words, if you make any changes to STARDEFS.H, then you'll have to recompile all three.
- Likewise, any changes to STARLIB.H will require GSCOMP and GETSTARS to be recompiled.
- Changes to GSPARSE.H means GETSTARS will have to be recompiled; the same is true of GSCOMP.H.
- Of course, any changes to any source code file (STARLIB.C, GSPARSE.C, etc.) means that file must be recompiled.
- Finally, if any recompiling is done, then the link has to be done again.

Quite a bit to keep track of, isn't it? What happens if you make a change to STARLIB.H, recompile GETSTARS.C, but forget to recompile GSCOMP.C? You could make a .BAT file to do the four compilations and the one linkage given above, but you'd have to do them every time you made a change. Let's see how MAKE can simplify things for you.

Creating a Makefile

A makefile is just a combination of two lists: file dependencies and the commands needed to satisfy them.

As an example, let's create a makefile for your program GETSTARS. It will look like this:

For example, let's take the lists given, combine them, massage them a little, and produce the following:

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
   tlink lib\c0m starlib gsparse gscomp getstars, getstars, \
           getstars, lib\emu lib\mathm lib\cm

getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
   tcc -c -mm -f getstars.c

gscomp.obj: gscomp.c stardefs.h starlib.h
   tcc -c -mm -f gscomp.c

gsparse.obj: gsparse.c stardefs.h
   tcc -c -mm -f gsparse.c

starlib.obj: starlib.c
   tcc -c -mm -f starlib.c
```

This just restates what was said in the previous section, but with the order reversed somewhat. Here's how MAKE interprets this file:

- The file GETSTARS.EXE depends on four files: GETSTARS.OBJ, GSCOMP.OBJ, GSPARSE.OBJ, and STARLIB.OBJ. If any of those four change, then GETSTARS.EXE must be relinked. How? By using the TLINK command.
- The file GETSTARS.OBJ depends on five files: GETSTARS.C, STARDEFS.H, STARLIB.H, GSCOMP.H, and GSPARSE.H. If any of those files change, then GETSTARS.OBJ must be recompiled by using the TCC command given.
- The file GSCOMP.OBJ depends on three files—GSCOMP.C, STARDEFS.H, and STARLIB.H—and if any of those three change, GSCOMP.OBJ must be recompiled using the TCC command given.
- The file GSPARSE.OBJ depends on two files—GSPARSE.OBJ and STARDEFS.H—and, again, must be recompiled using the TCC command given if either of those files change.
- The file STARLIB.OBJ depends on only one file—STARLIB.C—and must be recompiled via TCC if STARLIB.C changes.

What do you do with this? Type it into a file, which (for now) we'll call MAKEFILE. You're then ready to use MAKE.EXE.

Note: We have made all the rules in this example explicit in order to make the concept of dependencies clear. In most cases, you won't have to type in so much information about a program. Implicit rules and autodependencies can eliminate a lot of work in creating a MAKEFILE. See the sections below for more about these features.

Using a Makefile

Assuming you've created MAKEFILE as described above—and, of course, assuming that the various source code and header files exist—then all you have to do is type the command:

```
make
```

Simple, wasn't it? MAKE looks for MAKEFILE (you can call it something else; we'll talk about that later) and reads in the first line, describing the dependencies of GETSTARS.EXE. It checks to see if GETSTARS.EXE exists and is up-to-date.

This requires that it check the same thing about each of the files upon which GETSTARS.EXE depends: GETSTARS.OBJ, GSCOMP.OBJ, GSPARSE.OBJ, and STARLIB.OBJ. Each of those files depends, in turn, on other files, which must also be checked. The various calls to TCC are made as needed to update the .OBJ files, ending with the execution of the TLINK command (if necessary) to create an up-to-date version of GETSTARS.EXE.

What if GETSTARS.EXE and all the .OBJ files *already* exist? In that case, MAKE compares the time and date of the last modification of each .OBJ file with the time and date of its dependencies. If any of the dependency files are more recent than the .OBJ file, MAKE knows that changes have been made since the last time the .OBJ file was created and executes the TCC command.

If MAKE does update any of the .OBJ files, then when it compares the time and date of GETSTARS.EXE with them, it sees that it must execute the TLINK command to make an updated version of GETSTARS.EXE.

Stepping Through

Here's a step-by-step example to help clarify the previous description. Suppose that GETSTARS.EXE and all the .OBJ files exist, and that

GETSTARS.EXE is more recent than any of the .OBJ files, and, likewise, each .OBJ file is more recent than any of its dependencies.

If you then enter the command

```
make
```

nothing happens, since there is no need to update anything.

Now, suppose that you modify STARLIB.C and STARLIB.H, changing, say, the value of some constant. When you enter the command

```
make
```

MAKE sees that STARLIB.C is more recent than STARLIB.OBJ, so it issues the command

```
tcc -c -mm -f starlib.c
```

It then sees that STARLIB.H is more recent than GSCOMP.OBJ, so it issues the command

```
tcc -c -mm -f gscomp.c
```

STARLIB.H is also more recent than GETSTARS.OBJ, so the next command is

```
tcc -c -mm -f getstars.c
```

Finally, because of these three commands, the files STARLIB.OBJ, GSCOMP.OBJ, and GETSTARS.OBJ are all more recent than GETSTARS.EXE, so the final command issued by MAKE is

```
tlink lib\c0m starlib gsparse gscomp getstars, getstars,
        getstars, lib\emu lib\mathm lib\cm
```

which links everything together and creates a new version of the file GETSTARS.EXE. (Note that this TLINK command line must actually be one line.)

You have a good idea of the basics of MAKE: what it's for, how to create a makefile, and how MAKE interprets that file. Let's now look at MAKE in more detail.

Creating Makefiles

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up-to-date. You can create as many makefiles as you want and name them whatever you want; MAKEFILE is just the default

name that MAKE looks for if you don't specify a makefile when you run MAKE.

You create a makefile with any ASCII text editor, such as Turbo C's built-in interactive editor. All rules, definitions, and directives end with a newline; if a line is too long (such as the TLINK command in the previous example), you can continue it to the next line by placing a backslash (\) as the last character on the line.

Whitespace—blanks and tabs—is used to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Components of a Makefile

Creating a makefile is almost like writing a program, with definitions, commands, and directives. Here's a list of the constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules
- macro definitions
- directives: file inclusion directives, conditional execution directives, error detection directives, macro undefinition directives

Let's look at each of these in more detail.

Comments

Comments begin with a pound sign (#) character; the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere and never have to start in a particular column.

A backslash (\) will *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. If it precedes the #, it is no longer the last character on the line; if it follows the #, then it is part of the comment itself.

Here are some examples of comments in a makefile:

```
# makefile for GETSTARS.EXE
# does complete project maintenance
getstars.exe:  getstars.obj gscomp.obj gsparse.obj starlib.obj
# can't put a comment at the end of the next line
   tlink lib\c0m starlib gsparse gscomp getstars, getstars,\
      getstars, lib\emu lib\mathm lib\cm
# legal comment
# can't put a comment between the next two lines
getstars.obj:  getstars.c stardefs.h starlib.h gscomp.h gsparse.h
   tcc -c -mm -f getstars.c   # you can put a comment here
```

Explicit Rules

You are already familiar with explicit rules, since those are what you used in the makefile example given earlier. Explicit rules take the form

```
target [target ... ]: [source source ... ]
   [command]
   [command]
   ...
```

where *target* is the file to be updated, *source* is a file upon which *target* depends, and *command* is any valid DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more source files, and an optional list of commands to be performed. Target and source file names listed in explicit rules can contain normal DOS drive and directory specifications, but they cannot contain wildcards.

Syntax here is important. *target* must be at the start of a line (in column 1), and the source file(s) must be preceded by at least one space or tab, after the colon. Each *command* must be indented, (must be preceded by at least one blank or tab). As mentioned before, the backslash (\) can be used as a continuation character if the list of source files or a given command is too long for one line. Finally, both the source files and the commands are optional; it is possible to have an explicit rule consisting only of *target [target ...]* followed by a colon.

The idea behind an explicit rule is that the command or commands listed will create or update *target*, usually using the *source* files. When MAKE encounters an explicit rule, it first checks to see if any of the *source* files are themselves target files elsewhere in the makefile. If so, those rules are evaluated first.

Once all the *source* files have been created or updated based on other explicit (or implicit) rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *source*. If any *source* has been modified more recently than *target*, the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or to the end of the file. Blank lines are ignored.

Special Considerations

An explicit rule with no command lines following it is treated a little differently than an explicit rule with command lines.

- If an explicit rule exists for a target with commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on the files given in the explicit rule, and they also depend on any file that matches an implicit rule for the target(s).

See the following section for a discussion of implicit rules.

Examples

Here are some examples of explicit rules:

```
myprog.obj: myprog.c
   tcc -c  myprog.c

prog2.obj : prog2.c include\stdio.h
   tcc  -c -K prog2.c

prog.exe: myprog.c prog2.c include\stdio.h
   tcc -c myprog.c
   tcc -c -K prog2.c
   tlink lib\c0s myprog prog2, prog, , lib\cs
```

- The first explicit rule states that MYPROG.OBJ depends upon MYPROG.C, and that MYPROG.OBJ is created by executing the given TCC command.
- Similarly, the second rule states that PROG2.OBJ depends upon

PROG2.C and STDIO.H (in the INCLUDE subdirectory) and is created by the given TCC command.

- The last rule states that PROG.EXE depends on MYPROG.C, PROG2.C, and STDIO.H, and that should any of the three change, PROG.EXE can be rebuilt by the series of commands given. However, this may create unnecessary work, because, even if only MYPROG.C changes, PROG2.C will still be recompiled. This occurs because all of the commands under a rule will be executed as soon as that rule's target is out of date.
- If you place the explicit rule

  ```
  prog.exe: myprog.obj prog2.obj
     tlink lib\c0s myprog prog2, prog, , lib\cs
  ```

  as the first rule in a makefile and follow it with the rules given (for MYPROG.OBJ and PROG2.OBJ), only those files that need to be recompiled will be.

With explicit rules you must change your MAKEFILE every time you add or remove an include file in one of your C or assembly source files. MAKE works with TC, TCC, and TASM to eliminate this extra work. MAKE's -a command-line option will trigger an autodependency check.

TCC, TC, and TASM write include file information into the .OBJ files they create. When MAKE does an autodependency check, it reads the time and date information in the .OBJ file; all include files used to build the .OBJ file are then checked for time and date against the .OBJ file information.

For example, consider the following MAKEFILE:

```
.c.obj:
   tcc -c $*
```

Let's then assume that the following source file, called foo.c, has been compiled with TCC (version 2.0 or later):

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

void foo() {}
```

Then, if MAKE is invoked with the following command line

```
make  -a  foo.obj
```

it will check the time and date of foo.c, and also of stdio.h and dcl.h.

Implicit Rules

MAKE allows you to define *implicit* rules as well. Implicit

rules are generalizations of explicit rules. What do we mean by that?

Here's an example that illustrates the relationship between the two types of rules. Consider this explicit rule from the previous sample program:

```
starlib.obj: starlib.c
   tcc -c -mm -f starlib.c
```

This rule is a common one, because it follows a general principle: an .OBJ file is dependent on the .C file with the same file name and is created by executing TCC. In fact, you might have a makefile where you have several (or even several dozen) explicit rules following this same format.

By redefining the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.c.obj:
   tcc -c -mm -f $<
```

This rule means, "any file ending with .OBJ depends on the file with the same name that ends in .C, and the .OBJ file is created using the command

```
tcc -c -mm -f $<
```

where `$<` represents the file's name with the source (.C) extension." (The symbol `$<` is a special macro and is discussed in the next section; it will be replaced by the full name of the appropriate .C source file each time the command executes.)

The syntax for an implicit rule is:

```
.source_extension.target_extension:
    {command}
    {command}
    ...
```

where, as before, the commands are optional and must be indented.

The *source_extension* (which must begin with its period (.) in column 1) is the extension of the source file; that is, it applies to any file having the format

```
fname.source_extension
```

Likewise, the *target_extension* refers to the the file

```
fname.target_extension
```

where *fname* is the same for both files. In other words, this implicit rule replaces all explicit rules having the format:

```
fname.target_extension: fname.source_extension
    {command}
    {command}
    ...
```

for any *fname*.

Implicit rules are used if no explicit rule for a given target can be found, or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is found with the same name as the target, but with the mentioned source extension.

For example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.c.obj:
   tcc -c -ms -f $<
```

If you had a C program named RATIO.C that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
tcc -c -ms -f ratio.c
```

which, of course, does the compile step necessary to create RATIO.OBJ.

Implicit rules are also used if an explicit rule is given with no commands. Suppose, as mentioned before, you had the following implicit rule at the start of your makefile:

```
.c.obj:
   tcc -c -mm -f $<
```

You could then rewrite the last several explicit rules as follows:

```
getstars.obj: stardefs.h starlib.h gscomp.h gsparse.h
gscomp.obj: stardefs.h starlib.h
gsparse.obj: stardefs.h
```

Since you don't have explicit information on how to create these .OBJ files, MAKE applies the implicit rule defined earlier. And since STARLIB.OBJ depends only on STARLIB.C, that rule was dropped altogether from this list; MAKE automatically applies it.

If you enable autodependency checking in MAKE, you can remove all the rules that have .OBJ files as targets in the example above. With autodepencenies enabled and implicit rules, your MAKEFILE now looks like this:

```
.c.obj:
   tcc -c -mm -f $<

getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj

   tlink lib\c0m starlib gsparse gscomp getstars, getstars,\
         getstars, lib\emu lib\mathm\ lib\cm
```

Several implicit rules can be written with the same target extension, but only one such rule can apply at a time. If more than one implicit rule exists for a given target extension, each rule is checked in the order the rules appear in the makefile, until a match is found for the source extension, or until MAKE has checked all applicable rules.

MAKE uses the first implicit rule that discovers a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule are considered to be part of the command list for the rule, up to the next line that begins without white-space or to the end of the file. Blank lines are ignored. The syntax for a command line is provided later in the section "Using MAKE."

Special Considerations

Unlike explicit rules, MAKE does not know the full file name with an implicit rule. For that reason, special macros are provided with MAKE that allow you to include the name of the file being built by the rule in the commands to be executed. (See the discussion of macro definitions in this section for details.)

Examples

Here are some examples of implicit rules:

```
.c.obj:
   tcc -c $<

.asm.obj:
   tasm $* /mx;
```

In the first implicit rule example, the target files are .OBJ files and their source files are .C files. This example has one command line in the command list; command-line syntax is covered later in this section.

The second example directs MAKE to assemble a given file from its .ASM source file, using TASM with the `/mx` option.

Command Lists

We've talked about both explicit and implicit rules, and how they can have lists of commands. Let's talk about those commands and your options in setting them up.

Commands in a command list must be indented—that is, preceded by at least one blank or tab—and take the form

```
[ prefix ... ] command_body
```

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

Prefix

The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at (@) symbol or a dash (-) followed immediately by a number.

@ Prevents MAKE from displaying the command before executing it. The display is hidden even if the -s option is not given on the MAKE command line. This prefix applies only to the command on which it appears.

–num Affects how MAKE treats exit codes. If a number (*num*) is provided, then MAKE will abort processing only if the exit status exceeds the number given. In this example, MAKE will abort only if the exit status exceeds 4:

```
-4 myprog sample.x
```

If no *–num* prefix is given, MAKE checks the exit status for the command. If the status is nonzero, MAKE will stop and delete the current target file. (See . . .

– With a dash, but no number, MAKE will not check the exit status at all. Regardless of what the exit status was, MAKE will continue.

Command body

The command body is treated exactly as it would be if it were entered as a line to COMMAND.COM, with the exception that redirection and pipes are not supported.

MAKE executes the following built-in commands by invoking a copy of COMMAND.COM to perform them:

```
break     cd        chdir     cls       copy
ctty      date      del       dir       echo
erase     md        mkdir     path      prompt
rem       ren       rename    set       time
type      ver       verify    vol
```

MAKE searches for any other command name using the DOS search algorithm:

- The current directory is searched first, followed by each directory in the path.
- In each directory, first a file with the extension .COM is checked, then a .EXE, and finally a .BAT.
- If a .BAT file is found, a copy of COMMAND.COM is invoked to execute the batch file.

Obviously, if an extension is supplied in the command line, MAKE searches only for that extension.

Examples

This command will cause COMMAND.COM to execute the command:

```
cd c:\include
```

This command will be searched for using the full search algorithm:

```
tlink lib\c0s x y,z,z,lib\cs
```

This command will be searched for using only the .COM extension:

```
myprog.com geo.xyz
```

This command will be executed using the explicit file name provided:

```
c:\myprogs\fil.exe -r
```

Macros

Often certain commands, file names, or options are used again and again in your makefile. In the example at the start of this appendix, all the TCC commands used the switch `-mm`, which means to compile to the medium memory model; likewise, the TLINK command used the files C0M.OBJ, MATHM.LIB, and CM.LIB. Suppose you wanted to switch to the large memory model; what would you do? You could go through and change all the `-mm` options to `-ml`, and rename the appropriate files in the TLINK command. Or, you could define a *macro*.

A macro is a name that represents some string of characters. A macro definition gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
MDL = m
```

You've defined the macro MDL, which is now equivalent to the string m. You could now rewrite the makefile as follows:

```
MDL = m

getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
   tlink lib\c0$(MDL) starlib gsparse gscomp getstars, \
      getstars, getstars, lib\emu lib\math$(MDL) lib\c$(MDL)

getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
   tcc -c -m$(MDL) getstars.c

gscomp.obj: gscomp.c stardefs.h starlib.h
   tcc -c -m$(MDL) gscomp.c

gsparse.obj: gsparse.c stardefs.h
   tcc -c -m$(MDL) gsparse.c

starlib.obj: starlib.c
   tcc -c -m$(MDL) starlib.c
```

Everywhere a model is specified, you use the macro invocation $(MDL). When you run MAKE, $(MDL) is replaced with its expansion text, m. The result is the same set of commands you had before.

So, what have you gained? Flexibility. By changing the first line to

```
MDL = l
```

you've changed all the commands to use the large memory model. In fact, if you leave out the first line altogether, you can specify which memory model you want each time you run MAKE, using the -D (Define) option:

```
make -DMDL = l
```

This tells MAKE to treat MDL as a macro with the expansion text l.

Defining Macros

Macro definitions take the form

```
macro_name=expansion text
```

where *macro_name* is the name of the macro. *macro_name* should be a string of letters and digits with no whitespace in it, although you can have whitespace between *macro_name* and the equals sign (=). The *expansion text*

is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by newline.

If *macro_name* has previously been defined, either by a macro definition in the makefile or by the `-D` option on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macro names `mdl`, `Mdl`, and `MDL` are all different.

Using Macros

Macros are invoked in your makefile with the format

```
$(macro_name)
```

The parentheses are required for all invocations, even if the macro name is just one character long, with the exception of three special predefined macros that we'll talk about in just a minute. This construct—`$(macro_name)`—is known as a *macro invocation.*

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Special Considerations

Macros in macros: Macros cannot be invoked on the left (*macro_name*) side of a macro definition. They can be used on the right (*expansion text*) side, but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

Macros in directives: Macro invocations are expanded immediately in `!if` and `!elif` directives. If the macro being invoked in an `!if` or `!elif` directive is not currently defined, it is expanded to the value 0 (FALSE).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

Predefined Macros

MAKE comes with several special macros built in: `$d`, `$*`, `$<`, `$:`, `$.`, and `$&`. The first is a defined test macro, used in the conditional directives `!if` and `!elif`; the others are file name macros, used in explicit and implicit rules. In addition, the current SET environment strings are automatically loaded as macros, and the macro __MAKE__ is defined to be 1 (one).

Defined Test Macro ($d) The defined test macro `$d` expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in `!if` and `!elif` directives.

For example, suppose you want to modify your makefile so that if you don't specify a memory model, it'll use the medium one. You could put this at the start of your makefile:

```
!if !$d(MDL)            # if MDL is not defined
MDL=m                   # define it to m (MEDIUM)
!endif
```

If you invoke MAKE with the command line

```
make -DMDL=l
```

then `MDL` is defined as `l`. If, however, you just invoke MAKE by itself:

```
make
```

then `MDL` is defined as `m`, your "default" memory model.

Various File Name Macros

The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built.

Base File Name Macro ($*)

The base file name macro is allowed in the commands for an explicit or an implicit rule. This macro (`$*`) expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.C
$* expands to A:\P\TESTFILE
```

For example, you could modify the explicit GETSTARS.EXE rule already given to look like this:

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
   tlink lib\c0$(MDL) starlib gsparse gscomp $*, $*, $*, \
      lib\emu lib\math$(MDL) lib\c$(MDL)
```

When the command in this rule is executed, the macro `$*` is replaced by the target file name (sans extension), `getstars`. For implicit rules, this macro is very useful.

For example, an implicit rule for TCC might look like this (assuming that the macro `MDL` has been or will be defined, and that you are not using floating-point routines):

```
.c.obj:
   tcc -c -m$(MDL) $*
```

Full File Name Macro ($<)

The full file name macro (`$<`) is also used in the commands for an explicit or implicit rule. In an explicit rule, `$<` expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.C
$< expands to A:\P\TESTFILE.C
```

For example, the rule

```
starlib.obj: starlib.c
    copy $< \oldobjs
    tcc -c $*
```

will copy STARLIB.OBJ to the directory \OLDOBJS before compiling STARLIB.C.

In an implicit rule, `$<` takes on the file name plus the source extension. For example, the previous implicit rule

```
.c.obj:
   tcc -c $*.c
```

can be rewritten as

```
.c.obj:
   tcc -c $<
```

File Name Path Macro ($:)

This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.C
$: expands to A:\P\
```

File Name and Extension Macro ($.)

This macro expands to the file name, with extension but without the path name, like this:

```
File name is A:\P\TESTFILE.C
$. expands to TESTFILE.C
```

File Name Only Macro ($&)

This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.C
$& expands to TESTFILE
```

Directives

Turbo C's MAKE allows something that other versions of MAKE don't: directives similiar to those allowed for C itself. You can use these directives to include other makefiles, to make the rules and commands conditional, to print out error messages, and to "undefine" macros.

Directives in a makefile begin with an exclamation point (!) as the first character of the line, unlike C preprocessor statements, which begin with the pound sign (#). Here is the complete list of MAKE directives:

```
!include
!if
!else
!elif
!endif
!error
!undef
```

File-Inclusion Directive

A file-inclusion directive (`!include`) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

```
!include " filename "
```

These directives can be nested to any depth. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file MODEL.MAC which contained the following:

```
!if !$d(MDL)
MDL=m
!endif
```

You could then make use of this conditional macro definition in any makefile by including the directive

```
!include "MODEL.MAC"
```

When MAKE encounters the `!include` directive, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional Execution Directives

Conditional execution directives (`!if`, `!elif`, `!else`, and `!endif`) give the programmer a measure of flexibility in constructing makefiles. Rules and macros can be made conditional, so that a command-line macro definition (using the `-D` option) can enable or disable sections of the makefile.

The format of these directives parallels that of the C preprocessor:

```
!if expression
[ lines ]
!endif
```

```
!if expression
[ lines ]
!else
[ lines ]
!endif
```

```
!if expression
[ lines ]
!elif expression
[ lines ]
!endif
```

Note: `[lines]` can be any of the following statement types:

```
macro_definition
explicit_rule
implicit_rule
include_directive
if_group
error_directive
undef_directive
```

The conditional directives form a group, with at least an `!if` directive beginning the group and an `!endif` directive closing the group.

- One `!else` directive can appear in the group.
- `!elif` directives can appear between the `!if` and any `!else` directives.

- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested to any depth.

Any rules, commands, or directives must be complete within a single source file.

Any `!if` directives must have matching `!endif` directives within the same source file. Thus the following include file is illegal, regardless of what is contained in any file that might include it, because it does not have a matching `!endif` directive:

```
!if $(FILE_COUNT) > 5
   some rules
!else
   other rules
<end-of-file>
```

Expressions Allowed in Conditional Directives

The expression allowed in an `!if` or an `!elif` directive uses a C-like syntax. The expression is evaluated as a simple 32-bit signed integer expression.

Numbers can be entered as decimal, octal, or hexadecimal constants. For example, these are legal constants in an expression:

```
4536   # decimal constant
0677   # octal constant
0x23aF # hexadecimal constant
```

An expression can use any of the following unary operators:

–	negation
~	bit complement
!	logical not

An expression can use any of the following binary operators:

+	addition
–	subtraction
*	multiplication
/	division
%	remainder
>>	right shift
<<	left shift
&	bitwise and
|	bitwise or
^	bitwise exclusive or
&&	logical and
||	logical or
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
==	equality
!=	inequality

An expression can contain the following ternary operator:

? : The operand before the ? is treated as a test.

If the value of that operand is nonzero, then the second operand (the part between the ? and :) is the result. If the value of the first operand is zero, the value of the result is the value of the third operand (the part after the :).

Parentheses can be used to group operands in an expression. In the absence of parentheses, binary operators are grouped according to the same precedence given in the C language.

As in C, for operators of equal precedence, grouping is from left to right, except for the ternary operator (? :), which is right to left.

Macros can be invoked within an expression, and the special macro `$d()` is recognized. After all macros have been expanded, the expression must have proper syntax. Any words in the expanded expression are treated as errors.

Error Detection Directive

The error detection directive (`!error`) causes MAKE to stop and print a fatal diagnostic containing the text after `!error`. It takes the format

```
!error any_text
```

This directive is designed to be included in conditional directives to allow a user-defined abortion condition. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(MDL)
# if MDL is not defined
!error MDL not defined
!endif
```

If you reach this spot without having defined `MDL`, then MAKE will stop with this error message:

```
Fatal makefile 5: Error directive: MDL not defined
```

Macro Undefinition Directive

The macro undefinition directive (`!undef`) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is:

```
!undef macro_name
```

Using MAKE

You now know a lot about how to write makefiles; now's the time to learn how to use them with MAKE.

Command-Line Syntax

The simplest way to use MAKE is to type the command

```
make
```

at the DOS prompt. MAKE then looks for MAKEFILE; if it can't find it, it looks for MAKEFILE.MAK; if it can't find that, it halts with an error message.

What if you want to use a file with a name other than MAKEFILE or MAKEFILE.MAK? You give MAKE the file (`-f`) option, like this:

```
make -fstars.mak
```

The general syntax for MAKE is

```
make option option ... target target ...
```

where *option* is a MAKE option (discussed later), and *target* is the name of a target file to be handled by explicit rules.

Here are the syntax rules:

- The word *make* is followed by a space, then a list of make options.
- Each make option must be separated from its adjacent options by a space. Options can be placed in any order, and any number of these options can be entered (as long as there is room in the command line).
- After the list of make options comes a space, then an optional list of targets.
- Each target must also be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, recompiling their constituents as necessary.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they will be built as necessary.

Here are some more examples of MAKE command lines:

```
make -n -fstars.mak
make -s
make -Iinclude -DMDL = c
```

A Note About Stopping MAKE

MAKE will stop if any command it has executed is aborted via a control-break. Thus, a *Ctrl-C* will stop the currently executing command and MAKE as well.

The BUILTINS.MAK File

You will often find that there are MAKE macros and rules (usually implicit ones) that you use again and again. There are three ways of handling them. First, you can put them in every makefile you create. Second, you can put them all in one file and use the `!include` directive in each makefile you create. Third, you can put them all in a file named BUILTINS.MAK.

Each time you run MAKE, it looks for a file named BUILTINS.MAK; if it finds the file, MAKE reads it in before handling MAKEFILE (or whichever makefile you want it to process).

The BUILTINS.MAK file is intended for any rules (usually implicit rules) or macros that will be commonly used in files anywhere on your computer.

There is no requirement that any BUILTINS.MAK file exist. If MAKE finds a BUILTINS.MAK file, it interprets that file first. If MAKE cannot find a BUILTINS.MAK file, it proceeds directly to interpreting MAKEFILE (or whatever makefile you specify).

How MAKE Searches for BUILTINS.MAK and Makefiles

The first place MAKE searches for BUILTINS.MAK is the current directory. If it's not there, *and* if you're running under DOS 3.0 or higher, MAKE will then search the Turbo C directory, where MAKE.EXE resides. You should place the BUILTINS.MAK file in the same directory as the MAKE.EXE file.

MAKE always searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. The two files have identical syntax rules.

MAKE also searches for any `!include` files in the current directory. If you use the `-I` (Include) option, it will also search in the directory specified with the `-I` option.

MAKE Command-line Options

We've alluded to several of the MAKE command-line options; now we'll present a complete list of them. Note that case (upper or lower) *is* significant; the option `-d` is not a valid substitution for `-D`.

–a	Generates an autodependency check.
–D*identifier*	Defines the named identifier to the string consisting of the single character 1.
–D*iden=string*	Defines the named identifier *iden* to the string after the equal sign. The string cannot contain any spaces or tabs.
–I*directory*	MAKE will search for include files in the indicated directory (as well as in the current directory).
–U*identifier*	Undefines any previous definitions of the named identifier.

–s	Normally, MAKE prints each command as it is about to be executed. With the `-s` option, no commands are printed before execution.
–n	Causes MAKE to print the commands, but not actually perform them. This is useful for debugging a makefile.
–f*filename*	Uses *filename* as the MAKE file. If *filename* does not exist, and no extension is given, tries filename.mak.
–? or **–h**	Print help message.

MAKE Error Messages

MAKE diagnostic messages fall into two classes: fatal errors and errors. When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart the compilation. Errors indicate some sort of syntax or semantic error in the source makefile. MAKE completes interpreting the makefile and then stops.

Fatal Error Messages

XXXXXXXX does not exist – don't know how to make it

This message is issued when MAKE encounters a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.

Error directive: XXXX

This message is issued when MAKE processes an `#error` directive in the source file. The text of the directive is displayed in the message.

Incorrect command-line argument: XXX

This error occurs if MAKE is executed with incorrect command-line arguments.

Not enough memory

This error occurs when the total working storage has been exhausted. You should perform your make on a machine with more memory. If you already have 640K in your machine, you may have to simplify the source file.

Unable to execute command

This message is issued because a command failed to execute. This could be because the command file could not be found, or because it was misspelled, or less likely because the command itself exists but has been corrupted.

Unable to open makefile

This message is issued when the current directory does not contain a file named MAKEFILE and there is no MAKEFILE.MAK.

Errors

Bad file name format in include statement

Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.

Bad undef statement syntax

An `!undef` statement must contain a single identifier and nothing else as the body of the statement.

Character constant too long

Character constants can be only one or two characters long.

Command arguments too long

The arguments to a command executed by MAKE were more than 127 characters—a limit imposed by DOS.

Command syntax error

This message occurs if:

- The first rule line of the makefile contained any leading whitespace.
- An implicit rule did not consist of `.ext.ext:`.
- An explicit rule did not contain a name before the : character.
- A macro definition did not contain a name before the = character.

Division by zero

A divide or remainder in an `!if` statement has a zero divisor.

Expression syntax error in `!if` statement

The expression in an `!if` statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

File name too long

The file name given in an `!include` directive was too long for the compiler to process. File names in DOS must be no more than 64 characters long.

Illegal character in constant expression X

MAKE encountered some character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.

Illegal octal digit

An octal constant was found containing a digit of 8 or 9.

Macro expansion too long

A macro cannot expand to more than 4096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

Misplaced elif statement

An `!elif` directive was encountered without any matching `!if` directive.

Misplaced else statement

An `!else` directive was encountered without any matching `!if` directive.

Misplaced endif statement

An `!endif` directive was encountered without any matching `!if` directive.

No file name ending

The file name in an include statement was missing the correct closing quote or angle bracket.

Redefinition of target XXXXXXXX

The named file occurs on the left-hand side of more than one explicit rule.

Unable to open include file XXXXXXXXX.XXX

The named file could not be found. This could also be caused if an include file included itself. Check whether the named file exists.

Unexpected end of file in conditional started on line #

The source file ended before MAKE encountered an `!endif`. The `!endif` was either missing or misspelled.

Unknown preprocessor statement

A ! character was encountered at the beginning of a line, and the statement name following was not `error`, `undef`, `if`, `elif`, `include`, `else`, or `endif`.

The TOUCH Utility

There are times when you want to force a particular target file to be recompiled or rebuilt, even though no changes have been made to its sources. One way to do this is to use the TOUCH utility included with Turbo C. TOUCH changes the date and time of one or more files to the current date and time, making it "newer" than the files that depend on it.

To force a target file to be rebuilt, *touch* one of the files that target depends on. To touch a file (or files), enter

```
touch filename [filename ... ]
```

at the DOS prompt. TOUCH will then update the file's creation date(s).

Once you do this, you can invoke MAKE to rebuild the touched target file(s). (You can use the DOS wildcards * and ? with TOUCH.)

Turbo Link

In the Turbo C Integrated Development Environment (TC) the linker is built in. For the command-line version of Turbo C (TCC), the linker is invoked as a separate program. This separate program, TLINK, can also be used as a standalone linker.

TLINK is lean and mean; while it lacks some of the bells and whistles of other linkers, it is extremely fast and compact.

By default, TCC calls TLINK when compilation is successful; TLINK then combines object modules and library files to produce the executable file.

In this section, we describe how to use TLINK as a standalone linker.

Invoking TLINK

You can invoke TLINK at the DOS command line by typing `tlink` with or without parameters.

When it is invoked without parameters, TLINK displays a summary of parameters and options that looks like this:

```
Turbo Link  Version 2.0 Copyright (c) 1987, 1988 Borland International
The syntax is: TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m = map file with publics
         /x = no map file at all
         /i = initialize all segments
         /l = include source line numbers
         /s = detailed map of segments
         /n = no default libraries
         /d = warn if duplicate symbols in libraries
         /c = lower case significant in symbols
         /3 = enable 32-bit processing
         /v = include full symbolic debug information
         /e = ignore Extended Dictionary
         /t = generate COM file
```

In TLINK's summary display, the line

```
The syntax is: TLINK objfiles, exefile, mapfile, libfiles
```

specifies that you supply file names *in the given order*, separating the file *types* with commas.

For example, if you supply the command line

```
tlink /c mainline wd ln tx,fin,mfin,lib\comm lib\support
```

TLINK will interpret it to mean that

- Case is significant during linking (/c).
- The .OBJ files to be linked are MAINLINE.OBJ, WD.OBJ, LN.OBJ, and TX.OBJ.
- The executable program name will be FIN.EXE.
- The map file is MFIN.MAP.
- The library files to be linked in are COMM.LIB and SUPPORT.LIB, both of which are in subdirectory LIB.

TLINK appends extensions to file names that have none:

- .OBJ for object files
- .EXE for executable files
- .MAP for map files
- .LIB for library files

Be aware that where no .EXE file name is specified, TLINK derives the name of the executable file by appending .EXE to the first object file name listed. If for example, you had not specified FIN as the .EXE file name in the previous example, TLINK would have created MAINLINE.EXE as your executable file.

Note that when the `/t` option is used, the executable file extension defaults to .COM rather than .EXE.

TLINK always generates a map file, unless you explicitly direct it not to by including the `/x` option on the command line.

- If you give the `/m` option, the map file includes publics.
- If you give the `/s` option, the map file is a detailed segment map.

These are the rules TLINK follows when determining the name of the map file.

- If no .MAP file is specified, TLINK derives the map file name by adding a .MAP extension to the .EXE file name. (The .EXE file name can be given on the command line or in the response file; if no .EXE name is given, TLINK will derive it from the name of the first .OBJ file.)
- If a map file name is specified in the command line (or in the response file), TLINK adds the .MAP extension to the given name.

Note that even if you specify a map file name, if the `/x` option is specified then no map file will be created at all.

Using Response Files

TLINK lets you supply the various parameters on the command line, in a response file, or in any combination of the two.

A response file is just a text file that contains the options and/or file names that you would usually type in after the name TLINK on your command line.

Unlike the command line, however, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the previous command-line example as a response file, FINRESP, like this:

```
/c mainline wd+
ln tx,fin
mfin
lib\comm lib\support
```

You would then enter your TLINK command as:

```
tlink @finresp
```

Note that you must precede the file name with an "at" character (@) to indicate that the next name is a response file.

Alternately, you may break your link command into multiple response files. For example, you can break the previous command line into the following two response files:

File Name	Contents
LISTOBJS	mainline+ wd+ ln tx
LISTLIBS	lib\comm+ lib\support

You would then enter the TLINK command as:

```
tlink /c @listobjs,fin,mfin,@listlibs
```

Using TLINK with Turbo C Modules

Turbo C supports six different memory models: tiny, small, compact, medium, large, and huge. When you create an executable Turbo C file using TLINK, you must include the initialization module and libraries for the memory model being used.

The general format for linking Turbo C programs with TLINK is

```
tlink C0x <myobjs>, <exe>,[map],<mylibs> [emu|fp87 mathx] Cx
```

where these *<filenames>* represent the following:

<myobjs>	=	the .OBJ files you want linked
<exe>	=	the name to be given the executable file
[*map*]	=	the name to be given the map file (optional)
<mylibs>	=	the library files you want included at link time

The other file names on this general TLINK command line represent Turbo C files, as follows:

`C0x`	=	initialization module for memory model *t, s, c, m, l,* or *h*
`emu|fp87`	=	the floating-point libraries (choose one)
`mathx`	=	math library for memory model *s, c, m, l,* or *h*
`Cx`	=	run-time library for memory model *s, c, m, l,* or *h*

Note: If you are using the tiny model, and you want TLINK to produce a .COM file, you must also specify the `/t` option.

Initialization Modules

The initialization modules have the name C0*x*.OBJ, where *x* is a single letter corresponding to the model: *t, s, c, m, l, h*. Failure to link in the appropriate initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved and/or that no stack has been created.

The initialization module must also appear as the first object file in the list. The initialization module arranges the order of the various segments of the program. If it is not first, the program segments may not be placed in memory properly, causing some frustrating program bugs.

Be sure that you give an explicit .EXE file name on the TLINK command line. Otherwise, your program name will be C0*x*.EXE—probably not what you wanted!

Libraries

After your own libraries, the libraries of the corresponding memory model must also be included in the link command. These libraries must appear in a specific order; a floating-point library with the appropriate math library (these are optional), and the corresponding run-time library. We discuss those libraries in that order here.

If you are using any Turbo C graphics functions, you must link in graphics.lib. The graphics library is independent of memory models.

If your Turbo C program uses any floating-point, you must include a floating-point library (EMU.LIB or FP87.LIB) plus a math library (MATH*x*.LIB) in the link command.

Turbo C's two floating-point libraries are independent of the program's memory model.

- If you want to include floating-point emulation logic so that the program will work both on machines with and without a math coprocessor (8087 or 80287) chip, you must use EMU.LIB.
- If you know that the program will always be run on a machine with a math coprocessor chip, the FP87.LIB library will produce a smaller and somewhat faster executable program.

The math libraries have the name MATH*x*.LIB, where *x* is a single letter corresponding to the model: *s*, *c*, *m*, *l*, *h* (the tiny and small models share the library MATHS.LIB).

You can always include the emulator and math libraries in a link command line. If your program does no floating-point work, nothing from those libraries will be added to your executable program file. However, if you know there is no floating-point work in your program, you can save time in your links by excluding those libraries from the command line.

You must always include the C run-time library for the program's memory model. The C run-time libraries have the name C*x*.LIB, where *x* is a single letter corresponding to the model, as before.

Note: If you are using floating-point operations, you must include the math and emulator libraries *before* the C run-time library. Failure to do this could result in a failed link.

Using TLINK with TCC

You can also use TCC, the standalone Turbo C compiler, as a "front end" to TLINK that will invoke TLINK with the correct startup file, libraries, and executable-program name.

To do this, you give file names on the TCC command line with explicit .OBJ and .LIB extensions. For example, given the following TCC command line

```
tcc -mx mainfile.obj subl.obj mylib.lib
```

TCC will invoke TLINK with the files C0*x*.OBJ, EMU.LIB, MATH*x*.LIB and C*x*.LIB (initialization module, default 8087 emulation library, math library and run-time library for memory model *x*). TLINK will link these along

with your own modules MAINLINE.OBJ and SUB1.OBJ, and your own library MYLIB.LIB.

Note: When TCC invokes TLINK, it always uses the `/c` (case-sensitive link) option (unless it is overridden with `-l-c`).

TLINK Options

TLINK options can occur anywhere on the command line. The options consist of a slash (/) followed by the option-specifying letter (*m*, *x*, *i*, *l*, *s*, *n*, *d*, *c*, *3*, *v*, *e*, or *t*).

If you have more than one option, spaces are not significant (`/m/c` is the same as `/m  /c`), and you can have them appear in different places on the command line. The following sections describe each of the options.

The /x, /m, /s Options

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error messages produced during the link.

If you want to create a more complete map, the `/m` option will add a list of public symbols to the map file, sorted in increasing address order,. This kind of map file is useful in debugging. Many debuggers, such as SYMDEB, can use the list of public symbols to allow you to refer to symbolic addresses when you are debugging.

The /s option creates a map file with segments, public symbols and the program start address just like the /m option did, but also adds a detailed segment map. The following is an example of a detailed segment map:

```
                         [Detailed map of segments]

Address      Length   Class    Segment Name   Group      Module     Alignment/
             (Bytes)                                                Combining
-------------------------------------------------------------------------------
0000:0000    0E5B     C=CODE   S=SYMB_TEXT    G=(none)   M=SYMB.C   ACBP=28
00E5:000B    2735     C=CODE   S=QUAL_TEXT    G=(none)   M=QUAL.C   ACBP=28
0359:0000    002B     C=CODE   S=SCOPY_TEXT   G=(none)   M=SCOPY    ACBP=28
035B:000B    003A     C=CODE   S=LRSH_TEXT    G=(none)   M=LRSH     ACBP=20
035F:0005    0083     C=CODE   S=PADA_TEXT    G=(none)   M=PADA     ACBP=20
0367:0008    005B     C=CODE   S=PADD_TEXT    G=(none)   M=PADD     ACBP=20
036D:0003    0025     C=CODE   S=PSBP_TEXT    G=(none)   M=PSBP     ACBP=20
036F:0008    05CE     C=CODE   S=BRK_TEXT     G=(none)   M=BRK      ACBP=28
03CC:0006    066F     C=CODE   S=FLOAT_TEXT   G=(none)   M=FLOAT    ACBP=20
0433:0006    000B     C=DATA   S=_DATA        G=DGROUP   M=SYMB.C   ACBP=48
0433:0012    00D3     C=DATA   S=_DATA        G=DGROUP   M=QUAL.C   ACBP=48
0433:00E6    000E     C=DATA   S=_DATA        G=DGROUP   M=BRK      ACBP=48
0442:0004    0004     C=BSS    S=_BSS         G=DGROUP   M=SYMB.C   ACBP=48
0442:0008    0002     C=BSS    S=_BSS         G=DGROUP   M=QUAL.C   ACBP=48
0442:000A    000E     C=BSS    S=_BSS         G=DGROUP   M=BRK      ACBP=48
-------------------------------------------------------------------------------
```

For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Most of the information in the detailed segment map is self-explanatory, except for the ACBP field.

The ACBP field encodes the A (*alignment*), C (*combining*), and B (*big*) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (alignment)	00	An absolute segment.
	20	A byte aligned segment.
	40	A word aligned segment.
	60	A paragraph aligned segment.
	80	A page aligned segment.
	A0	An unnamed absolute portion of storage.
The C field (combination)	00	May not be combined.
	08	A public combining segment.
The B field (big)	00	Segment less than 64K
	02	Segment exactly 64K

The /l Option

The `/l` option creates a section in the .MAP file for source code line numbers. To use it, you must have created the .OBJ files by compiling with the `-y` (Line numbers...On) option. If you tell TLINK to create no map at all (using the `/x` option), this option will have no effect.

The /i Option

The `/i` option causes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. Note that this is not normally necessary.

The /n Option

The `/n` option causes the linker to ignore default libraries specified by some compilers. This option is necessary if the default libraries are in another directory, because TLINK does not support searching for libraries. You may want to use this option when linking modules written in another language.

The /c Option

The `/c` option forces the case to be significant in public and external symbols. For example, by default, TLINK regards *fred, Fred,* and *FRED* as equal; the `/c` option makes them different.

The /d Option

Normally, TLINK will not warn you if a symbol appears in more than one library file. If the symbol must be included in the program, TLINK will use the copy of that symbol in the first file on the command line in which it is found. Since this is a commonly used feature, TLINK does not normally warn about the duplicate symbols. The following hypothetical situation illustrates how you might want to use this feature.

Suppose you have two libraries: one called SUPPORT.LIB, and a supplemental one called DEBUGSUP.LIB. Suppose also that DEBUGSUP.LIB contains duplicates of some of the routines in SUPPORT.LIB (but the duplicate routines in DEBUGSUP.LIB include slightly different functionality, such as debugging versions of the routines). If you include DEBUGSUP.LIB *first* in the link command, you will get the debugging routines and *not* the routines in SUPPORT.LIB.

If you are not using this feature or are not sure which routines are duplicated, you may include the `/d` option. This will force TLINK to list all symbols duplicated in libraries, even if those symbols are not going to be used in the program.

The `/d` option also forces TLINK to warn about symbols that appear both in an .OBJ and a .LIB file. In this case, since the symbol that appears in the first (left-most) file listed on the command line is the one linked in, the symbol in the .OBJ file is the one that will be used.

With Turbo C, the distributed libraries you would use in any given link command do not contain any duplicated symbols. Thus while EMU.LIB and FP87.LIB (or CS.LIB and CL.LIB) obviously have duplicate symbols, they would never rightfully be used together in a single link. There are no symbols duplicated between EMU.LIB, MATHS.LIB, and CS.LIB, for example.

The /e Option

The library files that are shipped with Turbo C all contain an *Extended Dictionary* with information that enables TLINK to link faster with those libraries. This Extended Dictionary can also be added to any other library file using the /E option with TLIB (see the section on TLIB below).

Although linking with libraries that contain an Extended Dictionary is faster, there are two reasons you might want to use the /e switch, which disables the use of the Extended Dictionary:

- A program may need slightly more memory to link when an Extended Dictionary is used.
- TLINK will ignore any debugging information contained in a library that has an Extended Dictionary, unless /e is used.

The /t Option

If you compiled your file in the tiny memory model and link it with this switch toggled on, TLINK will generate a .COM file instead of the usual .EXE file.

When /t is used, the default extension for the executable file is .COM.

Note: .COM files may not exceed 64K in size, may not have any segment-relative fixups, may not define a stack segment, and must have a starting address equal to 0:100H. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100H.

The /v Option

The /v option directs TLINK to include debugging information in the executable file.

The /3 Option

The /3 option should be used when one or more of the object modules linked has been produced by TASM or a compatible asembler, and contains 32-bit code for the 80386 processor. This option increases the memory requirements of TLINK and slows down linking, so it should be used only when necessary.

Restrictions

As we said earlier, TLINK is lean and mean; it does not have an excessive supply of options. Following are the only serious restrictions to TLINK:

- Overlays are not supported.
- Common variables are only partly supported: A public must be supplied to resolve them.
- You can have a maximum of about 4000 logical segments.
- Segments that are of the same name and class should either *all* be able to be combined, or not. (Only assembler programmers might encounter this as a problem.)
- Code compiled in Microsoft C or Microsoft Fortran often cannot be linked with TLINK. This is because Microsoft languages have undocumented object record formats in their .OBJ files, which TLINK does not support.

TLINK is designed to be used with Turbo C (both the Integrated Environment and command-line versions), as well as with TASM, Turbo Prolog, and other compilers; however, it is not a general replacement for MS Link.

Error Messages

TLINK has three types of errors: fatal errors, nonfatal errors, and warnings.

- A fatal error causes TLINK to stop immediately; the .EXE file is deleted.
- A nonfatal error does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

```
<sname>      symbol name
<mname>      module name
<fname>      file name
<lsegname>   logical segment name
XXXXh        a 4-digit hexadecimal number, followed by 'h'
```

Fatal Errors

When fatal errors happen, TLINK stops and deletes the .EXE file.

XXXXXXXX.XXX: bad object file
An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was struck.

XXXXXXXX.XXX: unable to open file
This occurs if the named file does not exist or is misspelled.

Bad character in parameters
One of the following characters was encountered in the command line or in a response file:

" * < = > ? [ ] |

or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.

msdos error, ax = XXXXh
This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes where this error could occur are read, write, seek, and close.

Not enough memory
There was not enough memory to complete the link process. Try removing any terminate-and-stay-resident applications currently loaded, or reduce the size of any RAM disk currently active. Then run TLINK again.

<lsegname>: segment/group exceeds 64K
This message will occur if too much data was defined for a given data or code segment, when segments of the same name in different source files are combined. This message also occurs if a group exceeds 64K bytes when the segments of the group are combined.

Undefined symbol *name*
The function *name* was called, that does not exist in the current file or any other module or library that is being linked in. The symbol name must match *identically* the name of a defined function.

Invalid group definition
This message will generally occur only if a compiler produced a flawed object file. If this occurs in a file created by Turbo C, try recompiling the file. If the problem persists, contact Borland International.

Invalid segment definition
This message will generally occur only if a compiler produced a flawed object file. If this occurs in a file created by Turbo C, try recompiling the file. If the problem persists, contact Borland International.

Unknown option
A slash character (/) was encountered on the command line or in a response file without being followed by one of the allowed options.

Write failed, disk full?
This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.

Relocation table full
The file being linked contains more base fixups than the standard DOS relocation table can hold (base fixups are created mostly by calls to far functions).

32-bit record encountered in module XXXX : use "/3" option
This message will occur when an object file that contains special 32-bit records is encountered, and the /3 option has not been used. Simply restart TLINK with the /3 option.

Invalid entry point offset
This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.

Invalid initial stack offset
This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.

Base fixup offset overflow
This message occurs only when modules with 32-bit records are linked. It means that the offset of a base fixup exceeds the DOS limit of 64K.

Cannot generate COM file : invalid initial entry point address
The /t option has been used, but the program starting address is not equal to 100H, which is required with .COM files.

Cannot generate COM file : segment-relocatable items present
The /t option has been used, but the progam contains segment-relative fixups, which are not allowed with .COM files.

Cannot generate COM file : program exceeds 64K
The /t option has been used, but the total program size exceeds the .COM file limit.

Cannot generate COM file : stack segment present
The /t option has been used, but the program declares a stack segment, which is not allowed with .COM files.

Nonfatal Errors

TLINK has only two nonfatal errors. As mentioned, when a nonfatal error occurs, the .EXE and .MAP files are not deleted. However, these same errors are treated as fatal errors under the Integrated Environment. Here are the error messages:

XXX is unresolved in module YYY
The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check the spelling of the symbol for correctness. You will usually see this error from TLINK for Turbo C symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs.

Fixup overflow in module XXXX, at <lsegname>:xxxxh, target = <sname>
This indicates an incorrect data or code reference in an object file that TLINK must fix up at link time.

This message is most often caused by a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named module, so look in the source file of that module for the offending reference.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or some other high-level language besides Turbo C, there may be other possible causes for this message. Even in Turbo C, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Warnings

TLINK has only three warnings. The first two deal with duplicate definitions of symbols; the third, applicable to tiny model programs, indicates that no stack has been defined. Here are the messages:

Warning: XXX is duplicated in module YYY
The named symbol is defined twice in the named module. This could happen in Turbo C object files, for example, if two different **pascal** names were spelled using different cases in a source file.

Warning: XXX defined in module YYY is duplicated in module ZZZ
The named symbol is defined in each of the named modules. This could happen if a given object file is named twice in the command line, or if one of the two copies of the symbol were misspelled.

Warning: no stack
This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Turbo C, or for any application program that will be converted to a .COM file. For other programs, this indicates an error.

If a Turbo C program produces this message for any but the tiny memory model, check the C0*x* startup object files to be sure they are correct.

TLIB: The Turbo Librarian

TLIB is Borland's Turbo Librarian: It is a utility that manages libraries of individual .OBJ (object module) files. A library is a very convenient way of dealing with a collection of object modules as a single unit.

The libraries included with Turbo C were built with TLIB. You can use TLIB to build your own libraries, or to modify the Turbo C libraries, your own libraries, libraries furnished by other programmers, or commercial libraries you have purchased. You can use TLIB to

- *create* a new library from a group of object modules
- *add* object modules or other libraries to an existing library
- *remove* object modules from an existing library
- *replace* object modules from an existing library
- *extract* object modules from an existing library
- *list* the contents of a new or existing library

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension.

TLIB can also create (and include in the library file) an Extended Dictionary, which may be used to speed up linking. See the section on the / E option for details.

Although TLIB is not essential to creating executable programs with Turbo C, it is a useful programmer productivity tool. You will find TLIB indispensable for large development projects. If you work with object module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

The Advantages of Using Object Module Libraries

When you program in C, you often create a collection of useful C functions, like the functions in the C run-time library. Because of C's modularity, you are likely to split those functions into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. If you always include all the source files, on the other hand, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of C functions. When you link your program with a library, the linker scans the library and automatically selects only those modules needed for the current program. In addition, a library consumes less disk space than a collection of object module files, especially if each of the object files is small. A library also speeds up the action of the linker, because it only opens a single file, instead of one file for each object module.

The Components of a TLIB Command Line

You run TLIB by typing a TLIB command line at the DOS prompt. To get a summary of TLIB's usage, just type `TLIB` *Enter*.

The TLIB command line takes the following general form, where items listed in square brackets (`[like this]`) are optional:

```
tlib libname [/C] [/E] [operations] [, listfile]
```

This section summarizes each of these command-line components; the following sections provide details about using TLIB. For examples of how to use TLIB, refer to the "Examples" section below.

Component	Description
tlib	The command name that invokes TLIB.
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a *libname*. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both TCC and TC's project-make facility require the .LIB extension in order to recognize library files. Note that if the named library does not exist and there are *add* operations, TLIB creates the library.
/C	The case-sensitive flag. This option is not normally used; see "Advanced Operation: The /C Option" for a detailed explanation.
/E	Create Extended Dictionary; see "Creating an Extended Dictionary: The /E Option" for a detailed explanation.
operations	The list of operations TLIB performs. Operations may appear in any order. If you only want to examine the contents of the library, you don't have to give any operations at all.
listfile	The name of the file listing library contents. The *listfile* name (if given) must be preceded by a comma. If you do not give a file name, no listing is produced. The listing is an alphabetical list of each module, followed by an alphabetical list of each public symbol defined in that module.The default extension for the listfile is .LST. You may direct the listing to the screen by using the *listfile* name CON, or to the printer by using the name PRN.

The Operation List

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character *action symbol* followed by a file or module name.

White space may be used around either the action symbol or the file or module name, but it cannot appear in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to the DOS-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

Replacing a module means first removing it, then adding the replacement module.

File and Module Names

When TLIB adds an object module file to a library, the file is simply called a *module*. TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you only need to supply the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB Operations

TLIB recognizes three action symbols (-, +, *), which you can use singly or combined in pairs for a total of five distinct operations. For operations that use a pair of characters, the order of the characters in not important. The action symbols and what they do are listed here:

Action Symbol	Name	Description
+	**Add**	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	**Remove**	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message.
*	**Extract**	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.
-+ +-	**Replace**	TLIB replaces the named module with the corresponding file. This is just a shorthand for a *remove* followed by an *add* operation.
-* *-	**Extract & Remove**	TLIB copies the named module to the corresponding file name and then removes it from the library. This is just a shorthand for an *extract* followed by a *remove* operation.

A remove operation only needs a module name, but TLIB allows you to enter a full path name with drive and extension included. However, everything but the module name is ignored.

It is not possible to rename modules in a library. To rename a module, you first must extract and remove it, rename the file just created, and, finally, add it back into the library.

Creating a Library

To create a library, you simply add modules to a library that does not yet exist.

Using Response Files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using *response files*. A response file is simply an ASCII text file (which can be created with the Turbo C editor) that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one DOS command line.

To use a response file *pathname*, specify `@<pathname>` at any position on the TLIB command line.

- More than one line of text can make up a response file; you use the "and" character (`&`) at the end of a line to indicate that another line follows.
- You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest.
- You can use more than one response file in a single TLIB command line.

See "Examples" for a sample response file and a TLIB command line incorporating it.

Creating an Extended Dictionary: The /E Option

To speed up linking with large library files (such as the standard Cx.LIB library), you can direct TLIB to create an *Extended Dictionary* and append it to the library file. This dictionary contains, in a very compact form, information that is not included in the standard library dictionary. This information enables TLINK to process library files faster, especially when they are located on a floppy disk or a slow hard disk. All the libraries on the Turbo C distribution disks contain the Extended Dictionary.

To create an Extended Dictionary for a library that is being modified, just use the `/E` option when you invoke TLIB to add remove, or replace modules in the library. To create an Extended Dictionary for an existing library that you don't want to modify, use the `/E` option and ask TLIB to remove a non-existent module from the library. TLIB will display a warning that the

specified module was not found in the library, but it will also create an Extended Dictionary for the specified library. For example, enter

```
tlib /E mylib -bogus
```

Advanced Operation: The /C Option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add to the library a module that would cause a duplicate symbol, TLIB will display a message and not add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates. Since C *does* treat uppercase and lowercase letters as distinct, you should use the `/C` option to add a module to a library that includes a symbol differing *only in case* from one already in the library. The `/C` option forces TLIB to accept a module with symbols in it that differ only in case from symbols already in the library.

It may seem odd that, without the `/C` option, TLIB rejects symbols that differ only in case, especially since C is a case-sensitive language. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case.

TLINK has no problem distinguishing uppercase and lowercase symbols, and it will properly accept a library containing symbols that differ only in case. As long as you use the library only with TLINK, you can use the TLIB `/C` option without any problems.

However, if you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should not use the `/C` option.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB.

1. To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

```
tlib mylib +x +y +z
```

2. To create a library as in #1 and get a listing in MYLIB.LST too, type

   ```
   tlib mylib +x +y +z, mylib.lst
   ```

3. To get a listing in CS.LST of an existing library CS.LIB, type

   ```
   tlib cs, cs.lst
   ```

4. To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

   ```
   tlib mylib -+x +a -z
   ```

5. To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type

   ```
   tlib mylib *y, mylib.lst
   ```

6. To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

   First create a text file, ALPHA.RSP, with

   ```
   +a.obj +b.obj +c.obj &
   +d.obj +e.obj +f.obj &
   +g.obj
   ```

   Then use the TLIB command, which produces a listing file named ALPHA.LST:

   ```
   tlib alpha @alpha.rsp, alpha.lst
   ```

GREP: A File-Search Utility

GREP is a powerful search utility that can search for text in several files at once.

The general command-line syntax for GREP is:

```
grep [options] searchstring [filespec ... ]
```

For example, if you want to see in which source files you call the **setupmodem** function, you can use GREP to search the contents of all the .C files in your directory to look for the string **setupmodem**, like this:

```
grep setupmodem *.c
```

The GREP Options

In the command line, *options* are one or more single characters preceded by a dash symbol (-). Each individual character is a switch that you can turn

on or off: Type the plus symbol (+) after a character to turn the option on, or type a dash (-) after the character to turn the option off.

The default is on (the + is implied); for example, `-r` means the same thing as `-r+`. You can list multiple options individually (like this: `-i -d -l`) or you can combine them (like this: `-ild` or `-il -d`, etc.); they're all the same to GREP.

Here is a list of the option characters used with GREP and their meanings:

-c *Count only*: Only a count of matching lines is printed. For each file that contains at least one matching line, GREP prints the file name and a count of the number of matching lines. Matching lines are not printed.

-d *Directories*: For each *filespec* specified on the command line, GREP searches for all files that match the file specification, both in the directory specified *and* in all subdirectories below the specified directory. If you give a *filespec* without a path, GREP assumes the files are in the current directory.

-i *Ignore case*: GREP ignores upper/lowercase differences (case folding). GREP treats all letters *a*-*z* as being identical to the corresponding letters *A*-*Z* in all situations.

-l *List match files*: Only the name of each file containing a match is printed. After GREP finds a match, it prints the file name and processing immediately moves on to the next file.

-n *Numbers*: Each matching line that GREP prints is preceded by its line number.

-o *UNIX output format*: Changes the output format of matching lines to support more easily the UNIX style of command-line piping. All lines of output are preceded by the name of the file that contained the matching line.

-r *Regular expression search*: The text defined by *searchstring* is treated as a regular expression instead of as a literal string.

-u *Update options*: GREP will combine the options given on the command line with its default options and write these to the GREP.COM file as the new defaults. (In other words, GREP is self-configuring.) This option allows you to tailor the default option settings to your own taste.

–v *Non-match*: Only non-matching lines are printed. Only lines that *do not* contain the search string are considered to be non-matching lines.

–w *Word search*: Text found which matches the regular expression will be considered a match only if the character immediately preceding and following cannot be part of a word. The default word character set includes *A-Z*, 9-0, and the underscore (_). An alternate form of this option allows you to specify the set of legal word characters. Its form is `-w[set]`, where `set` is any valid regular expression set definition. If alphabetic characters are used to define the set, the set will automatically be defined to contain both the upper and lower case values for each letter in the set, regardless of how it is typed, even if the search is case-sensitive. If the `-w` option is used in combination with the `-u` option, the new set of legal characters is saved as the default set.

–z *Verbose*: GREP prints the file name of every file searched. Each matching line is preceded by its line number. A count of matching lines in each file is given, even if the count is zero.

Order of Precedence

Remember that each of GREP's options is a switch: its state reflects the way you last set it. At any given time, each option can only be on or off. Each occurrence of a given option on the command line overrides its previous definition. For example, you might type in the following command line:

```
grep -r -i- -d -i -r-  main( my*.c
```

Given this command line, GREP will run with the `-d` option on, the `-i` option on, and the `-r` option off.

You can install your preferred default setting for each option in GREP.COM with the `-u` option. For example, if you want GREP to always do a verbose search (`-z` on), you can install it with the following command:

```
grep -u -z
```

The Search String

The value of *searchstring* defines the pattern GREP will search for. A search string can be either a *regular expression* or a *literal string*. In a regular expression, certain characters have special meanings: they are operators

that govern the search. In a literal string, there are no operators: each character is treated literally.

You can enclose the search string in quotation marks to prevent spaces and tabs from being treated as delimiters. Matches will not cross line boundaries (a match must be contained in a single line).

An expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.

Operators in Regular Expressions

When you use the `-r` option, the search string is treated as a regular expression (not a literal expression), and the following characters take on special meanings:

- **^** A circumflex at the start of the expression matches the start of a line.
- **$** A dollar sign at the end of the expression matches the end of a line.
- **.** A period matches any character.
- ***** An expression followed by an asterisk wildcard matches zero or more occurrences of that expression. For example: in `fo*`, the `*` operates on the expression *o*; it matches *f*, *fo*, *foo*, etc. (*f* followed by zero or more *o*s), but doesn't match *fa*.
- **+** An expression followed by a plus sign matches one or more occurrences of that expression: `fo+` matches *fo*, *foo*, etc., but not *f*.
- **[ ]** A string enclosed in brackets matches any character in that string, but no others. If the first character in the string is a circumflex (`^`), the expression matches any character *except* the characters in the string. For example, `[xyz]` matches *x*, *y*, or *z*, while `[^xyz]` matches *a* and *b*, but not *x*, *y*, or *z*. You can specify a range of characters with two characters separated by a dash (-). These can be combined to form expressions (like `[a-bd-z?]` to match ? and any lowercase letter except *c*).
- **** The *backslash escape character* tells GREP to seach for the literal character that follows it. For example, `\.` matches a period instead of "any character."

Note: Four of the previously-described characters (`$`, `.`, `*`, and `+`) do not have any special meaning when used within a bracketed set. In addition,

the character ^ is only treated specially if it immediately follows the beginning of the set definition (that is, immediately after the [).

Any ordinary character not mentioned in the preceding list matches that character (> matches >, # matches #, and so on).

The File Specification

The third item in the GREP command line is *filespec*, the file specification; it tells GREP which files (or groups of files) to search. *filespec* can be an explicit file name, or a "generic" file name incorporating the DOS ? and * wildcards. In addition, you can enter a path (drive and directory information) as part of *filespec*. If you give *filespec* without a path, GREP only searches the current directory.

If you don't specify any file specifications, input to GREP must be specified by redirecting *stdin* or by piping.

Examples with Notes

The following examples assume that all of GREP's options default to off:

Example 1

Command line: `grep main( *.c`

Matches:
```
main()
mymain(
```

Does not match:
```
mymainfunc()
MAIN(i: integer);
```

Files Searched: *.C in current directory.

Note: By default, the search is case-sensitive.

Example 2

Command line: `grep -r [^a-z]main\ *( *.c`

Matches:
```
main(i:integer)
main(i,j:integer)
if (main  ()) halt;
```

Does not match:
```
mymain()
MAIN(i:integer);
```

Files Searched: *.C in current directory.

Note: The search string here tells GREP to search for the word *main* with no preceding lowercase letters (`[^a-z]`), followed by zero or more occurrences of blank spaces (`\ *`), then a left parenthesis.

Since spaces and tabs are normally considered to be command-line delimiters, you must *quote* them if you want to include them as part of a regular expression. In this case, the space after *main* is quoted with the backslash escape character. You could also accomplish this by placing the space in double quotes (`[^a-z]main" "*`).

Example 3

Command line: `grep -ri [a-c]:\\data\.fil *.c *.inc`

Matches:

```
A:\data.fil
c:\Data.Fil
B:\DATA.FIL
```

Does not match:

```
d:\data.fil
a:data.fil
```

Files Searched: *.C and *.INC in current directory.

Note: Because the backslash and period characters (\ and .) usually have special meaning in path and file names, if you want to search for them, you must place the backslash escape character immediately in front of them.

Example 4

Command line: `grep -ri [^a-z]word[^a-z] *.doc`

Matches:

```
every new word must be on a new line.
MY WORD!
word--smallest unit of speech.
In the beginning there was the WORD, and the WORD
```

Does not match:

```
Each file has at least 2000 words.
He misspells toward as toword.
```

Files Searched: *.DOC in the current directory.

Note: This format basically defines how to search for a given word.

Example 5

Command line: `grep -iw word *.doc`

Matches:

```
every new word must be on a new line  However,
MY WORD!
word: smallest unit of speech which conveys meaning.
In the beginning there was the WORD, and the WORD
```

Does not match:

```
each document contains at least 2000 words!
He seems to continually misspell "toward" as "toword."
```

Files searched: *.doc in the current directory.

Note: This format defines a basic "word" search.

Example 6

Command line:

```
grep "search string with spaces" *.doc *.asm
a:\work\myfile.*
```

Matches:

```
This is a search string with spaces in it.
```

Does not match:

```
THIS IS A SEARCH STRING WITH SPACES IN IT.
This is a search string with many spaces in it.
```

Files Searched: *.DOC and *.ASM in the current directory, and MYFILE.* in a directory called \WORK on drive A:.

Note: This is an example of how to search for a string with embedded spaces.

Example 7

Command line: `grep -rd "[ ,.:?'\"]"$ *.doc`

Matches:

```
He said hi to me.
Where are you going?
Happening in anticipation of a unique situation,
Examples include the following:
"Many men smoke, but fu man chu."
```

Does not match:

```
He said "Hi" to me
Where are you going? I'm headed to the beach this
```

Files Searched: *.DOC in the root directory and all its subdirectories on the current drive.

Note: This example searches for the characters `,.:?'` and `"` at the end of a line. Notice that the double quote within the range is preceded by an escape character so it is treated as a normal character instead of as the ending quote for the string. Also, notice how the `$` character appears

outside of the quoted string. This demonstrates how regular expressions can be concatenated to form a longer expression.

Example 8

Command line: `grep -ild " the " *.doc`
or `grep -i -l -d " the " *.doc`
or `grep -il -d " the " *.doc`

Matches:
```
Anyway, this is the time we have
do you think? The main reason we are
```

Does not match:
```
He said "Hi" to me just when I
Where are you going? I'll bet you're headed to
```

Files Searched: *.DOC in the root directory and all its subdirectories on the current drive.

Note: This example ignores case and just prints the names of any files that contain at least one match. The three command-line examples show different ways of specifying multiple options.

Example 9

Command line: `grep -w[=] = *.c`

Matches:
```
i = 5;
j=5;
i += j;
```

Does not match:
```
if (i == t) j++;
/* ================================================= */
```

Files searched: *.c in the current directory.

Note: This example redefines the current set of legal characters for a word as the assignment operator (=) only, then does a word search. It matches C assignment statements, but not equality tests.

BGIOBJ: Conversion Utility for Graphics Drivers and Fonts

BGIOBJ is a utility you can use to convert graphics driver files and character sets (stroked font files) to object (.OBJ) files. Once they're converted, you can link them into your program, making them part of the executable file. This is offered in addition to the graphics package's

dynamic loading scheme, in which your program loads graphics drivers and character sets (stroked fonts) from disk at execution time.

Linking drivers and fonts directly into your program is advantageous because the executable file contains all (or most) of the drivers and/or fonts it might need, and doesn't need to access the driver and font files on disk when running. However, linking the drivers and fonts into your executable file increases its size.

To convert a driver or font file to a linkable object file, use the BGIOBJ.EXE utility. This is the simplified syntax:

```
BGIOBJ  <source file>
```

where *<source file>* is the driver or font file to be converted to an object file. The object file created has the same file name as the source file, with the extension .OBJ; for example, EGAVGA.BGI yields EGAVGA.OBJ, SANS.CHR gives SANS.OBJ, etc.

Adding the New .OBJ Files to GRAPHICS.LIB

You should add the driver and font object modules to GRAPHICS.LIB, so the linker can locate them when it links in the graphics routines. If you don't add these new object modules to GRAPHICS.LIB, you'll have to add them to the list of files in the TC project (.PRJ) file, on the TCC command line, or on the TLINK command line. To add these object modules to GRAPHICS.LIB, invoke the Turbo Librarian (TLIB) with the following command line:

```
tlib graphics + <object file name> [ + <object file name> ... ]
```

where *<object file name>* is the name of the object file created by BGIOBJ.EXE (such as CGA, EGAVGA, GOTH, etc.); the .OBJ extension is implied, so you don't need to include it. You can add several files with one command line to save time; see the example in the following section.

Registering the Drivers and Fonts

After adding the driver and font object modules to GRAPHICS.LIB, you have to *register* all the drivers and fonts that you want linked in; you do this by calling **registerbgidriver** and **registerbgifont** in your program (before calling **initgraph**). This informs the graphics system of the presence of

those files, and ensures that they will be linked in when the executable file is created by the linker.

The registering routines each take one parameter; a symbolic name defined in GRAPHICS.H. Each registering routine returns a non-negative value if the driver or font is successfully registered.

The following table is a complete list of drivers and fonts included with Turbo C. It shows the names to be used with **registerbgidriver** and **registerbgifont**.

Driver file (*.BGI)	registerbgidriver Symbolic name	Font file (*.CHR)	registerbgifont Symbolic name
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_driver	LITT	small_font
HERC	Herc_driver	SANS	sansserif_font
ATT	ATT_driver	GOTH	gothic_font
PC3270	PC3270_driver		
IBM8514	IBM8514_driver		

An Example

Here's a complete example. Suppose you want to convert the files for the CGA graphics driver, the gothic font, and the triplex font to object modules, then link them into your program.

1. Convert the binary files to object files using BGIOBJ.EXE, as shown in the following separate command lines:

   ```
   bgiobj  cga
   bgiobj  trip
   bgiobj  goth
   ```

   This creates three files: CGA.OBJ, TRIP.OBJ, and GOTH.OBJ.
2. You can add these object files to GRAPHICS.LIB with this TLIB command line:

   ```
   tlib  graphics  +cga  +trip  +goth
   ```

3. If you don't add the object files to GRAPHICS.LIB, you must add the object file names CGA.OBJ, TRIP.OBJ, and GOTH.OBJ to your project list (if you are using Turbo C's integrated environment), or to the TCC command line. For example, the TCC command line would look like this:

   ```
   tcc  niftgraf graphics.lib  cga.obj  trip.obj  goth.obj
   ```

4. You register these files in your graphics program like this:

```
/* Header file declares CGA_driver, triplex_font, and gothic_font */
#include <graphics.h>

/* Register and check for errors (one never knows ....) */

if (registerbgidriver(CGA_driver) < 0) exit(1);
if (registerbgifont(triplex_font) < 0) exit(1);
if (registerbgifont(gothic_font)  < 0) exit(1);

/* ... */

initgraph(....);          /* initgraph should be called after registering */

/* ... */
```

If you ever get a linker error `Segment exceeds 64k` after linking in some drivers and/or fonts, refer to the following section.

The /F option

This section explains what steps to take if you get the linker error `Segment exceeds 64k` (or a similar error) after linking in several driver and/or font files (especially with small and compact model programs).

By default, the files created by BGIOBJ.EXE all use the same segment (called _TEXT). This can cause problems if your program links in many drivers and/or fonts, or when you're using the small or compact memory model.

To solve this problem, you can convert one or more of the drivers or fonts with the BGIOBJ `/F` option. This option directs BGIOBJ to use a segment name of the form *<filename>*_TEXT, so that the default segment is not overburdened by all the linked-in drivers and fonts (and, in small and compact model programs, all the program code). For example, the following two BGIOBJ command lines direct BGIOBJ to use segment names of the form EGAVGA_TEXT and SANS_TEXT.

```
bgiobj  /F  egavga
bgiobj  /F  sans
```

When you select the `/F` option, BGIOBJ also appends F to the target object file (EGAVGAF.OBJ, SANSF.OBJ, etc.), and appends *_far* to the name that will be used with **registerfarbgidriver** and **registerfarbgifont**. (For example, *EGAVGA_driver* becomes *EGAVGA_driver_far*.) For files created with `/F`, you must use these far registering routines instead of the regular **registerbgidriver** and **registerbgifont**. For example,

```
if (registerfarbgidriver(EGAVGA_driver_far) < 0) exit(1);
if (registerfarbgifont(sansserif_font_far)  < 0) exit(1);
```

Advanced BGIOBJ Features

This section explains some of BGIOBJ's advanced features, and the routines **registerfarbgidriver** and **registerfarbgifont**. Only experienced users should use these features.

This is the full syntax of the BGIOBJ.EXE utility:

```
BGIOBJ  [/F] <source> <destination> <public name> <seg-name> <seg-class>
```

Component	Description
/F or -F	This option instructs BGIOBJ.EXE to use a segment name other than _TEXT (the default), and to change the public name and destination file name. (See the previous section for a detailed discussion of /F.)
<source>	This is the driver or font file to be converted. If the file is not one of the driver/font files shipped with Turbo C, you should specify a full file name (including extension).
<destination>	This is the name of the object file to be produced. The default destination file name is *<source>*.OBJ, or *<source>*F.OBJ if you use the /F option.
<public name>	This is the name that will be used in the program in a call to **registerbgidriver** or **registerbgifont** (or their respective **far** versions) to link in the object module. The public name is the external name used by the linker, so it should be the name used in the program, prefixed with an underscore. If your program uses Pascal calling conventions, use only uppercase letters, and do not add an underscore.
<seg-name>	This is an optional segment name; the default is _TEXT (or *<filename>*_TEXT if /F is specified)
<seg-class>	This is an optional segment class; the default is CODE.

All parameters except *<source>* are optional. If you need to specify an optional parameter, all the parameters preceding it must also be specified.

If you choose to use your own public name(s), you have to add declaration(s) to your program, using one of the following forms:

```
void public_name(void);          /* if /F not used, default segment name used */

extern int far public_name[];        /* if /F used, or segment name not _TEXT */
```

In these declarations, `public_name` matches the *<public name>* you used when converting with BGIOBJ. The GRAPHICS.H header file contains declarations of the default driver and font public names; if you use those default public names you don't have to declare them as just described.

After these declarations, you have to register all the drivers and fonts in your program. If you don't use the `/F` option and don't change the default segment name, you should register drivers and fonts through **registerbgidriver** and **registerbgifont**; otherwise use **registerfarbgidriver** and **registerfarbgifont**.

Here is an example of a program that loads a font file into memory:

```
/* example of loading a font file into memory */

#include <graphics.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>

main()
{
   void    *gothic_fontp;                 /* points to the font buffer in memory */
   int      handle;                                /* file handle used for I/O */
   unsigned fsize;                                /* size of file (and buffer) */

   int errorcode;
   int graphdriver;
   int graphmode;

/* open font file */
   handle = open("GOTH.CHR", O_RDONLY|O_BINARY);
   if (handle == -1)
   {
      printf("unable to open font file 'GOTH.CHR'\n");
      exit(1);
   }
/* find out size of the file */
   fsize = filelength(handle);
/* allocate buffer */
```

```
   gothic_fontp = malloc(fsize);
   if (gothic_fontp == NULL)
   {
      printf("unable to allocate memory for font file 'GOTH.CHR'\n");
      exit(1);
   }
/* read font into memory */
   if (read(handle, gothic_fontp, fsize) != fsize)
   {
      printf("unable to read font file 'GOTH.CHR'\n");
      exit(1);
   }
/* close font file */
   close(handle);
/* register font */
   if (registerfarbgifont(gothic_fontp) != GOTHIC_FONT)
   {
      printf("unable to register font file 'GOTH.CHR'\n");
      exit(1);
   }
/* detect and initialize graphix */
   graphdriver = DETECT;
   initgraph(&graphdriver, &graphmode, "..");
   errorcode = graphresult();
   if (errorcode != grOk)
   {
      printf("graphics error: %s\n",grapherrormsg(errorcode));
      exit(1);
   }
   settextjustify(CENTER_TEXT, CENTER_TEXT);
   settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
   outtextxy( getmaxx() / 2, getmaxy() / 2, "Borland Graphics Interface (BGI)");
/* hit a key to terminate */
   getch();
/* shut down graphics system */
   closegraph();
   return(0);
}
```

OBJXREF: The Object Module Cross-Reference Utility

OBJXREF is a utility that examines a list of object files and library files and produces reports on their contents. One type of report lists definitions of

public names and references to them. The other type lists the segment sizes defined by *object modules*.

There are two categories of public names, *global variables* and *function names*. The TEST1.C and TEST2.C files in the section "Sample OBJXREF Reports" illustrate definitions of public names and external references to them.

Object modules are object (.OBJ) files produced by TC, TCC or TASM. A library (.LIB) file contains multiple object modules. An object module generated by TC is given the same name as the .C source file it was compiled from. This is also true for TCC, unless a different output file name is specifically indicated with the -o TCC command-line option.

The OBJXREF Command Line

The OBJXREF command line consists of the word `OBJXREF`, followed by a series of command-line options and a list of object and library file names, separated by a space or tab character. The syntax is as follows:

```
OBJXREF < options >  filename < filename ... >
```

The command-line options determine the kind of reports that OBJXREF will generate and the amount of detail that OBJXREF will provide. They are discussed in more detail in the section "The OBJXREF Command-Line Options" below.

Each option begins with a forward slash (/) followed by a one- or two-character option name.

Object files and library files may be specified either on the command line or in a response file. On the command line, file names are separated by a space or a tab. All object modules specified as .OBJ files are included in reports. Like TLINK, however, OBJXREF includes only those modules from .LIB files which contain a public name referenced by an .OBJ file or by a previously included module from a .LIB file.

As a general rule, you should list all the .OBJ and .LIB files that are needed if the program is to link correctly, including the startup .OBJ file and one or more C libraries.

File names may include a drive and directory path. The DOS ? and * wildcard characters may be used to identify more than one file. File names may refer to .OBJ object files or to .LIB library files. (If no file extension is given, the .OBJ extension is assumed.)

Options and file names may occur in any order in the command line.

OBJXREF reports are written to the DOS standard output. The default is the screen. The reports may be sent to a printer (as with `>LPT1:`) or to a file (as with `>lstfile`) with the DOS redirection character (>).

Entering `OBJXREF` with no file names or options produces a summary of available options.

The OBJXREF Command-Line Options

OBJXREF command-line options fall into two categories: *control options* and *report options.*

Control Options

Control options modify the default behavior of OBJXREF (the default is that none of these options are enabled).

/I Ignore case differences in public names. Use this option if you use TLINK without the `/C` option (which makes case differences significant.)

/F Include Full library. All object modules in specified .LIB files are included even if no public names they contain are referenced by an object module being processed by OBJXREF. This provides information on the entire contents of a library file. (See example 4 in the section "OBJXREF Examples.")

/V Verbose output. Lists names of files read and displays totals of public names, modules, segments, and classes.

/Z Include Zero Length Segment Definitions. Object modules may define a segment without allocating any space in it. Listing these zero length segment definitions normally makes the module size reports harder to use but it can be valuable if you are trying to remove all definitions of a segment.

Report Options

Report options govern what sort of report is generated, and the amount of detail that OBJXREF provides.

/RC **Report by Class Type:** Module sizes ordered by class type of segment

/RM **Report by Module:** Public names ordered by defining module

/RP **Report by Public Names:** Public names in order with defining module name

/RR **Report by Reference:** Public name definitions and references ordered by name. (This is the default if no report option is specified.)

/RS **Report of Module Sizes:** Module sizes ordered by segment name

/RU **Report of Unreferenced Symbol Names:** Unreferenced public names ordered by defining module

/RV **Verbose Reporting:** OBJXREF produces a report of every type

/RX **Report by External Reference:** External references ordered by referencing module name

Public names defined in .C files appear in reports with a leading underscore in the reports unless the `-U-` option was specified when the file was compiled. (**main** appears as **_main**.)

Response Files

The command line is limited by DOS to a maximum of 128 characters. If your list of options and file names will exceed this limit, you must place your file names in a *response file.*

A response file is a text file that you make with an text editor. Since you may already have prepared a list of the files that make up your program for other Turbo C programs, OBJXREF recognizes several response file types.

Response files are called from the command line using one of the following options. The response file name must follow the option without an intervening space (`/Lresp`, not `/L resp`).

More than one response file can be specified on the command line, and additional .OBJ and .LIB file names may precede or follow them.

Freeform Response Files

You can create a freeform response file with a text editor. Just list the names of all .OBJ and .LIB files needed to make your .EXE file.

To use freeform files with OBJXREF, type in each file name on the command line, preceded by a @, and separate it from other command line entries with a space or tab:

```
@filename  @filename ...
```

Note: Any file name that is listed in the response file without an extension is assumed to be a .OBJ file.

Project Files

You can also use project files of the type generated by TC as response files. In the command line, precede the project file name with /P.

```
/Pfilename
```

If the file name does not include an explicit extension, a .PRJ extension is assumed.

File names in the project file with a .C extension or no extension are interpreted as specifying the corresponding .OBJ file. You need not remove file dependencies specified inside parentheses; they are ignored by OBJXREF.

Note: By itself, the list of files in a .PRJ file does not specify a complete program—you must also specify a startup file (C0x.OBJ) and one or more Turbo C library files (mathx.lib, emu.lib, and Cx.lib, for example). In addition, you may need to use the /O command to specify the directory where OBJXREF is to look for your .OBJ files.

Linker Response Files

Files in TLINK response file format can also be used by OBJXREF. A linker response file called from the command line is preceded by /L:

```
/Lfilename
```

To see how to use one of these files, refer to Example 2 in the section "Examples of How to Use OBJXREF."

The /O Command

If you want OBJXREF to look for .OBJ files in a directory other than the current one, include the directory name on the command line, prefixed with /O:

```
/Omyobjdir
```

The /N Command

You can limit the modules, segments, classes, or public names that OBJXREF reports on by entering the appropriate name on the command line prefixed with the /N command. For example,

```
OBJXREF <filelist> /RM /NC0
```

tells OBJXREF to generate a report listing information only for the module named C0.

Sample OBJXREF Reports

Suppose you have two source files in your Turbo C directory, and wish to generate OBJXREF reports on the object files compiled from them. The source files are called TEST1.C and TEST2.C, and they look like this:

```
/* test1.c */
int i1;                                         /* defines i1 */
extern int i2;                                /* refers to i2 */
static int i3;                             /* not a public name */
extern void look(void);                     /* refers to look */

void main(void)                               /* defines main */
{
   int i4;                                 /* not a public name */

   look();                                  /* refers to look */
}
```

```
/* test2.c */
#include <process.h>
extern int i1;                                    /* refers to i1 */
int i2;                                            /* defines i2 */

void look(void)                                   /* defines look */
{
   exit(i1);                                /* refers to exit... */
}                                                  /* and to i1 */
```

The object modules compiled from them are TEST1.OBJ and TEST2.OBJ. You can tell OBJXREF what kind of report to generate about these .OBJ files by entering the file names on the command line, followed by a /R and a second letter denoting report type.

Note: The examples below show only fragments of the output.

Report by Public Names (/RP)

A report by public names lists each of the public names defined in the object modules being reported on, followed by the name of the module in which it is defined.

If you enter this on the command line:

```
OBJXREF  /RP  test1  test2
```

OBJXREF will generate a report that looks like this:

```
SYMBOL        DEFINED IN
_i1           TEST1
_i2           TEST2
_look         TEST2
_main         TEST1
```

Report by Module (/RM)

A report by module lists each object module being reported on, followed by a list of the public names defined in it.

If you enter this on the command line:

```
OBJXREF  /RM  test1  test2
```

OBJXREF will generate a report that looks like this:

```
MODULE: TEST1 defines the following symbols:
     public: _i1
     public: _main
MODULE: TEST2 defines the following symbols:
     public: _i2
     public: _look
```

Report by Reference (/RR) (Default)

A report by reference lists each public name with the defining module in parentheses on the same line. Modules that refer to this public name are listed on following lines indented from the left margin.

If you enter this on the command line:

```
OBJXREF  /RR  C0  test1  test2  CS.LIB
```

OBJXREF will generate a report that looks like this:

```
_exit (EXIT)
     C0
     TEST2
_i1 (TEST1)
     TEST2
_i2 (TEST2)
_look (TEST2)
     TEST1
_main (TEST1)
     C0
```

Report by External References (/RX)

A report by external references lists each module followed by a list of external references it contains.

If you enter this on the command line:

```
OBJXREF  /RX  C0  test1  test2  CS.LIB
```

OBJXREF will generate a report that looks like this:

```
MODULE: C0 references the following symbols:
        _main
MODULE: TEST1 references the following symbols:
        _i2
        _look
MODULE: TEST2 references the following symbols:
        _exit
        _i1
```

Report of Module Sizes (/RS)

A report by sizes lists segment names followed by a list of modules that define the segment. Sizes in bytes are given in decimal and hexadecimal notation. The word `unitialized` appears where no initial values are assigned to any of the symbols defined in the segment. Segments defined at absolute addresses in a .ASM file are flagged `Abs` to the left of the segment size.

If you enter this on the command line:

```
OBJXREF  /RS  test1  test2
```

OBJXREF will generate a report that looks like this:

```
TEST1_TEXT
        6 (00006h)   TEST1
        6 (00006h)   total
TEST2_TEXT
       10 (0000Ah)   TEST2
       10 (0000Ah)   total
_BSS
        4 (00004h)   TEST1, uninitialized
        2 (00002h)   TEST2, uninitialized
        6 (00006h)   total
```

Report by Class Type (/RC)

A report by class type lists segment size definitions by segment class. The CODE class contains instructions, DATA class contains initialized data and BSS class contains unitialized data. Segments which do not have a class type will be listed under the notation `No class type`.

If you enter this on the command line:

```
OBJXREF /RC C0 test1 test2 CS.LIB
```

OBJXREF will generate a report that looks like this:

```
BSS
        4  (00004h)   TEST1
        2  (00002h)   TEST2
      ...
      132  (00084h)   total
CODE
        6  (00006h)   TEST1
       10  (0000Ah)   TEST2
       16  (00010h)   total
DATA
      143  (0008Fh)   C0
      143  (0008Fh)   total
```

Report of Unreferenced Symbol Names (/RU)

A report of unreferenced symbol names lists modules that define public names not referenced in other modules. Such a symbol is either:

- referenced only from within the defining module, and does not need to be defined as a public symbol (in that case, if the module is in C, the keyword **static** should be added to the definition; if the module is in TASM, just remove the public definition).
- never used (therefore, it can be deleted to save code or data space

If you enter this on the command line:

```
OBJXREF /RU test1 test2
```

OBJXREF will generate a report that looks like this:

```
MODULE: TEST2 defines the unreferenced symbol _i2.
```

Verbose Reporting (/RV)

If you enter /RV on the command line, one report of each type will be generated.

Examples of How to Use OBJXREF

These examples assume that the application files are in the current directory of the default drive and that the Turbo C startup files (C0x.OBJ) and the library files are in the \TURBOC\LIB directory.

Example 1

```
C>OBJXREF \turboc\lib\c0l test1 test2 \turboc\lib\cl.lib
```

In addition to the TEST1.OBJ and TEST2.OBJ files, the Turbo C startup file \TURBOC\LIB\C0L.OBJ and the library file \TURBOC\LIB\CL.LIB are specified. Since no report type is specified, the resulting report is the default report by reference, listing public names and the modules that reference them.

Example 2

```
C>OBJXREF /RV /Ltest1.arf
```

The TLINK response file TEST1.ARF contains the same list of files as the command line in Example 1. The /RV option is specified, so a report of every type will be generated. TEST1.ARF contains

```
\turboc\lib\c0l
test1 test2
test1.exe
test1.map
\turboc\lib\cl
```

Example 3

```
C>OBJXREF /RC  B:c0s  /Ptest1  @libs
```

The TC project file TEST1.PRJ specifies TEST1.OBJ and TEST2.OBJ. The response file @libs specifies libraries on a disk in the B drive. TEST1.PRJ contains

```
test1
test2.c
```

The file LIBS contains

```
b:maths.lib b:emu.lib b:cs.lib
```

The startup and library files specified depend on the memory model and floating point options used in compilation. The /RC causes a report of class type to be output.

Example 4

```
C>OBJXREF /F /RV \turboc\lib\cs.lib
```

This example reports on all the modules in the Turbo C library file CS.LIB; OBJXREF can produce useful reports even when the files specified do not make a complete program. The /F causes all modules in CS.LIB file to be included in the report.

OBJXREF Error Messages and Warnings

OBJXREF generates two sorts of diagnostic messages, error messages and warnings.

Error Messages

Out of memory
OBJXREF performs its cross referencing in RAM memory and may run out of memory even if TLINK is able to link the same list of files successfully. When this happens, OBJXREF aborts. Remove memory resident programs to get more space or add more RAM memory.

Warnings

WARNING: Unable to open input file rrrr
The input file *rrrr* could not be located or opened. OBJXREF proceeds to the next file.

WARNING: Unknown option – oooo
The option name *oooo* is not recognized by OBJXREF. OBJXREF ignores the option.

WARNING: Unresolved symbol nnnn in module mmmm
The public name *nnnn* referenced in module *mmmm* is not defined in any of the .OBJ or .LIB files specified. OBJXREF flags the symbol in any reports it generates as being referenced but not defined.

WARNING: Invalid file specification ffff
Some part of the file name *ffff* is invalid. OBJXREF proceeds to the next file.

WARNING: No files matching ffff
The file named *ffff* listed on the command line or in a response file could not be located or opened. OBJXREF skips to the next file.

WARNING: Symbol nnnn defined in mmmm1 duplicated in mmmm2

Public name *nnnn* is defined in modules *mmmm1* and *mmmm2*. OBJXREF ignores the second definition.

A P P E N D I X

E

Language Syntax Summary

This appendix uses a modified Backus-Naur form to summarize the syntax for Turbo C constructs. These constructs are arranged categorically, as follows:

- *Lexical Grammar*: tokens, keywords, identifiers, constants, string literals, operators and punctuators
- *Phrase Structure Grammar*: expressions, declarations, statements, external definitions
- *Preprocessing Directives*

Optional elements in a construct are enclosed in <angle brackets>.

Lexical Grammar

Tokens

token:
  keyword
  identifier
  constant
  string-literal
  operator
  punctuator

Keywords

keyword: one of the following

asm	**do**	**goto**	**return**	**union**
auto	**double**	**huge**	**short**	**unsigned**
break	**else**	**if**	**signed**	**void**
case	**enum**	**int**	**sizeof**	**volatile**
cdecl	**extern**	**interrupt**	**static**	**while**
char	**far**	**long**	**struct**	**_cs**
const	**float**	**near**	**switch**	**_ds**
continue	**for**	**pascal**	**typedef**	**_es**
default		**register**		**_ss**

Identifiers

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of the following
a b c d e f g h i j k l m n o p q r s t u v w x y z _ $
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of the following
0 1 2 3 4 5 6 7 8 9

Constants

constant:
floating-constant
integer-constant
enumeration-constant
character-constant

floating-constant:
fractional-constant <exponent-part> <floating-suffix>
digit-sequence exponent-part <floating-suffix>

fractional-constant:
<digit-sequence> . *digit-sequence*
digit-sequence .

exponent-part:
e <*sign*> *digit-sequence*
E <*sign*> *digit-sequence*

sign: one of the following
\+ −

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of the following
f l F L

integer-constant:
decimal-constant <*integer-suffix*>
octal-constant <*integer-suffix*>
hexadecimal-constant <*integer-suffix*>

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
0
octal-constant octal-digit

hexadecimal-constant:
0 x *hexadecimal-digit*
0 X *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit: one of the following
1 2 3 4 5 6 7 8 9

octal-digit: one of the following
0 1 2 3 4 5 6 7

hexadecimal-digit: one of the following
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:
unsigned-suffix <*long-suffix*>
long-suffix <*unsigned-suffix*>

unsigned-suffix: one of the following
u U

long-suffix: one of the following
l L

enumeration-constant:
identifier

character-constant:
c-char-sequence

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any character in the source character set except
the single-quote ('), backslash (\), or newline character

escape-sequence

escape-sequence: one of the following

\'	\b	\v	\xhh
\"	\f	\o	\xhhh
\?	\n	\oo	\Xh
\\	\r	\ooo	\Xhh
\a	\t	\xh	\Xhhh

String Literals

string-literal:
" *<s-char-sequence>* "

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any character in the source character set except
the double-quote ("), backslash (\), or newline ( ) character

escape-sequence

Operators

operator: one of the following

```
[]        ()      .       ->      ++      --
&         *       +       -       ~       !
sizeof    /       %       .<<     >>      <
>         <=      >=      ==      !       =
^         |       &&      ||      ?:      =
*=        /=      %=      +=      -=      <<=
>>=       &=      ^=      |=      .       #
##
```

Punctuators

punctuator: one of the following

```
[ ]   ( )   { }   *   ,   :   =   ;   ...   #
```

Phrase Structure Grammar

Expressions

primary-expression:
  identifier
  constant
  pseudo-variable
  string-literal
  (*expression*)

pseudo-variable:

_AX **_AL** **_AH** **_SI** **_ES**
_BX **_BL** **_BH** **_DI** **_SS**
_CX **_CL** **_CH** **_BP** **_CS**
_DX **_DL** **_DH** **_SP** **_DS**

postfix-expression:
  primary-expression
  postfix-expression [*expression*]
  postfix-expression (<*argument-expression-list*>)
  postfix-expression . *identifier*

postfix-expression -> *identifier*
postfix-expression ++
postfix-expression --

argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*

unary-expression:
postfix-expression
++ *unary-expression*
- - *unary-expression*
unary-operator *cast-expression*
sizeof *unary-expression*
sizeof (*type-name*)

unary-operator: one of the following
& * + – ~ !

cast-expression:
unary-expression
(*type-name*) *cast-expression*

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression – *multiplicative-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

equality-expression:
relational-expression
equality expression == *relational-expression*
equality expression != *relational-expression*

AND-expression:
equality-expression
AND-expression & *equality-expression*

exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ *AND-expression*

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*

logical-OR-expression:
logical-AND-expression
logical-OR-expression || *logical-AND-expression*

conditional-expression:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*

assignment-expression:
conditional-expression
unary-expression *assignment-operator* *assignment-expression*

assignment-operator: one of the following

```
=      *=     /=     %=     +=     -=
<<=    >>=    &=     ^=     |=
```

expression:
assignment-expression
expression , *assignment-expression*

constant-expression:
conditional-expression

Declarations

declaration:
declaration-specifiers *<init-declarator-list>*

declaration-specifiers:
storage-class-specifier *<declaration-specifiers>*
type-specifier *<declaration-specifiers>*

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

init-declarator:
declarator
declarator = *initializer*

storage-class-specifier:
typedef
extern
static
auto
register

type-specifier:
void
char
short
int
long
float
double
signed
unsigned
const
volatile
struct-or-union-specifier
enum-specifier
typedef-name

struct-or-union-specifier:
struct-or-union *<identifier>* { *struct-declaration-list* }
struct-or-union *identifier*

struct-or-union:

struct
union

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
type-specifier-list struct-declarator-list;

type-specifier-list:
type-specifier
type-specifier-list type-specifier

struct-declarator-list:
struct-declarator
struct-declarator-list , *struct-declarator*

struct-declarator:
declarator
<declarator> : *constant-expression*

enum-specifier:
enum *<identifier>* { *enumerator-list* }
enum *identifier*

enumerator-list:
enumerator
enumerator-list , *enumerator*

enumerator:
enumeration-constant
enumeration-constant = *constant-expression*

declarator:
<pointer> direct-declarator
<modifier-list>

direct-declarator:
identifier
(*declarator*)
direct-declarator [*<constant-expression>*]
direct-declarator (*parameter-type-list*)
direct-declarator (*<identifier-list>*)

pointer:
* *<type-specifier-list>*
* *<type-specifier-list> pointer*

modifier-list:
  modifier
  modifier-list modifier

modifier:
  cdecl
  pascal
  interrupt
  near
  far
  huge

parameter-type-list:
  parameter-list
  parameter-list , ...

parameter-list:
  parameter-declaration
  parameter-list , *parameter-declaration*

parameter-declaration:
  declaration-specifiers declarator
  declaration-specifiers <abstract-declarator>

identifier-list:
  identifier
  identifier-list , *identifier*

type-name:
  type-specified-list <abstract-declarator>

abstract-declarator:
  pointer
  <pointer> <direct-abstract-declarator>
  <modifier-list>

direct-abstract-declarator:
  (*abstract-declarator*)
  <direct-abstract-declarator> [*<constant-expression>*]
  <direct-abstract-declarator> (*<parameter-type-list>*)

typedef-name:
  identifier

initializer:
  assignment-expression
  { *initializer-list* }
  { *initializer-list* , }

initializer-list:
initializer
initializer-list , *initializer*

Statements

statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement
asm-statement

asm-statement
asm *tokens newline*
asm *tokens;*

labeled-statement:
identifier : *statement*
case *constant-expression* : *statement*
default : *statement*

compound-statement:
{ *<declaration-list>* *<statement-list>* }

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

expression-statement:
<expression> ;

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*

iteration-statement:
while (*expression*) *statement*
do *statement* **while** (*expression*) ;

for (*<expression>* ; *<expression>* ; *<expression>*) statement

jump-statement
goto *identifier* ;
continue ;
break ;
return *<expression>*;

External Definitions

file:
external-definition
file external-definition

external-definition:
function-definition
declaration

asm-statement
asm *tokens newline*
asm *tokens;*

function-definition:
<declaration-specifiers> declarator <declaration-list> compound-statement

Preprocessing Directives

preprocessing-file:
group

group:
group-part
group group-part

group-part:
<pp-tokens> newline
if-section
control-line

if-section:
if-group <elif-groups> <else-group> endif-line

if-group:
`#if` *constant-expression newline <group>*

`#ifdef` *identifier newline <group>*
`#ifndef` *identifier newline <group>*

elif-groups:
elif-group
elif-groups elif-group

elif-group:
`#elif` *constant-expression newline <group>*

else-group:
`#else` *newline <group>*

endif-line:
`#endif` *newline*

control-line:
`#include` *pp-tokens newline*
`#define` *identifier replacement-list newline*
`#define` *identifier lparen <identifier-list>*) *replacement-list newline*
`#undef` *identifier newline*
`#line` *pp-tokens newline*
`#error` *<pp-tokens> newline*
`#pragma` *<pp-tokens> newline*
`#pragma` warn *action abbreviation newline*
`#pragma` inline *newline*
`#` *newline*

action:
+
–
.

abbreviation:

```
amb   def   rch   stu
amp   dup   ret   stv
apt   eff   rng   sus
aus   mod   rpt   ucp
big   par   rvl   use
cln   pia   sig   voi
cpt   pro   str   zst
```

lparen:
the left-parenthesis character without preceding white space

replacement-list:
<*pp-tokens*>

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

preprocessing-token:
header-name (only within an `#include` directive)
identifier (no *keyword* distinction)
constant
string-literal
operator
punctuator
each non-whitespace character that cannot be one of the preceding

header-name:
<*h-char-sequence*>

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any character in the source character set except the newline
greater than (>) character

newline:
the newline character

A P P E N D I X

F

TCINST: Customizing Turbo C

TCINST is the Turbo C customization program; you use it to customize TC.EXE, the integrated development environment version of Turbo C. Through TCINST, you can change various default settings in the TC operating environment, such as the screen size, editing modes, menu colors, and default directories. TCINST lets you change the environment in which you operate Turbo C: It directly modifies certain default values within your copy of TC.EXE.

With TCINST, you can do any of the following:

- specify default primary file and project names
- set up paths to the directories where your include, library, configuration, Help, pick, and output files are located
- choose default settings for the integrated debugger
- customize the editor command keys
- set up Turbo C's editor defaults and on-screen appearance
- set up the default video display mode
- change screen colors
- resize Turbo C's Edit and Message windows

Turbo C comes ready to run: You do not need to run TCINST if you don't want to. You can install the files from the distribution disks onto your working floppies or hard disk, as described in Chapter 1 of the *Turbo C User's Guide*, then run Turbo C. However, if you do want to change the defaults already set in TC.EXE, TCINST provides you with a handy means

of doing it. All you have to do is run TCINST, then choose items from the TCINST menu system.

Note: These menus are very similar to the menus in the TC integrated development environment. For detailed information on the features refer to Chapter 5 in the *Turbo C User's Guide*, which discusses the TC menu system in depth.

Note: Any option that you install with TCINST that *also* appears as a menu option in TC.EXE will be overridden whenever you load a configuration file that contains a different setting for that option, or when you change the setting via the menu system of the integrated development environment.

Running TCINST

The syntax for TCINST is

```
tcinst [option] [pathname]
```

Both *pathname* and `option` are optional. If *pathname* is not supplied, TCINST looks for TC.EXE in the current directory. Otherwise, it uses the given path name.

[`option`] lets you specify whether you want to run TCINST in color (type in `/c`) or in black and white (type in `/b`). Normally, TCINST comes up in color if it detects a color adapter in a color mode. You can override this default if, for instance, you are using a composite monitor with a color adapter, by using the `/b` option.

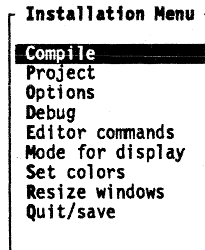
Note: You can use one version of TCINST to customize several different copies of Turbo C on your system. These various copies of TC.EXE can have different executable program names; all you need to do is invoke TCINST and give a path name to the copy of TC.EXE you're customizing; for example,

```
tcinst tc.exe
tcinst ..\..\bwtc.exe
tcinst /c c:\borland\colortc.exe
```

In this way, you can customize the different copies of Turbo C on your system to use different editor command keys, different menu colors, and so on.

The TCINST Installation Menu

The first menu to appear on the screen is the TCINST installation menu.



Turbo C Installation Program 2.0

Figure F.1: The TCINST Installation Menu

- The **C**ompile option lets you specify a default name for the primary C file to be compiled.
  Choosing **P**roject lets you assign a default name for your project file, and also to set various defaults for compiling your project.
- The **O**ptions command gives you access to default settings for a great many features, including memory model, degree of optimization, display of error messages, linker and environment settings, and path names to the directories holding header and library files.
- **D**ebug lets you set the **S**ource Debugging and **D**isplay Swapping defaults for the integrated debugger.
- You can use the **E**ditor commands option to reconfigure (customize) the interactive editor's keystroke commands.
- With **M**ode for Display, you can specify the video display mode that TC will operate in, and whether yours is a "snowy" video adapter.
- You can customize the colors of almost every part of TC's integrated environment through the **S**et Colors menu.

- The **R**esize Windows option allows you to change the sizes of the Edit and Message/Watch windows.
- The **Q**uit/Save option lets you save the changes you have made to the integrated development environment, and returns you to system level.

To choose a menu item, just press the key for the highlighted capital letter of the given option. For instance, press *S* to choose the Set Colors option. Or use the *Up* and *Down* arrow keys to move the highlight bar to your choice, then press *Enter*.

Pressing *Esc* (more than once if necessary) returns you from a submenu to the main installation menu.

The Compile Menu

The Compile menu contains only one option, **P**rimary File. If you choose it, a prompt box appears in which you can type the default name for the source file that is to be compiled and linked in the event that you are doing a one-file program that includes multiple header files. This option is useful if you will frequently be compiling one particular primary file.

The Project Menu

The choices in the **P**roject menu allow you to set a default name for your project file, and to specify default settings for features responsible for compiling and linking your project.

Project Name

When you choose this option, a prompt box appears in which you type the default name for your project file. The .PRJ extension will be supplied automatically.

The Break Make On Menu

This menu lets you specify the default condition for stopping a make: if the file has **W**arnings, **E**rrors, or **F**atal errors, or before **L**inking.

Auto Dependencies

This option lets you set the default for the **Auto** Dependencies toggle to On or Off.

Clear Project

This option cancels a previously set project name, so that, for example, you can specify a different one.

The Options Menu

In the **Options** menu you can set defaults for various features that determine how the integrated environment works.

The Compiler Menu

The options in the **Compiler** menu allow you to set defaults for particular hardware configurations, memory models, code optimizations, diagnostic message control, and macro definitions.

Model

The choices let you choose the default memory model (method of memory addressing) that TC will use. The options are **Tiny**, **Small**, **Compact**, **Medium**, **Large**, and **Huge**. Refer to Chapter 12 in the *Turbo C User's Guide* for more information about these memory models.

Defines

When you choose this option, a prompt box appears in which you can enter macro definitions that will be available by default to TC.

The Code Generation Menu

The items in this menu let you set defaults for how the compiler will compile your source code.

Calling Convention: Set to either C or Pascal calling sequence.

Instruction Set: Set to either 8088/8086 or 80186/80286.

Floating Point: Set to 8087/80287, Emulation, or None.

Default Char Type: Set to Signed or Unsigned.

Alignment: Set to word-aligning or byte-aligning.

Generate Underbars: Set On or Off.

Merge Duplicate Strings: Set On or Off.

Standard Stack Frame: Set On or Off. **Note:** If you are going to be running your program with the integrated debugger, this option should be turned On.

Test Stack Overflow: Set On or Off.

Line Numbers: Set On or Off.

OBJ Debug Information: Set On or Off

The Optimization Menu

The options in this menu let you set defaults for code optimization when your code is compiled.

Optimize For: Set to Size or Speed.

Use Register Variables: Set On or Off.

Register Optimization: Set On or Off.

Jump Optimization: Set On or Off. **Note:** If you are going to be running your program with the integrated debugger, this option should be turned Off.

The Source Menu

With this menu you can set defaults for identifier length in characters, whether or not TC will recognize nested comments, and whether TC will recognize extension keywords, or ANSI keywords only.

The Errors Menu

Use this menu to

- set the default number of errors or warnings after which your code will stop compiling (0 to 255).

- turn On or Off the display for chosen warning messages.
- choose error/warning messages to be displayed (toggle them On/Off). These are of four types, each with its own menu:
  - Portability Warnings
  - ANSI Violations
  - Common Errors
  - Less Common Errors

The Names Menu

With the items in this menu, you can set the default segment, group, and class names for Code, Data, and BBS sections. When you choose one of these items, the asterisk (*) on the next menu that appears tells the compiler to use the default names.

Don't change this option unless you are an expert and have read Chapter 12 in the Turbo C User's Guide *on advanced programming techniques.*

The Linker Menu

The Linker menu lets you set defaults for how your program will be linked to various library routines. Refer to Appendix D for more information about these settings.

Map File

This option determines the default type for the map file. The choices are Off, Segments, Publics, or Detailed.

Initialize Segments

Set On or Off. If this toggle is set to On, the linker will initialize uninitialized segments.

Default Libraries

Set On or Off. When you're linking with modules that have been created by a compiler other than Turbo C, the other compiler may have placed a list of default libraries in the object file. If this option is on, the linker will try to find any undefined routines in these libraries, as well as in the default

libraries supplied by Turbo C. If this option is set to Off, only the default libraries supplied by Turbo C will be searched; any defaults in .OBJ files will be ignored.

Graphics Library

Controls whether the linker links in BGI graphics library functions. Defaults to On; set it Off to prevent the linker from searching GRAPHICS.LIB.

Warn Duplicate Symbols

Sets On or Off the linker warning for duplicate symbols in object and library files.

Stack Warning

Sets On or Off the `No stack specified` message generated by the linker.

Case-Sensitive Link

Sets On or Off case sensitivity during linking. The usual setting is On, since C is a case-sensitive language.

The Environment Menu

With the items in the Environment menu, you can set defaults for various features of the TC working environment.

Look at the Quick-Ref line for directions on how to choose these options. You can change the operating environment defaults to suit your preferences (and your monitor), then save them as part of Turbo C. Of course, you'll still be able to change these settings from inside Turbo C's editor (or from the Options/Environment menu).

Message Tracking

This option determines the range of syntax error tracking available to you after your program has compiled. Set it to either Current file, All files, or Off.

Keep Messages

Set On or Off. This option determines whether error messages from earlier compiles are saved in the Message window or deleted.

Config Auto Save

Set On or Off. When this option is set to On, TC automatically saves the current configuration to the configuration file whenever you run your program, shell to DOS, or exit the integrated environment, if you haven't loaded, retrieved, or saved a configuration file.

Edit Auto Save

Set On or Off. If On, this feature automatically saves your source file whenever you run your program or shell to DOS, if you have modified it since the last save.

Backup Source Files

Set On or Off. If you choose On, Turbo C will automatically create a backup of your source file whenever you do a save.

Zoomed Windows

Set On or Off. When this option is set to On, the active window (Edit or Message/Watch) is zoomed on startup to fill the whole screen; when it is set to Off, both windows are visible by default.

Full Graphics Save

In order to save graphics screens, TC reserves 8K of memory as a buffer for the palettes. If you are going to be using only text screens, you can make this memory available to TC by turning Full Graphics Save to Off. This option is available only through TCINST, not through the integrated development environment, since the buffer must be reserved when TC loads.

The Screen Size Menu

The Screen Size menu allows you to specify whether your default integrated environment screen will display 25 lines or 43/50 lines.

- **25 Lines**

  This is the standard PC display: 25 lines by 80 columns. This is the only screen size available to systems with a Monochrome Display Adapter (MDA) or Color Graphics Adapter (CGA).

- **43/50 Lines**

  If your PC is equipped with an EGA or VGA, choose 43/50 lines to make your screen display 43 lines by 80 columns (for an EGA) or 50 lines by 80 columns (for a VGA).

The Options for Editor Menu

This menu lets you set defaults for various features of the integrated development environment's Editor.

Insert Mode: Toggle On or Off. With Insert Mode set to On, the editor inserts anything you enter from the keyboard at the cursor position, and pushes existing text to the right of the cursor even further right. Toggling Insert Mode Off allows you to overwrite text at the cursor.

Autoindent Mode: Toggle On or Off. With Autoindent Mode set to On, the cursor returns to the starting column of the previous line when you press *Enter*. When Autoindent Mode is toggled Off, the cursor always returns to column one.

Use Tabs: Toggle On or Off. With Use Tabs set to On, when you press the *Tab* key, the editor places a tab character (*Ctrl-I*) in the text, using the tab size specified with Tab Size. With Use Tabs Off, when you press the *Tab* key, the editor inserts enough space characters to align the cursor with the first letter of each word in the previous line.

Optimal Fill: Toggle On or Off. Optimal fill mode has no effect unless Tab Mode is also set to On. When both these modes are enabled, the beginning of every autoindented and unindented line is filled optimally with tabs and spaces. This produces lines with a minimum number of characters.

Backspace Unindents: Toggle On or Off. When it is set to On, this feature *outdents* the cursor; that is, it aligns the cursor with the first nonblank character in the first outdented line above the current or immediately preceding nonblank line.

Tab Size: When you choose this option, a prompt box appears in which you can enter the number of spaces you want to tab over at each tab command.

Editor Buffer Size: If you normally write programs using small files, you can free extra memory for debugging by using a smaller editor buffer. You can size the editor buffer to any size between 20000 and 65534 bytes.

Make Use of EMS Memory: If your machine is equipped with 64K of EMS memory, the Editor will automatically use it for its text buffer. This will free 64K of RAM for compiling, linking, and running your programs. Default is On; turn this toggle to Off to prevent the Editor from using EMS memory.

The Directories Menu

With Directories, you can specify a path to each of the TC.EXE default directories. These are the directories Turbo C searches when looking for an alternate configuration file, the Help file, the include and library files, and the directory where it will place your program output.

When you choose Turbo C Directories, TCINST brings up a submenu. The items on this submenu are

- Include Directories
- Library Directories
- Output Directory
- Turbo C Directory
- Pick File Name

You enter names for each of these just as you do for the corresponding menu items in TC.EXE. If you are not certain of each item's syntax, refer first to Chapter 5 in the *Turbo C User's Guide*.

After typing a path name (or names) for any of the Directories menu items, press *Enter* to accept. When you exit the program, TCINST prompts you on whether you want to save the changes. Once you save the Turbo C Directories paths, the locations are written to disk and become part of TC.EXE's default settings.

Include Directories

This option lets you specify default directories in which the Turbo C standard include (header) files are stored. A prompt box appears in which you can enter the directory names.

You can enter multiple directories in Include Directories. You must separate the directory path names with a semicolon (;), and you can enter a maxi-

mum of 127 characters with either menu item. You can enter absolute or relative path names.

An Example:

```
c:\turboc\lib; c:\turboc\mylibs; a:newturbo\mathlibs; a:..\vidlibs
```

Library Directories

Use the Library Directories option to specify default directories for the Turbo C start-up object files (C0x.OBJ) and run-time library files (.LIB). A prompt box appears in which you can enter the directory names.

As with Include Directories, you can enter multiple directories in Library Directories. You must separate the directory path names with a semicolon (;), and you can enter a maximum of 127 characters with either menu item. You can enter absolute or relative path names.

Output Directory

Use this option to name the default directory where the compiler will store the .OBJ, .EXE, and .MAP files it creates.

The Output Directory menu item takes one directory path name; it accepts a maximum of 64 characters.

Turbo C Directory

This option lets you specify the name of the directory where TC looks for the Help file and TCCONFIG.TC (the default configuration file) if they aren't in the current directory.

The Turbo C Directory menu items each take one directory path name; each item accepts a maximum of 64 characters.

Pick File Name

When you choose this menu item, a prompt box pops up. In it, you type the path name of the pick file you want Turbo C to load or create.

Arguments

This setting allows you to set default command-line arguments that will be passed to your running programs, exactly as if you had typed them on the DOS command line (redirection is not supported). It is only necessary to give the arguments here; the program name is omitted.

The Debug Menu

The items in the Debug menu let you set certain default settings for the Turbo C integrated debugger.

Source Debugging

Selects debugging. When you compile your program with this toggle On, you can debug it using either the integrated debugger or the standalone debugger. When it is set to Standalone, only the standalone debugger can be used. When it is set to None, no debugging information is placed in the .EXE file.

Display Swapping

This option allows you to set the default level of Display Swapping to Smart, Always, or None.

When you run your program in debug mode with the default setting Smart, the Debugger looks at the code being executed to see whether the code will affect the screen (that is, output to the screen). If the code outputs to the screen (or if it calls a function), the screen is swapped from the Editor screen to the Execution screen long enough for output to take place, then is swapped back. Otherwise, no swapping occurs. The Always setting causes the screen to be swapped every time a statement executes. The None setting causes the debugger not to swap the screen at all.

The Editor Commands Option

Turbo C's interactive editor provides many editing functions, including commands for

- cursor movement

- text insertion and deletion
- block and file manipulation
- string search (plus search-and-replace)

These editing commands are assigned to certain keys (or key combinations): They are explained in detail in Appendix A of this volume.

When you choose Editor commands from TCINST's main installation menu, the Install Editor screen comes up, displaying three columns of text.

- The first column (on the left) describes all the functions available in TC's interactive editor.
- The second column lists the *Primary* keystrokes: what keys or special key combinations you press to invoke a particular editor command.
- The third column lists the *Secondary* keystrokes: These are optional alternate keystrokes you can also press to invoke the same editor command.

Note: Secondary keystrokes always take precedence over primary keystrokes.

The bottom lines of text in the Install Editor screen summarize the keys you use to choose entries in the Primary and Secondary columns.

Key	Legend	What It Does
Left, Right *Up* and *Down* arrow keys	choose	Chooses the editor command you want to rekey.
Page Up and *Page Down* arrow keys	page	Scrolls up or down one full screen page.
Enter	modify	Enters the keystroke-modifying mode.
R	restore factory defaults	Resets all editor commands to the factory default keystrokes.
Esc	exit	Leaves the Install Editor screen and returns to the main TCINST installation menu.
F4	Key Modes	Toggles between the three keystroke combinations: WordStar-like, Ignore case, and Verbatim.

After you press *Enter* to enter the modify mode, a pop-up window lists the current defined keystrokes for the chosen command, and the bottom lines of text in the Install Editor screen summarize the keys you use to change those keystrokes.

Key	Legend	What It Does
Backspace	backspace	Deletes keystroke to left of cursor.
Enter	accept	Accepts newly defined keystrokes for the chosen editor command.
Esc	abandon changes	Abandons changes to the current choice, restoring the command's original keystrokes, and returns to the Install Editor screen (ready to choose another editor command).
F2	restore	Abandons changes to current choice, restoring the command's original keystrokes, but keeps the current command chosen for redefinition.
F3	clear	Clears current choice's keystroke definition, but keeps the current command chosen for re-definition.
F4	Key Modes	Toggles between the three keystroke combinations: WordStar-like, Ignore case, and Verbatim.

Note: To enter the keys *F2, F3,* or *F4* as part of an editor command key sequence, first press the backquote (‘) key, then the appropriate function key.

Keystroke combinations come in three flavors: WordStar-like, Ignore case, and Verbatim. These are listed on the bottom line of the screen; the highlighted one is the flavor of the current choice. In all cases, the first character of the combination must be a special key or a control character. The combination flavor governs how the subsequent characters are handled.

- WordStar-like: In this mode, if you type a letter or one of the following characters:

  [ ] \ ^ –

  ..it is automatically entered as a control-character combination. For example,

  Typing *a* or *A* or *Ctrl A*   yields *<Ctrl-A>*

Typing *y* or *Y* or *Ctrl Y* yields *<Ctrl-Y>*

Typing *[* yields *<Ctrl-[>*

Thus, if you customize an editor command to be *< Ctrl A > < Ctrl B >* in WordStar-like mode, you can type any of the following in the TC editor to activate that command:

< Ctrl A > < Ctrl B >
< Ctrl A > B
< Ctrl A > b

- Ignore case: In this mode, all alpha (letter) keys you enter are converted to their uppercase equivalents. When you type a letter in this mode, it is *not* automatically entered as a control-character combination; if a keystroke is to be a control-letter combination, you must hold down the *Ctrl* key while typing the letter. For example, in this mode, *<Ctrl-A> B* and *<Ctrl-A> b* are the same, but differ from *<Ctrl A> <Ctrl B>*.
- Verbatim: If you type a letter in this mode, it is entered exactly as you type it. So, for example, *<Ctrl A> <Ctrl B>*, *<Ctrl A> B* , and *<Ctrl A> b* are all distinct.

Allowed Keystrokes

Although TCINST provides you with almost boundless flexibility in customizing the Turbo C editor commands to your own tastes, there are a few rules governing the keystroke sequences you can define. Some of the rules apply to any keystroke definition, while others come into effect only in certain keystroke modes.

1. You can enter a maximum of six keystrokes for any given editor command. Certain key combinations are equivalent to two keystrokes: These include *Alt (any valid key)*; the cursor-movement keys (*Up, Page Down, Del*, and so on); and all function keys or function key combinations (*F4, Shift-F7, Alt-F8*, and so on.
2. The first keystroke must be a character that is non-alphanumeric and non-punctuation: that is, it must be a control key or a special key.
3. To enter the *Esc* key as a command keystroke, type *Ctrl [*.
4. To enter the *Backspace* key as a command keystroke, type *Ctrl H*.
5. To enter the *Enter* key as a command keystroke, type *Ctrl M*.
6. The Turbo C predefined Help function keys (*F1* and *Alt-F1*) can't be reassigned as Turbo C editor command keys. Any other function key can, however. If you enter a Turbo C hot key as part of an editor

command key sequence, TCINST will issue a warning that you are overriding a hot key in the editor and verify that you want to override that key. Chapter 5 of the *Turbo C User's Guide* contains a complete list of Turbo C's predefined hot keys.

The Mode for Display Menu

Normally, Turbo C correctly detects your system's video mode. You should only change the **M**ode for Display menu if one of the following holds true:

- You want to choose a mode other than the current video mode.
- You have a Color Graphics Adapter that doesn't "snow."
- You think Turbo C is incorrectly detecting your hardware.
- You have a laptop or a system with a composite screen (which acts like a CGA with only one color). For this situation, choose **B**lack and White.

Press *M* to choose Mode for Display from the installation menu. A pop-up menu appears; from this menu, you can choose the screen mode Turbo C will use during operation. Your options include **D**efault, **C**olor, **B**lack and White, or **M**onochrome. These are fairly intuitive.

Default By default, Turbo C always operates in the mode that is active when you load it.

Color Turbo C uses 80-column color mode if a color adapter is detected, no matter what mode is active when you load TC.EXE, and switches back to the previously active mode when you exit.

Black and White Turbo C uses 80-column black-and-white mode if a color adapter is detected, no matter what mode is active, and switches back to the previously active mode when you exit. Use this with laptops and composite monitors.

Monochrome Turbo C uses monochrome mode if a monochrome adapter is detected, no matter what mode is active.

When you choose one of the first three options, the program conducts a video test on your screen; refer to the Quick-Ref line for instructions on what to do. When you press any key,
a window comes up with the query

```
Was there Snow on the screen?
```

You can choose

- Yes, the screen was "snowy"
- No, always turn off snow checking
- Maybe, always check the hardware

Look at the Quick-Ref line for more about **Maybe**. Press *Esc* to return to the main installation menu.

The Set Colors Menu

Pressing *S* from the main installation menu allows you to make extensive changes to the colors of your version of Turbo C. After you press *S*, a menu with these options appears:

- Customize colors
- Default color set
- Turquoise color set
- Magenta color set

Because there are nearly 50 different screen items that you can color-customize, you will probably find it easier to choose a *preset* set of colors to your liking.

There are three preset color sets to choose from. Press *D*, *T*, or *M*, and scroll through the colors for the Turbo C screen items using the *PgUp* and *PgDn* keys. If none of the preset color sets is to your liking, you can design your own.

To make custom colors, press *C* for **Customize colors**. Now you have a choice of 12 types of items that can be color-customized in Turbo C; some of these are text items, some are screen lines and boxes. Choose one of these items by pressing a letter *A* through *L*.

Once you choose a screen item to color-customize, you will see a pop-up menu and a *viewport*. The viewport is an example of the screen item you chose, while the pop-up menu displays the components of that choice. The viewport also reflects the change in colors as you scroll through the color palette.

For example, if you choose *H* to customize the colors of Turbo C's error boxes, you'll see a new pop-up menu with the four different parts of an error box: its **Title**, **Border**, **Normal Text**, and **Highlighted Text**.

You can now choose one of the components from the pop-up menu. Type the appropriate highlighted letter, and you're treated to a color palette for the item you chose. Using the arrow keys, choose a color to your liking from the palette. Watch the viewport to see how the item looks in that color. Press *Enter* to record your choice.

Repeat this procedure for every screen item you want to color-customize. When you are finished, press *Esc* until you are back at the main installation menu.

Note: Turbo C maintains three internal color tables: one each for color, black and white, and monochrome. TCINST allows you to change only one of these three sets of colors at a time, based upon your current video mode. For example, if you want to change to the black-and-white color table, you set your **M**ode for Display to **B**lack and White, and then set the attributes for black-and-white mode.

Resize Windows

This option allows you to set the maximum size of Turbo C's Message/ Watch window. Press *R* to choose **R**esize Windows from the main installation menu.

Using the *Up arrow* and *Down arrow* keys, you can move the bar dividing the Edit window from the Message/Watch window. Neither window can be smaller than one line. When you have resized the window to your liking, press *Enter*. The dividing bar operates as a ratio of how large the Edit window will be in relation to the Message/Watch window. This applies whether the line mode is 25 lines or 43/50 lines.

You can discard your changes and return to the Installation menu by pressing *Esc*.

Quitting the Program

Once you have made all desired changes, choose **Q**uit/Save at the main installation menu. The message

```
Save changes to TC.EXE? (Y/N)
```

appears at the bottom of the screen.

- If you press *Y* (for Yes), all the changes you have made are permanently installed into Turbo C. (You can always run TCINST again if you want to change them.)
- If you press *N* (for No), your changes are ignored and you are returned to the operating system prompt without Turbo C's defaults or startup appearance being changed.

If you decide you want to restore the original Turbo C factory defaults, simply copy TC.EXE from your master disk onto your work disk. You can also restore the **E**ditor commands by choosing the *E* option at the TCINST main menu, then press *R* (for **R**estore Factory Defaults) and *Esc*.

A P P E N D I X

G

MicroCalc

MicroCalc, written in Turbo C, is a spreadsheet program. Its source code files and an object file are provided with your TURBO C system as an example program. The spreadsheet program is an electronic piece of paper on which you can enter text, numbers and formulas, and have MicroCalc do calculations on them automatically.

About MicroCalc

Since MicroCalc is only a demonstration program, it has its limitations (which you may have fun eliminating):

- You cannot copy formulas from one cell to another.
- You cannot copy text or values from one cell to another.
- Cells that are summed must be in the same column or row.

In spite of its limitations, MicroCalc does provide some interesting features. Among these are the following:

- writing directly to video memory for maximum display speed
- full set of mathematical functions
- built-in line editor for text and formula editing
- ability to enter text across cells

In addition to these, MicroCalc offers many of the usual features of a spreadsheet program; you can do all of the following:

- Load a spreadsheet from the disk.
- Save a spreadsheet on the disk.
- Automatically recalculate after each entry (can be disabled).
- Print the spreadsheet on the printer.
- Clear the current spreadsheet.
- Delete columns and rows.
- Set a column's width.
- Insert blank columns and rows between existing ones.

How to Compile and Run MicroCalc

Compiling MicroCalc is easy. All you need to do is copy all the MC*.* files from your distribution disk to your TURBOC directory (where TC.EXE and/or TCC.EXE reside). You can compile and run MicroCalc with either version of Turbo C. In both cases, compiling under a large data model (compact, large, or huge) will give you much more memory for your spreadsheets.

With TC.EXE

After you have set the INCLUDE and LIB directories in the **O**ptions/**D**irectories menu, do the following:

1. Run TC.EXE.
2. In the **P**roject menu, specify the project name "MCALC.PRJ."
3. From the **R**un menu, choose the **R**un option.

With TCC.EXE

Compile from DOS with the following command line:

```
TCC mcalc mcparser mcdisply mcinput mcommand mcutil
```

Note: You must also specify the INCLUDE and LIB directories with the `-I` and `-L` command-line options, respectively.

How to use MicroCalc

Once you have compiled MicroCalc, you can run it in one of two ways.

If you compiled with the **Run/Run** command from TC, MicroCalc will come up on your screen; when you exit, you will return to Turbo C.

If you want to run MCALC.EXE from the DOS command line, just type MCALC. (If you already have a spreadsheet file, you can automatically load it by typing

```
MCALC <your_file>
```

at the DOS prompt.)

This is an example of what you will see once MicroCalc is loaded:

```
          A        B        C        D        E        F        G
1       22.00
2        1.00
3        2.00
4        3.00
5       28.00

...

20
A5 Formula
A1+A2+A3+A4
```

The MicroCalc screen is divided into cells. A cell is a space on the spreadsheet where a column and row intersect. The column name and the row number are the *cell coordinates*. By default, each column is 10 characters wide; you can change this to a maximum of 77 characters (each).

The columns are named A-Z and AA-CV; the rows are numbered 1-100. This gives a total of 10000 cells. You can change these limits by modifying the constants MAXROWS and MAXCOLS in the header file MCALC.H.

A cell may contain a value, a formula or some text; these are known as cell types. The type of the cell and its coordinates are shown in the bottom left corner of the screen:

`A5 Formula`	Cell A5 contains a formula.
`A1 Text`	Cell A1 contains text.
`A2 Value`	Cell A2 contains a value and no cell references.

In this example, the line `A5 Formula` shows that the active cell is cell A5 and that it contains a formula. The last line, `A1+A2+A3+A4`, says the active cell

contains the sum of A1 through A4. These two lines mean that the numbers in cells A1, A2, A3 and A4 should be added and the result placed in cell A5.

The formula can be abbreviated to `A1:A4`, meaning "add all cells from A1 to A4."

The following are examples of valid cell formulas:

`A1+(B2-C7)`	Subtract cell C7 from B2 and add the result to cell A1
`A1:A23`	The sum of cells: A1,A2,A3..A23

The formulas may be as complicated as you want; for example,

```
SIN(A1)*COS(A2)/((1.2*A8)+LOG(ABS(A8)+8.9E-3))+(C1:C5)
```

To enter data in any cell, move the cursor to that cell and enter the data. MicroCalc automatically determines if the cell's type is value, formula, or text.

Standard MicroCalc Functions and Operators

+, –, *, /	addition, subtraction, multiplication, division
^ –	raises a number to a power (e.g., `2^3 = 8`)
:	returns the sum of a group of cells (for example, `A1:A4 = A1+A2+A3+A4`)
ABS	absolute value
ACOS	arc cosine
ASIN	arc sine
ATAN	arc tangent
COS	cosine
COSH	hyperbolic cosine
EXP	exponential function
LOG	logarithm
LOG10	base 10 logarithm
POW10	raise argument to the 10th power
ROUND	round to the nearest whole number
SIN	sine
SINH	hyperbolic sine
SQR	square
SQRT	square root
TAN	tangent
TANH	hyperbolic tangent
TRUNC	return the whole part of a number

Standard MicroCalc Commands

/	Brings up the main menu
/SL	Loads a spreadsheet
/SS	Saves the current spreadsheet
/SP	Prints the current spreadsheet
/SC	Clears the current spreadsheet
/F	Formats a group of cells
/D	Deletes the current cell
/G	Moves the cursor to a selected cell
/CI	Inserts a column
/CD	Deletes the current column
/CW	Changes the width of the current column
/RI	Inserts a row
/RD	Deletes the current row
/E	Edits the current cell
/UR	Recalculates the formulas in the spreadsheet
/UF	Toggles the display of the text of formulas in cells instead of the value of the formulas
/A	Toggles AutoCalc on/off
/Q	Quits from MicroCalc
Del	Deletes the current cell
Home	Moves to cell A1
End	Moves to the rightmost column and bottom row of the spreadsheet
PgUp, PgDn	Moves up or down a full screen
F2	Allows you to edit the data in the current cell.

While you're editing, the following commands work:

Esc	Disregards changes made to the data.
Left arrow, Right arrow	Moves to the left and right.
Up arrow, Down arrow, Enter	Enters the input, then returns to the current cell.
Home	Moves to the start of the input.
End	Moves to the end of the input.
Del	Deletes the character under the cursor.
Ins	Changes between Insert/Overwrite mode.

Backspace	Deletes the character to the left of the cursor.

The MicroCalc Parser

This information is provided in case you want to modify the MicroCalc parser (for instance, you might want to add a function that takes two parameters). The state and "goto" information for the parser was created using the UNIX YACC utility. The input to YACC was as follows:

```
%token CONST CELL FUNC
%%
e : e '+' t
                          | e '-' t
                          | t
                          ;
t : t '*' f
                          | t '/' f
                          | f
                          ;
f : x '^' f
                          | x
                          ;
x : '-' u
                          | u
                          ;
u : CELL ':' CELL
                          | o
                          ;
o : CELL
                          | '(' e ')'
                          | CONST
                          | FUNC '(' e ')'
                          ;
%%
```

Index

8087/80287 coprocessor
exception handler 75
floating-point problems with 137
status word 75
8087/80287 exception handler 361
8087/80287 status word 361
_*8087* (global variable) 23
80186 instructions, extended 448
43/50 line screen setting 564
8087 coprocessor
calls, emulation of 448
floating-point emulation library routines 448
instructions, inline 448
8086 interrupt vectors 184, 338
__emit__ (function) 103
25 line screen setting 564
80x86 processors 105
_*argc* (global variable) 22
_*argv* (global variable) 22
.ASM files 455
_chmod (function) 71
_clear87 (function) 75
_close (function) 78
_control87 (function) 80
_creat (function) 86
#defines command-line options 446, 447
ganging 447
_*doserrno* (global variable) 24
.EXE file
user-selected name for 455
_exit (function) 109
_*fmode* (global variable) 27
_fpreset (function) 136
_graphfreemem (function) 192
_graphgetmem (function) 193
_*heaplen* (global variable) 28
_lrotl (function) 235
_lrotr (function) 235
_matherr (function) 241
_open (function) 256
_*osmajor* (global variable) 29
_*osminor* (global variable) 29
_*psp* (global variable) 29
_read (function) 290
_rotl (function) 298
_rotr (function) 299
_status87 (function) 361
_*stklen* (global variable) 30
_strerror (function) 365
_tolower (function) 390
_toupper (function) 391
_*version* (global variable) 31
_write (function) 406

A

abort (function) 35
abort operation command (TC editor) 417
abs (function) 35
absolute disk sectors 36, 37
absolute value
complex number 67
floating-point number 111
integer 35
long integer 226
absread (function) 36
abswrite (function) 37
access
flags 349
mode 359
changing 71, 72
read/write 38, 72, 87, 89, 147, 257, 258, 360
access (function) 37
acos (function) 39
action symbols, TLIB 511
active page 312
active window 409
address, mailing, Borland 4
address segment, of far pointer 138, 251
addresses
passed to __emit__ 104
Alignment option, TCINST 560
alloc.h 8
allocation, memory 39
data segment 65
changing 299
dynamic 68, 140, 239, 292
far heap 111, 113
graphics memory 193

heap 68, 140, 239, 292
allocmem (function) 39
allowed keystrokes 571
ANSI C standard 5
ANSI-compatible code 452
ANSI violations 453
ANSI Violations menu, TCINST 561
arc (function) 40
arc cosine 39
arc sine 44
arc tangent 46
argc (argument to main) 17
Args option, TCINST 567
argument list, variable 449
argv (argument to main) 17
ASCII, conversion to 43, 390
asctime (function) 43
asin (function) 44
aspect ratio 151
correction factor 315
assembly code
inline 455
output files 455
assert (function) 45
assert.h 8
assertion 45
atan2 (function) 46
atan (function) 46
atexit (function) 47
atof (function) 48
atoi (function) 49
atol (function) 50
attribute bits 86, 89, 258
attribute word 72, 86, 89
attributes, text 381
Autodependencies option, TCINST 559
autodetection 160, 207
Autoindent mode 411, 564
Autoindent Mode option, TCINST 564
Autoindent On/Off command (TC editor) 417
automatic recalculation, MicroCalc 578

B

background color 152, 191
setting 315
backspace command (TC editor) 415
Backspace Unindents option, TCINST 564
Backup Source Files option, TCINST 563
Backus-Naur form 541
bar
three-dimensional 51
two-dimensional 50
bar (function) 50
bar3d (function) 51
base 10 logarithm 232
base file name macro 481
basic cursor movement commands (TC editor) 412, 413
baud rate 55
BBS segment
class, renaming 561
group, renaming 561
renaming 561
bdos (function) 52
bdosptr (function) 53
BGIOBJ (graphics converter) 201, 461, 522
/F option 525
advanced features 526
command-line syntax 523, 526
components 526
example 524
binary mode 27, 87, 89, 118, 134, 142, 328
binary search 66
bios.h 8
BIOS interrupts
0x11 60
0x12 63
0x13 57
0x16 61
0x17 64
0x1A 65
BIOS timer 64
bioscom (function) 54
biosdisk (function) 57

biosequip (function) 60
bioskey (function) 61
biosmemory (function) 63
biosprint (function) 64
biostime (function) 64
bit image
  memory required to store 200
  saving to memory 169
  writing to screen 281
bit mask 146, 360
bit rotation
  long integer 235
  unsigned integer 298, 299
blank columns and rows, inserting, MicroCalc 578
blink-enable bit 381
blocks 416
  adjusting size of 316
    in far heap 115
    in heap 292
  commands 412, 416
  copying 246, 248, 249, 254
  initializing 249, 328
  markers 414, 416
  searching, for character 247
Borland
  CompuServe Forum 4
  mailing address 4
  technical support 4
Break Make On menu, TCINST 558
break value 65, 299
brk (function) 65
bsearch (function) 66
buffered stream 317, 336
buffering
  file 336
  stream 317, 336
buffers
  clearing 129
  flushed when stream closed 116
  freeing 116
  graphics, internal 324
  keyboard, pushing character to 395
  writing to ouput streams 129
built-in DOS commands, executed by MAKE 477
BUILTINS.MAK 488
byte aligning 560
bytes
  copying 252
  reading from hardware ports 206
  swapping 378
  writing to hardware ports 259

C

C calling sequence 559
C usage 449
C0x.OBJ start-up object file 456
cabs (function) 67
Calling Convention option, TCINST 559
calling sequences 559
calloc (function) 68
carry flag 210, 211, 212, 213
case sensitive flag, TLIB 510, 514
Case-Sensitive Link option, TCINST 562
case sensitivity in TLINK 502
cdecl statement 449
ceil (function) 69
cells 579
  formulas in, examples 580
  types 579
CGA graphics problems 40, 264
cgets (function) 69
char treated as type unsigned 448
characters
  color, setting 381, 384
  device 219
  lowercasing 390, 391
  magnification, user-defined 334
  pushing
    to input stream 394
    to keyboard buffer 395
  reading
    from console 154, 155
    from *stdin* 121, 154
    from stream 121, 153
  searching
    in block 247
    in string 362
  sets 522
    linking 522

size 333
uppercasing 391, 392
writing
to screen 280
to *stdout* 139, 280
to stream 138, 279
chdir (function) 71
checking
current driver 160
device type 219
end-of-file 105, 119, 291
keystroke 225
child process 106, 352
chmod (function) 72
choosing menu items, TCINST 558
chsize (function) 73
circle (function) 74
Clear Project option, TCINST 559
cleardevice (function) 75
clearerr (function) 76
clearing
screen 80, 325
to end-of-line 79
clearviewport (function) 76
clock (function) 77
close (function) 78
closegraph (function) 79
clreol (function) 79
clrscr (function) 80
co-routines 233, 326
code generation command-line options 446, 448
Code Generation menu, TCINST 559
code segment
class, renaming 454, 561
group, renaming 561
renaming 454, 561
color table, palette 314, 315, 329
colors
background 152, 191
setting 315
character, setting 381, 384
drawing 155, 191, 264, 293, 311
setting 318
fill 50, 51, 124, 128, 264, 311
information on, returning 165
setting 322
text background, setting 381, 383
value, maximum 172
column width, setting, MicroCalc 578
COMMAND.COM 379
invoked by MAKE 477
command line
arguments 567
compiler options
#defines 447
code generation 446, 448
compilation control 446, 455
error-reporting 446, 452
macro definition 446, 447
memory model 446, 447
optimization 446, 449
segment-naming control 446, 454
source code 446
configuration files 458
error 423
options 443, 446
CPP 462
environment 455
GREP 515
default settings 517
order of precedence 517
linker 455
MAKE 488, 489
syntax of 445
table 443
TLIB 509
toggling 445
switches, enabling and disabling 445
syntax
BGIOBJ 523, 526
CPP 462
GREP 515
MAKE 487
OBJXREF 529, 532, 533, 538
wildcards in 529
TLIB 509
TLINK 494
Turbo C *See also* TCC

command list (MAKE)
command body 476, 477
prefixes 476
commands *See also* menu commands
Interactive Editor 411
macros expanded in 480
MicroCalc 581
comments
makefile 469
nested 452
common errors 453
Common Errors menu, TCINST 561
comparison function, user-defined 284
compilation 447
control command-line options 446, 455
rules governing 445
Compile menu, TCINST 557, 558
compiler
command-line options 446
#defines 446, 447
code generation 446, 448
compilation control 446, 455
error-reporting 446, 452
macro definition 446, 447
memory model 446, 447
optimization 446, 449
segment-naming control 446, 454
source code 446
diagnostic messages 423
Compiler menu, TCINST 559
CompuServe Forum, Borland 4
COMSPEC environment variable 379
conditional execution directive 484
syntax 484
conditional execution directives 469
Config Auto Save option, TCINST 563
configuration
current, saved automatically 563
configuration files
command-line 458
TCC 462
overriding 445
TCINST overridden by 556
conio.h 8
console I/O 90, 154, 155
constants 542
manifest 446
symbolic 446
constructs, Turbo C, syntax 541
continuation character 469
control-break
handler 91
interrupt 198
setting 318
returning 153
control characters 418
control word, floating-point 80
conversion
date-time 43, 91
to DOS format 395
to Greenwich Mean Time 188
to structure 230
to UNIX format 98
double to integer and mantissa 252
double to mantissa and exponent 142
floating-point to string 101, 117, 149
integer to string 224
long integer to string 238
lowercase to uppercase 378, 391, 392
specifications (printf) 268
string
to double 374
to floating-point 48
to integer 49
to long integer 50, 376
to unsigned long integer 377
to ASCII 43, 390
unsigned long integer to string 394
uppercase to lowercase 368, 390, 391
coordinates
arc, returning 150
screen, maximum 173
coprocessor, 8087/
80287, floating-point problems 137

copy block command (TC editor) 416
coreleft (function) 82
correction factor of aspect ratio 315
cos (function) 82
cosh (function) 83
cosine 82
cosine, hyperbolic 83
country (function) 83
country-dependent data 83
CP 187, 188, 191, 228, 229, 325, 331, 339
  moving 253, 254
CPP (preprocessor) 461
  command-line options 462
  command-line syntax 462
cprintf (function) 85
cputs (function) 85
creat (function) 87
creatnew (function) 88
creattemp (function) 89
cscanf (function) 90
ctime (function) 91
ctrlbrk (function) 91
ctype.h 8
currency symbols 84
current position (graphics) 187, 188, 191, 228, 229, 325, 331, 339
  moving 253, 254
cursor 409
  position in text window, returning 405
  positioning in text window 190
customization program (TCINST) 555
Customize colors menu, TCINST 573
customizing
  keystroke commands 557
  multiple versions of Turbo C 556
  TC.EXE 555
Cx.LIB 456

D

data bits 55
data segment 28, 68, 82, 239
  allocation 65
    changing 299
  class, renaming 454, 561
  group, renaming 454, 561
  renaming 454, 561
date
  file 167, 323
  system 43, 91, 98, 148, 188, 230, 395
    returning 158
    setting 320, 361
date-time conversion 43, 91, 98, 188, 230, 395
daylight (global variable) 22
Debug menu, TCINST 557, 567
debugger, symbolic 449
debugging information, in .EXE or OBJ file 449
declarations 548
Default Char Type option, TCINST 560
Default color set menu, TCINST 573
default graphics settings 191
Default Libraries option, TCINST 562
defined test macro 481
Defines option, TCINST 559
delay (function) 93
delete block command (TC editor) 416
delete character command (TC editor) 415
delete line command (TC editor) 415
delete to EOL command (TC editor) 416
deletion
  file 396
  line 93
delline (function) 93
dependencies, file 465
  checked by MAKE 467
detectgraph (function) 94
detection
  error, on stream 120
  graphics adapter 94, 201
device
  channels 216
  character 219
  driver table 207
  drivers
    DOS 216

vendor added 207
errors 196
type checking 219
diagnostic messages
compiler 423
difftime (function) 96
dir.h 8
directives 469, 483
conditional execution 469, 484
syntax 484
error detection 469, 486
syntax 486
file-inclusion 469, 483
macro undefinition 469, 487
syntax 487
macros in 480
nested 483, 485
Directories menu 456
TCINST 565
directory
creating 250
deleting 297
disk, search of 125, 127
working 353
changing 71
returning 156, 157
directvideo (global variable) 23
disable (function) 96
disabling
command-line switches 445
interrupts 97
warning messages 452
disk
directory, search of 125, 127
drive, setting 320
errors 196
access 423
I/O 57
operations sent to BIOS 57
sectors
absolute 36, 37
reading/writing 36, 37, 58
space, returning 159
writes, verifying 186, 338
disk-transfer address, DOS 125, 127, 288
returning 161
setting 321
Display Swapping option, TCINST 567
div (function) 97
division, integer 97
long 227
DOS
commands 379
device drivers 216
disk-transfer address 125, 127, 288
returning 161
setting 321
environment
adding data to 281
returning data from 161
error codes 25
error information, extended 98
functions
0x19 160
0x31 225
interrupt functions 184, 338
interrupt handlers 92, 196
interrupt interface 212, 213
interrupts
0x21 212, 213
0x23 92, 198
0x24 196
0x25 36
0x26 37
memory, memory freeing in 141
path, searching for file 310
search algorithm 106, 477
system calls 52, 53, 197, 290
0x27 288
0x28 289
0x29 262
0x33 153, 318
0x44 216
0x48 39
0x59 98
0x62 179
0x4E 125
version numbers 29
dos.h 8
dosexterr (function) 98

dostounix (function) 98
drawing color 155, 191, 264, 293, 311
  setting 318
drawpoly (function) 99
drive number, returning current 160
driver, current, name of 160
DTA 125, 127, 288
  returning 161
  setting 321
dup2 (function) 101
dup (function) 100
duplicate symbols, TLINK warning 502
dynamic memory allocation 68, 140, 239, 292

E

echoing to screen 155
ecvt (function) 101
Edit Auto Save option, TCINST 563
Edit window 409
  status line 410
Editor commands (TC) 411
  abort operation 417
  Autoindent On/Off 417
  backspace 415
  basic cursor movement 412, 413
  block 412, 416
  copy block 416
  delete block 416
  delete character 415
  delete line 415
  delete to EOL 416
  delete word 415
  find place marker 420
  hide/display block 416
  insert and delete 412, 415
  insert control character 418
  insert line 415
  Insert mode On/Off 415
  load file 420
  mark block-begin 416
  mark block-end 416
  mark single word 416
  move block 417
  Optimal fill On/Off 420
  outdent 415
  pair matching 420
  print file 420
  quick cursor movement 412, 414
  quit-no save 420
  read block 417
  repeat last search 420
  restore line 420
  save file 420
  search 418
    backward 418
    examples 419
    local 418
    not case sensitive 418
    *n*th occurrence 418
    whole word 418
  search and replace 419
    examples 419
    global 419
    next *n* occurrences 419
  set place marker 421
  tab 421
  tab On/Off 421
  table of 412
  unindent On/Off 421
  write block 417
Editor commands option, TCINST 557, 567
editors
  MicroCalc 577
  Sidekick 409
  Turbo C Interactive 409
  Turbo Pascal 409
ellipse 124
ellipse (function) 102
elliptical arc 102
elliptical pie slice 311
EMU.LIB 456
emulation
  option *See* –f emulation option
emulation of 8087 calls 448
enable (function) 105
enabling
  command-line switches 445
  interrupts 105
  warning messages 452
end-of-file checking 105, 119, 291

end-of-line, clearing to 79
env (argument to main) 17
environ (global variable) 18, 24
environment
DOS
adding data to 281
returning data from 161
variables 24
COMSPEC 379
PATH 107, 353
environment command-line options 455
Environment menu, TCINST 562
eof (function) 105
errno (global variable) 24
errno.h 8
error handlers
floating-point 241
hardware 196, 198
user-modifiable math 242
error-reporting command-line options 446, 452
errors
codes 25
graphics, returning 194
mnemonics 8, 25, 26
command line 423
common 453
detection, on stream 120
detection directives 469
syntax 486
disk access 423
fatal 423
information, extended DOS 98
less common 453
locked file 231
MAKE 491
memory access 423
messages 24, 424
compiler 423
fatal 424
graphics, returning 191, 194
MAKE 490
OBJXREF 539
pointer to, returning 365, 366
system, returning 263
TC compiler 561
read/write 120
syntax 423
Errors menu, TCINST 560
exception handlers, 8087/80287 361
exception handlers, 8087/80287 coprocessor 75
exceptions, floating-point 80
exec... (function) 106
execution, suspending 93, 348
exit (function) 110
exit codes 35
exit function 47
exit status 110, 225
exp (function) 110
expansion, macro 461, 462
explicit library files 457
explicit rule 469, 470
command list 476
examples 471
executed by MAKE 470
source files in 470
syntax 470
target file in 470
exponent 142
exponential 110
expressions 545
extended 80186 instructions 448
Extended Dictionary 503, 508, 513
creating 513
flag, TLIB 510
extended error information, DOS 98
extension keywords, Turbo C 452
extensions, file, supplied by TLINK 494
external definitions 552

F

fabs (function) 111
far heap
allocation of memory from 111, 113
measure of unused memory in 112
memory freeing in 113
reallocation of memory in 115
far pointer
address segment of 251

returning 138
creation 251
offset of 251
returning 136
to block in far heap 112, 113, 115
farcalloc (function) 111
farcoreleft (function) 112
farfree (function) 113
farmalloc (function) 113
farrealloc (function) 115
fatal errors 423
MAKE 490
messages 424
TLINK 504, 505
FCB 288, 289
fclose (function) 116
fcloseall (function) 116
fcntl.h 8
fcvt (function) 117
fdopen (function) 117
features, MicroCalc 577
feof (function) 119
ferror (function) 120
fflush (function) 120
fgetc (function) 121
fgetchar (function) 121
fgetpos (function) 122
fgets (function) 122
figures, flood-filling 127
file-access permissions 72, 360
file-allocation table 163
file handles 78, 100, 101, 258
duplication of 100, 101
linking to stream 117
returning 123
file-inclusion directive 483
file overwrite command (TC editor) 417
file-sharing 396
attributes 256
locks 231, 396
filelength (function) 123
fileno (function) 123
files
access, read/write 38, 72, 87, 89, 147, 257, 258, 360
accessibility, determining 37
attribute bits 86, 89, 258
attribute word 72, 86, 89
attributes 71, 72, 87, 258
binary 389
buffering 336
closing 78, 116, 141
control block 288, 289
creation of 86, 87, 88, 89
date 167, 323
date and time of 167, 323
deleting 295, 396
dependencies 465
checked by MAKE 467
graphics driver 201
I/O 121, 122, 137, 138, 139, 140, 143, 149, 153, 187, 290, 291, 336, 399, 400, 406, 407
inclusion directive 469
information on, returning 146, 359
library 529
linker response+, used by OBJXREF 538
linker response, used by OBJXREF 532
linking file handles to 117
name
parsing 261
unique, generating 251, 389
name and extension macro 482
name macros 481
name only macro 483
name path macro 482
object 529
opening 134, 141, 256, 257
for update 118, 134, 142, 389
overwriting 87
pointer
initializing 297
read/write 238
resetting 144, 291
returning 122, 147, 380
setting 145, 258
project
used by OBJXREF 532, 538
reading from 290, 291

renaming 296
replacing 141
response 529, 531
freeform 531, 538
linker 532, 538
rewriting 86, 87
scratch 389
searcher (GREP) 461, 515
size of
changing 73
returning 123
specifications, GREP 519
time 167, 323
translation 27
fill colors 50, 51, 124, 128, 264, 311
information on, returning 165
setting 322
Fill mode 411
fill patterns 50, 51, 124, 128, 191, 264, 311
information on, returning 165
predefined 165
setting 322
user-defined 164, 165, 321, 322
fill style 191
fillellipse (function) 124
filling a figure 127
fillpoly (function) 124
find place marker command (TC editor) 420
findfirst (function) 125
findnext (function) 127
flags
access 349
read/write 348
float.h 8
floating-point
chip 448
control word 80
conversion 101, 117, 149
error handling 241
exceptions 80
libraries 448
math package 136
operations 448
status word 75
Floating Point option, TCINST 560
floating-point status word 361
floodfill (function) 127
flooding a figure 127
floor (function) 129
flushall (function) 129
flushing, stream 120, 129
fmod (function) 130
fnmerge (function) 130
fnsplit (function) 132
fonts
adding to graphics library 523
bit-mapped 333
files, converting to .OBJ 523
included with Turbo C 524
linked-in 295
linking 522
registering 523
stroked 209, 333, 334
fopen (function) 134
format specifications 85, 90, 137, 143, 268, 300, 301, 357, 359, 400, 401, 402, 403, 404
argument-type modifiers 302, 308
assignment-suppression character 302, 307, 309
conversion type characters 268, 269
flag characters 268, 272
alternate forms 273
inappropriate character in 309
input-size modifiers 268, 276
precision specifier 268, 274
size modifiers 302, 308
type characters 302, 303
width specifier 268, 274, 302, 307, 309
format string 85, 90, 137, 143, 268, 300, 301, 357, 359, 400, 401, 402, 403, 404
conventions 304
input fields 304
range facility shortcut 305
using hyphen to set range 306
FP87.LIB 456
FP_OFF (function) 136

FP_SEG (function) 138
fprintf (function) 137
fputc (function) 138
fputchar (function) 139
fputs (function) 139
frame base pointer 233, 326
fread (function) 140
free (function) 140
freemem (function) 141
freopen (function) 141
frexp (function) 142
fscanf (function) 143
fseek (function) 144
fsetpos (function) 145
fstat (function) 146
ftell (function) 147
ftime (function) 148
full file name macro 482
full link map 455
functions, MicroCalc 577, 580
fwrite (function) 149

G

ganging
  #defines command-line options 447
  include command-line options 456
  library command-line options 456
  macro definition command-line options 447
gcvt (function) 149
Generate Underbars option, TCINST 560
geninterrupt (function) 150
getarccoords (function) 150
getaspectratio (function) 151
getbkcolor (function) 152
getc (function) 153
getcbrk (function) 153
getch (function) 154
getchar (function) 154
getche (function) 155
getcolor (function) 155
getcurdir (function) 156
getcwd (function) 157
getdate (function) 158
getdefaultpalette (function) 159
getdfree (function) 159
getdisk (function) 160
getdrivername (function) 160
getdta (function) 161
getenv (function) 161
getfat (function) 163
getfatd (function) 163
getfillpattern (function) 164
getfillsettings (function) 165
getftime (function) 167
getgraphmode (function) 168
getimage (function) 169
getlinesettings (function) 170
getmaxcolor (function) 172
getmaxmode (function) 172
getmaxx (function) 173
getmaxy (function) 173
getmodename (function) 174
getmoderange (function) 175
getpalette (function) 175
getpalettesize (function) 177
getpass (function) 178
getpixel (function) 178
getpsp (function) 179
gets (function) 179
gettext (function) 180
gettextinfo (function) 181
gettextsettings (function) 182
gettime (function) 183
getvect (function) 184
getverify (function) 185
getviewsettings (function) 186
getw (function) 187
getx (function) 187
gety (function) 188
global time variables, setting 392
global variables 22
  _8087 23
  _argc 22
  _argv 22
  _doserrno 24
  _fmode 27
  _heaplen 28
  _osmajor 29
  _osminor 29
  _psp 29

_stklen 30
_version 31
daylight 22
directvideo 23
environ 24
errno 24
sys_errlist 24
sys_nerr 24
timezone 31
tzname 31
GMT 31, 148, 392
gmtime (function) 188
goto, nonlocal 92, 233, 325
gotoxy (function) 190
graphdefaults (function) 191
grapherrormsg (function) 191
graphics
adapters 94
buffer
internal 324
converter (BGIOBJ) 461, 522
drivers 94, 201, 522
adding to graphics library 523
code 293
converting to .OBJ 523
file 201
included with Turbo C 524
linking 522
modes, range of 175
registering 523
error codes, returning 194
error messages 194
I/O 312, 339
library 192
memory
allocation of memory from 193
memory freeing in 192
mode 94, 201, 296, 325, *See* screen
operating mode, *See* operating
mode of screen
current, returning 168
modes
name of 174
screens, clearing 75
settings, default 191
system
closing down 79
initializing 201
text font 191
information on, returning 182
graphics.h 8
GRAPHICS.H header file 527
graphresult (function) 194
Greenwich Mean Time 31, 96, 148,
188, 392
GREP (file searcher) 461, 515
command-line options 515
default settings 517
order of precedence 517
command-line syntax 515
examples 519
file specifications 519
operators in regular expressions
518
search string 517
whitespace in 520
GREP.COM 517

H

handlers 341
error 196, 198, 241, 242
exception 75
interrupt 343
signal 286, 341, 345
user-specified 341
handles, file 78, 100, 101, 258
duplication of 100, 101
linking to stream 117
returning 123
harderr (function) 196
hardresume (function) 198
hardretn (function) 198
hardware
error handlers 196, 198
information, returning 60
interrupts 105
ports 205, 206, 259
header files 34
GRAPHICS.H 527
MCALC.H 579

heap 82
allocation of memory from 68, 140, 239, 292
length 28
memory freeing in 140
reallocation of memory in 292
heap, far
allocation of memory from 111, 113
measure of unused memory in 112
memory freeing in 113
reallocation of memory in 115
hide/display block command (TC editor) 416
high intensity 199
bit, setting 199
highvideo (function) 199
hotkeys, TC 412
hyperbolic cosine 83
hyperbolic sine 347
hyperbolic tangent 380
hypot (function) 199
hypotenuse 199

I

I/O
console 90, 154, 155
disk 57
file 121, 122, 137, 138, 139, 140, 143, 149, 153, 187, 290, 291, 336, 399, 400, 406, 407
graphics 312, 339
port 54, 205, 206, 259
screen 85, 280
stream 118, 121, 134, 137, 138, 139, 140, 143, 149, 153, 154, 179, 187, 267, 279, 280, 283, 284, 300, 317, 336, 399, 400, 401, 402, 404
terminated 309
string 69, 85, 122, 139, 179, 260, 261, 283, 357, 358, 403
identifiers 542
Pascal-type 449
significant length of 452
imagesize (function) 200
implicit library files 457
implicit rule 469, 473
command list 476
examples 473, 474, 475
source extension 473
syntax 473
target extension 473
include command-line options
ganging 456
multiple listings 456
include directories 456
multiple 458
Include Directories option, TCINST 565
Include directories setting 456
include files 5, 8
search algorithms 457
user-specified
search for 455
indicator
end-of-file 76
error 76
infinity, floating-point 80
initgraph (function) 201
initialization modules, used with TLINK 496, 497
Initialize Segments option, TCINST 561
initialized data segment
class, renaming 454
group, renaming 454
renaming 454
inline 8087 instructions 448
inline assembly code 455, *See* assembly code, inline
inport (function) 205
inportb (function) 206
input fields
not scanned 309
scanned but not stored 309
insert control character command (TC editor) 418
insert line command (TC editor) 415
Insert mode 411, 564
Insert mode On/Off command (TC editor) 415
Insert Mode option, TCINST 564
insline (function) 206

Install Editor screen 568, 569
Installation menu, TCINST 557
installuserdriver (function) 207
installuserfont (function) 209
Instruction Set option, TCINST 560
instruction sets 560
int86 (function) 209
int86x (function) 211
intdos (function) 212
intdosx (function) 213
integers
  aligned on word boundary 448
  conversion 224
  division 97
    long 227
  reading from stream 187
  writing to stream 284
integrated debugger 560, 567
intensity
  high 199
  low 234
  normal 255
interactive editor, Turbo C 409
internal graphics buffer 324
interrupt control 97, 150
interrupt functions, DOS 184, 338
interrupt handlers 343
  DOS 196
interrupt vectors 92
  8086 184, 338
  returning 184
  setting 338
interrupts
  disabling 97
  enabling 105
  hardware 105
  software 150, 210, 211, 215
intr (function) 215
invoking
  MicroCalc 579
  TCINST 556
io.h 8
ioctl (function) 216
isalnum (function) 218
isalpha (function) 218
isascii (function) 219
isatty (function) 219
iscntrl (function) 220
isdigit (function) 220
isgraph (function) 221
islower (function) 221
isprint (function) 222
ispunct (function) 222
isspace (function) 223
isupper (function) 223
isxdigit (function) 224
itoa (function) 224

J

Jump Optimization option, TCINST 560

K

kbhit (function) 225
keep (function) 225
Keep Messages option, TCINST 563
keyboard operations 61
Keystroke commands
  Ignore case 570
  Verbatim 570
  WordStar-like 570
keystroke commands
  customizing 557, 567, 568, 570
  Ignore case 571
  Verbatim 571
  WordStar-like 570
keystrokes
  allowed 571
  checking 225
keywords 542
  extension in Turbo C 452

L

labs (function) 226
ldexp (function) 226
ldiv (function) 227
less common errors 453
Less Common Errors menu, TCINST 561
lexical grammar 541
lfind (function) 228
librarian (TLIB) 461, 508

libraries
command-line options
ganging 456
multiple listings 456
default 562
default, ignored by TLINK 501
directories 456
multiple 458
entry headings 33
files 5
explicit 457
implicit 457
search algorithms 457
Turbo C 456
user-specified 457
user-specified search for 456
floating-point 448
name, TLIB 509
object file 508
routines
8087 floating-point emulation 448
Turbo C
floating point 498
math 498
rebuilding 449
run-time 498
used with TLINK 496, 497
Library Directories option, TCINST 566
Library directories setting 456
library files 529
library routines 5
License statement 3
limits.h 8
line (function) 228
line numbers, in object files 449
Line Numbers option, TCINST 560
linear search 228, 236
linerel (function) 229
lines
blank, inserting 206
deletion of 93
drawing
between points 228
from CP 229
relative to CP 229
pattern of 170
style of 170, 293, 326
thickness of 170, 293, 326
lineto (function) 229
link map, full 455
linked-in font 295
linked-in graphics drivers code 293
linker (TLINK) 461, 493
linker command-line options 455
linker error: `segment exceeds 64K` 525
Linker menu, TCINST 561
linker response files
used by OBJXREF 532, 538
linking
character sets 522
fonts 522
graphics drivers 522
list file, TLIB 510
listing file, preprocessor 462
compiling 462
examples 463
used in debugging 462
literal strings 517
merging 448
load file command (TC editor) 420
load operations
redundant, suppressing 450
localtime (function) 230
lock (function) 231
locks, file-sharing 231, 396
log10 (function) 232
log (function) 232
logarithm
base 10 232
natural 232
long integer conversion 238, 394
longjmp (function) 233
low intensity 234
lowvideo (function) 234
lsearch (function) 236
lseek (function) 238
ltoa (function) 238

M

machine language instructions
inserted into object code 103
macros
base file name 481
defined test 481
definition 478
definitions
command-line options 446, 447
example 478
in makefile 469
syntax 479
expansion 461, 462
file name 481
file name and extension 482
file name only 483
file name path 482
full file name 482
macros in 480
predefined 481
preprocessor 462
undefinition directive 487
syntax 487
undefinition directives 469
macros. definitions
options
ganging 447
macros, definitions, default 559
Magenta color set menu, TCINST 573
main (function) 17
arguments passed to 17
declaring 17
example 18
command-line arguments 19
wildcard expansion 19
compiled with Pascal calling
conventions 20
declared as C type 20
value returned by 21
main menu 409
MAKE (program manager) 461, 463
BUILTINS.MAK file 488
command-line examples 488
command-line options 488, 489
command-line syntax 487
command-line target files 488
error messages 490
errors 491
examples 464, 467
fatal errors 490
file updating by 470
makefile search algorithm 489
terminating execution of 488
TOUCH utility 493
using TCC and TLINK with 464
makefile 466
base file name macro 481
command lists 476
command body 476, 477
prefixes 476
commands, macros expanded in
480
comments 469
examples 469
components 469
conditional execution directive
469, 484
syntax 484
continuation character 469
creating 466, 469
defined test macro 481
directives 469, 483
macros in 480
error detection directive 469, 486
syntax 486
explicit rules 469, 470
command lists 476
examples 471
executed by MAKE 470
source files in 470
syntax 470
target file in 470
file inclusion directive 469
file-inclusion directives 483
file name and extension macro 482
file name macros 481
file name only macro 483
file name path macro 482
full file name macro 482
implicit rules 469, 473
command lists 476
examples 473, 474, 475

source extension 473
syntax 473
target extension 473
interpreted by MAKE 466
macro definitions 469, 478
example 478
syntax 479
macro invocation 480
macro undefinition directive 469, 487
syntax 487
macros, macros in 480
predefined macros 481
SET environment strings 481
using 467
makes, default conditions for stopping 558
malloc (function) 239
manifest constants 446
mantissa 142, 252
map
of executable file, generated by TLINK 499
map file 561
generated by TLINK 495
Map File option, TCINST 561
mark block-begin command (TC editor) 416
mark block-end command (TC editor) 416
mark word command (TC editor) 416
marked text 416
math error handler, user-modifiable 242
math.h 8
math package, floating-point 136
matherr (function) 242
MATHx.LIB 456
max (function) 245
maximum color value 172
MCALC.H, header file 579
mem.h 8
memccpy (function) 246
memchr (function) 247
memcmp (function) 247
memcpy (function) 248
memicmp (function) 248
memmove (function) 249
memory
access error 423
address specified
returning byte from 262
returning word from 262
storing byte at 265
storing integer at 265
addressing 559
allocation of 39
data segment 65
data segment, changing 299
dynamic 68, 140, 239, 292
far heap 111, 113
graphics memory 193
heap 68, 140, 239, 292
bit image, saving to 169
copying 246, 248, 249, 254
in small and medium memory models 252
freeing
in DOS memory 141
in far heap 113
in graphics memory 192
in heap 140
in small and medium memory models 113
initializing 249
management of graphics library 192
measure of unused, returning 82
in far heap 112
models 5
reallocation of
far heap 115
heap 292
screen segment, copying to 180
memory model command-line options 446, 447
memory models 559
memset (function) 249
menu items, TCINST, choosing 558
menu options, TCINST
Alignment 560
Args 567

Auto Dependencies 559
Autoindent Mode 564
Backspace Unindents 564
Backup Source Files 563
Calling Convention 559
Case-Sensitive Link 562
Clear Project 559
Config Auto Save 563
Debug 567
Default Char Type 560
Default Libraries 562
Defines 559
Display Swapping 567
Edit Auto Save 563
Editor commands 557, 567
Floating Point 560
Generate Underbars 560
Include Directories 565
Initialize Segments 561
Insert Mode 564
Instruction Set 560
Jump Optimization 560
Keep Messages 563
Library Directories 566
Line Numbers 560
Map File 561
Merge Duplicate Strings 560
Message Tracking 562
Model 559
Optimal Fill 564
Optimize For 560
Output Directory 566
Pick File Name 566
Primary File 558
Project Name 558
Quit/Save 558, 574
Register Optimization 560
Resize Windows 558, 574
Stack Warning 562
Standard Stack Frame 560
Tab Size 564
Test Stack Overflow 560
Turbo C Directory 566
Use Register Variables 560
Use Tabs 564
VGA/EGA Save Fonts 563
Warn Duplicate Symbols 562
Zoomed Windows 563
menu settings
Tab size 421
menu system
TC, TCINST overridden by 556
TCINST 556
menus
Directories 456
main 409
Optimization 450
Options 443, 456
TCINST
ANSI Violations 561
Break Make On 558
Code Generation 559
Common Errors 561
Compile 557, 558
Compiler 559
Customize colors 573
Debug 557
Default color set 573
Directories 565
Environment 562
Errors 560
excaping out of 558
Installation 557
Less Common Errors 561
Linker 561
Magenta color set 573
Mode for Display 557, 572
Names 561
Optimization 560
Options 557, 559
Options for Editor 564
Portability Warnings 561
Project 557, 558
Screen Size 563
Set Colors 557, 573
Source 560
Turquoise color set 573
Merge Duplicate Strings option, TCINST 560
merging, path 130
Message Tracking option, TCINST 562

MicroCalc 577
commands 581
compiling 578
entering data 580
features 577
automatic recalculation 578
inserting blank columns and rows 578
line editor 577
math functions 577
parser 582
setting column width 578
functions 580
invoking 579
operators 580
sample screen 579
min (function) 250
MK_FP (function) 251
mkdir (function) 250
mktemp (function) 251
mnemonics, error code 8, 25, 26
Mode for Display menu, TCINST 557, 572
Model option, TCINST 559
modes
access 359
changing 71, 72
binary 27, 87, 89, 118, 134, 142, 328
current graphics, returning 168
file-translation 87, 89
graphics 94, 201, 296, 325
graphics, name of 174
maximum number, for current driver 172
range of, on graphics driver 175
screen, restoring 296
text 27, 87, 89, 118, 134, 142, 180, 181, 283, 291, 325, 328, 386
modf (function) 252
module names, TLIB 511
modules, object 529
modulo 130
monochrome adapter graphics problems 40, 264
move block command (TC editor) 417
movedata (function) 252
moverel (function) 253
movetext (function) 253
moveto (function) 254
movmem (function) 254
multi-file programs, managing 465
multiple directories, include and library 458
multiple listings
#defines command-line options 447
include command-line options 456
library command-line options 456
macro definition command-line options 447

N

names, public 529
Names menu, TCINST 561
natural logarithm 232
nested comments 452
nested directives 483, 485
NMI interrupt 97
nonfatal errors, TLINK 504, 507
nonlocal goto 92, 233, 325
normvideo (function) 255
nosound (function) 255
numbers, pseudo-random 287

O

object files 529
directories, searched by OBJXREF 533
libraries 508
advantages of using 509
creating 512
managed by TLIB 508
line numbers in 449
object module cross-referencer (OBJXREF) 461, 528
command-line options 529, 530, 532, 533
control options 530, 539
file type 532, 538
report options 530, 533
error messages 539
examples using 538

reports
examples 533
options 530
outputting 530
response files 531
summary of available options 530
warnings 539
object modules 529
OBJXREF (object module cross-referencer) 528
offset, of far pointer 136, 251
open (function) 257
operation list, TLIB 510
operators 545
GREP 518
MicroCalc 580
Optimal fill mode 411
Optimal fill On/Off command (TC editor) 420
Optimal Fill option, TCINST 564
optimization command-line options 446, 449
Optimization menu 450
TCINST 560
Optimize For option, TCINST 560
options
command-line 443, 446
linker 455
syntax of 445
table 443
toggling 445
compiler 446
Options for Editor menu, TCINST 564
Options menu 443, 456
TCINST 557, 559
outdent command (TC editor) 415
outport (function) 259
outportb (function) 259
Output Directory option, TCINST 566
output files, assembly code 455
outtext (function) 260
outtextxy (function) 261
Overwrite mode 411

P

page, active 312
page numbers, visual 339
pair matching command (TC editor) 420
palettes 191, 204, 315, 318, 325
changing colors of 313, 329
color table 314, 315, 329
default 159
definition structure 159
information on, returning 159, 175
size of, returning 177
user-defined, for the IBM8514 330
parameter-passing sequence, Pascal 449
parent process 106, 352
parity 55
parser, MicroCalc 582
parsfnm (function) 261
Pascal *See* Turbo Pascal
calling conventions
compiling main with 20
calling sequence 559
identifiers of type 449
parameter-passing sequence 449
password 178
PATH environment variable 107, 353
path merging 130
path splitting 132
patterns, fill 50, 51, 124, 128, 191, 264, 311
information on, returning 165
setting 322
user-defined 164, 322
PC speaker 255, 350
peek (function) 262
peekb (function) 262
permissions, file-access 72, 360
perror (function) 24, 263
phrase structure grammar 541, 545
Pick File Name option, TCINST 566
pie slice 264
elliptical 311
pieslice (function) 264
pipes 477, 519

pixel color
  plotting 282
  returning 178
poke (function) 265
pokeb (function) 265
poly (function) 266
polygon 99, 124
polynomial equation 266
port I/O 54, 205, 206, 259
portability 34
  warnings 453
Portability Warnings menu, TCINST 561
pow10 (function) 267
pow (function) 266
powers
  calculating ten to 267
  calculating values to 266
precision, floating-point 80
predefined macros 481
preprocessing directives 541, 552
preprocessor (CPP) 461
  listing file 462
    compiling 462
    examples 463
    used in debugging 462
  macro 462
Primary File option, TCINST 558
print file command (TC editor) 420
printer functions 64
printf (function) 267
process.h 9
program manager (MAKE) 461, 463
program segment prefix 29, 179
program termination 109, 110
project files
  used by OBJXREF 532, 538
Project-Make 464
Project menu, TCINST 557, 558
Project Name option, TCINST 558
Prolog *See* Turbo Prolog
prototype 34
pseudo-random numbers 287
pseudo-variables, register 452
PSP 29, 179
public names 529
punctuators 545
putc (function) 279
putch (function) 280
putchar (function) 280
putenv (function) 281
putimage (function) 281
putpixel (function) 282
puts (function) 283
puttext (function) 283
putw (function) 284

Q

qsort (function) 284
quick cursor movement commands (TC editor) 412, 414
Quick Reference Line *See also* Quick-Ref Line
quicksort algorithm 284
quit-no save command (TC editor) 420
Quit/Save option, TCINST 558, 574
quotient 97, 227

R

raise (function) 286
RAM
  measure of unused, returning 82
  resident program 225
  size of, returning 63
rand (function) 287
randbrd (function) 288
randbwr (function) 288
random (function) 289
random block read 288
random block write 288
random number generator 287, 289
  initializing 290, 358
random record field 288, 289
randomize (function) 290
range facility shortcut 305
read (function) 291
read access 38, 72, 87, 89, 147, 258, 360
read block command (TC editor) 417
read error 120
read/write flags 348
realloc (function) 292

reallocation, memory
far heap 115
heap 292
rebuilding Turbo C libraries 449
rectangle 293
rectangle (function) 293
redirection 477, 567
Register Optimization option, TCINST 560
register pseudo-variables 452
register variables 233, 326, 450
suppressed 450
toggle 450
registerbgidriver (function) 293, 523, 525, 527
registerbgifont (function) 294, 523, 525, 527
registerfarbgidriver (function) 525, 526, 527
registerfarbgifont (function) 525, 526, 527
registering routines 524
REGPACK structure 215
regular expressions 517
GREP, operators in 518
remainder 97, 130, 227
remove (function) 295
rename (function) 296
repeat last search command (TC editor) 420
reports (OBJXREF)
by class type 530, 536, 538
by external reference 531, 535
by module 530, 534
by public names 531, 534
by reference 531, 535, 538
default type 531, 538
examples 533
of module sizes 531, 536
of unreferenced symbol names 531, 537
options 530
verbose reporting 531, 537, 538, 539
Resize Windows option, TCINST 558, 574
response files 529
formats 531
freeform 531, 538
TLIB 513
TLINK 495
restore line command (TC editor) 420
restorecrtmode (function) 296
restoring screen 283
restrictions, TLINK 504
rewind (function) 297
rmdir (function) 297
rotation, bit
long integer 235
unsigned integer 298, 299
rounding
down 129
up 69
rounding, modes, floating-point 80
RS-232 communications 54
rule
explicit 469, 470
command list 476
examples 471
executed by MAKE 470
source files in 470
syntax 470
target file in 470
implicit 469, 473
command list 476
example 473
examples 474, 475
source extension 473
syntax 473
target extension 473
run-time library
functions by category 10
source code, licensing 7

S

save file command (TC editor) 420
saving screen 180
sbrk (function) 299
scanf (function) 300
Screen Size menu, TCINST 563
screens
clearing 80, 325
coordinates, maximum 173

echoing to 155
I/O 85, 280
MicroCalc 579
mode, restoring 296
restoring 283
saving 180
segment, copying to memory 180
settings
43/50 Line Display 564
25 Line Display 564
search and replace command (TC editor) 419
examples 419
next *n* occurrences 419
search command (TC editor) 418
backward 418
examples 419
local 418
not case sensitive 418
nth occurrence 418
whole word 418
search key 236
searches
algorithms
DOS 106
include files 457
library file 457
binary 66
block, for character 247
DOS path, for file 310
linear 228, 236
string
for character 362
for tokens 375
string, GREP 517
whitespace in 520
searching and appending 236
searchpath (function) 310
sector (function) 311
seed number 358
segment-naming control command-line options 446, 454
segread (function) 312
sequential records 228
Set Colors menu, TCINST 557, 573
SET environment strings 481
set place marker command (TC editor) 421
setactivepage (function) 312
setallpalette (function) 313
setaspectratio (function) 315
setbkcolor (function) 315
setblock (function) 316
setbuf (function) 317
setcbrk (function) 318
setcolor (function) 318
setdate (function) 320
setdisk (function) 320
setdta (function) 321
setfillpattern (function) 321
setfillstyle (function) 322
setftime (function) 323
setgraphbufsize (function) 324
setgraphmode (function) 325
setjmp (function) 325
setjmp.h 9
setlinestyle (function) 326
setmem (function) 328
setmode (function) 328
setpalette (function) 329
setrgbpalette (function) 330
settextjustify (function) 331
settextstyle (function) 332
settime (function) 334
settings *See also* menu settings
settings, graphics, default 191
setusercharsize (function) 334
setvbuf (function) 336
setvect (function) 338
setverify (function) 338
setviewport (function) 339
setvisualpage (function) 339
setwritemode (function) 340
share.h 9
shortcuts *See* hot keys
signal (function) 341
signal.h 9
signal handlers 286, 341, 345
user-specified 341
signals, software 286
sin (function) 347
sine 347

sine, hyperbolic 347
sinh (function) 347
size
  character 333
  file, changing 73
  palette, returning 177
sleep (function) 348
smart screen swapping 567
software
  interrupts 150, 210, 211
    interface 209, 211
  signal 286
software interrupts 215
  interface 215
sopen (function) 348
sort, quick 284
sound (function) 350
source code, run-time library, licensing 7
source code command-line options 446
source files 470
  backed up automatically 563
  extension 473
  saved automatically 563
  separately compiled 509
Source menu, TCINST 560
space on disk, returning 159
spawn... (function) 352
speaker, PC 255, 350
splitting, path 132
spreadsheet 577
sprintf (function) 357
sqrt (function) 357
square root 357
srand (function) 358
sscanf (function) 358
stack 68, 82, 239
  length 30
  overflow logic 448
  overflow message 448
  pointer 233, 326
stack, frame, standard 448
Stack Warning option, TCINST 562
standalone utilities 461
  file searcher (GREP) 461, 515
  graphics converter (BGIOBJ) 461, 522
  librarian (TLIB) 461, 508
  linker (TLINK) 461, 493
  object module cross-referencer (OBJXREF) 461, 528
  preprocessor (CPP) 461
  program manager (MAKE) 461, 463
standard stack frame 448
Standard Stack Frame option, TCINST 560
stat (function) 359
stat structure 146, 359
statements 551
status bits 55
status byte 59
status line, Edit window 410
status word
  8087/80287 361
  8087/80287 coprocessor 75
  floating-point 75, 361
stdargs.h 9
stdaux 116
stddef.h 9
stderr 9, 116, 141
stdin 9, 116, 121, 141, 179, 300, 402
stdio.h 9
stdlib.h 9
stdout 9, 116, 139, 141, 267, 280, 283, 401
stdprn 9, 116
stime (function) 361
stop bit 55
stpcpy (function) 362
strcat (function) 362
strchr (function) 362
strcmp (function) 363
strcmpi (function) 364
strcpy (function) 364
strcspn (function) 365
strdup (function) 365
streams
  buffered 317, 336
  closing 116, 141
  flushing 120, 129

I/O 118, 121, 134, 137, 138, 139, 140, 143, 149, 153, 154, 179, 187, 267, 279, 280, 283, 284, 300, 317, 336, 399, 400, 401, 402, 404
terminated 309
input, pushing character to 394
linking file handles to 117
opening 134, 141
replacing 141
unbuffered 317, 336
strerror (function) 366
stricmp (function) 367
string.h 9
string I/O 403
strings
appending 362, 368
comparison 247, 363, 368
ignoring case 248, 364, 367, 369, 370
conversion 48, 49, 50, 374, 376, 377
copying 362, 364, 365, 370
date-time 91
height, returning 386
I/O 69, 85, 122, 139, 179, 260, 261, 283, 357, 358
initializing 371, 372
length, calculating 367
literals 544
lowercasing 368
reversing 372
scanning
for character in set 371
for characters not in set 365
for last occurrence of character 372
for segment in set 373
for substring 373
searching
for character 362
for tokens 375
uppercasing 378
width, returning 388
strings, merging, literal 448
strlen (function) 367
strlwr (function) 368
strncat (function) 368
strncmp (function) 368
strncmpi (function) 369
strncpy (function) 370
strnicmp (function) 370
strnset (function) 371
stroked fonts 209, 334, 522
code, linked-in 294
strpbrk (function) 371
strrchr (function) 372
strrev (function) 372
strset (function) 372
strspn (function) 373
strstr (function) 373
strtod (function) 374
strtok (function) 375
strtol (function) 376
strtoul (function) 377
strupr (function) 378
style, fill 191
suffixes
exec... 106
spawn... 353
suppressing load operations 450
suspending execution 93, 348
swab (function) 378
symbolic constants 446
symbolic debugger 449
syntax
command-line
BGIOBJ 523, 526
CPP 462
GREP 515
MAKE 487
TLIB 509
TLINK 494
errors 423
tracking 562
explicit rule 470
implicit rule 473
sys_errlist (global variable) 24
sys_nerr (global variable) 24
sys\stat.h 9
sys\timeb.h 9
sys\types.h 9
system
date 43, 91, 98, 148, 188, 230, 395

returning 158
setting 320, 361
time 43, 91, 98, 148, 188, 230, 395
returning 183
setting 334, 361
system (function) 379

T

tab command (TC editor) 421
Tab mode 411
tab On/Off command (TC editor) 421
Tab size menu setting 421
Tab Size option, TCINST 564
tan (function) 379
tangent 379
tangent, hyperbolic 380
tanh (function) 380
target files 470
extension 473
MAKE command-line 488
task state 233
TASM 445, 446, 455
TC *See also* Turbo C integrated development environment
TC.EXE, customizing 555
TCC *See also* command-line Turbo C
TCC configuration file 462
TCC linker (TLINK) 493, 498
TCINST 555, 556
black and white option 556
color option 556
invoking 556
menu system 556
overridden by configuration file 556
overridden by TC menu system 556
technical support, Borland 4
tell (function) 380
template 251
termination
function 47
program 109, 110
Test Stack Overflow option, TCINST 560
text
attributes 381
background color, setting 381, 383
characteristics 332
copying
from one screen rectangle to another 253
to memory 180
to screen 283
entering in Edit window 410
fonts, graphics 191, 332
information on, returning 182
justifying 331
marked 416
mode 27, 87, 89, 118, 134, 142, 180, 283, 291, 325, 328, 386, *See* screen operating mode, *See* operating mode of screen
video information, returning 181
windows,defining 406
textattr (function) 381
textbackground (function) 383
textcolor (function) 384
textheight (function) 386
textmode (function) 386
textwidth (function) 388
time
elapsed
calculation of 77, 96
returning 77, 388
file 167, 323
system 43, 91, 98, 148, 188, 230, 395
returning 183
setting 334, 361
time (function) 388
time.h 9
timezone (global variable) 31
TLIB (librarian) 461, 508
action symbols 511
case sensitive flag 510, 514
command-line options 509
command-line syntax 509
examples 514
Extended Dictionary 508, 513
Extended Dictionary flag 510
library name 509
list file 510
module names 511

operation list 510
operations 511
order of 511
response files 513
TLINK (linker) 461
called by MAKE 467
case sensitivity 502
command-line syntax 494
error messages 504
executable file map generated by 499
Extended Dictionary 503
fatal errors 504, 505
file extensions supplied by 494
generating .COM files 503
invoking 493
linker for TCC 498
map file generated by 495
nonfatal errors 504, 507
options 499
response files 495
example 496
restrictions 504
used with Turbo C modules 496
warnings 504, 507
tmpfile (function) 389
tmpnam (function) 389
toascii (function) 390
toggles *See also* menu toggles
tokens 541
searching in string 375
tolower (function) 391
TOUCH utility 461, 493
toupper (function) 392
trailing segments, uninitialized 501
translation mode 87, 89
TSR program 225
Turbo Assembler 445
Turbo C
constructs
constants 542
declarations 548
expressions 545
external definitions 552
identifiers 542
keywords 542
lexical grammar 541
operators 545
phrase structure grammar 541, 545
preprocessing directives 541, 552
punctuators 545
statements 551
string literals 544
syntax 541
tokens 541
customization program (TCINST) 555
extension keywords 452
integrated development environment *See also* TC
library files 456
Turbo C Directory option, TCINST 566
TURBOC.CFG 458, 462
Turquoise color set menu, TCINST 573
tzname (global variable) 31
tzset (function) 392

U

ultoa (function) 394
unbuffered stream 317, 336
underscore 449
ungetc (function) 394
ungetch (function) 395
Unindent mode 411, 415, 564
unindent On/Off command (TC editor) 421
uninitialized data segment
class, renaming 454
group, renaming 454
renaming 454
UNIX, porting Turbo C files to 452
UNIX format, conversion to 405
unixtodos (function) 395
unlink (function) 396
unlock (function) 396
updating, file, by MAKE 470
Use Register Variables option, TCINST 560
Use Tabs option, TCINST 564

user-defined comparison function
284
user-defined fill pattern 164, 321, 322
user-loaded graphics driver code 293
user-modifiable math error handler
242
user-specified library files 457
user-specified signal handlers 341
utilities, standalone 461, *See* standalone utilities

V

va... (function) 397
values.h 9
variable argument list 397, 449
variables
global 22, 529
global time, setting 392
register 450
vectors, interrupt 92
8086 184, 338
returning 184
setting 338
vendor-added device driver 207
verify flag, disk 186, 338
vfprintf (function) 399
vfscanf (function) 400
VGA/
EGA Save Fonts option, TCINST
563
video information, text mode 181
viewport 76, 191, 325, 573
displaying string in 260, 261
returning information on 186
setting for graphics output 339
vprintf (function) 401
vscanf (function) 402
vsprintf (function) 403
vsscanf (function) 404

W

Warn Duplicate Symbols option, TCINST
562
warnings 423, 437
enabling and disabling 452
TC compiler 561
TLINK 504, 507
wherex (function) 405
wherey (function) 405
WILDARGS.OBJ 19
wildcards 417
expansion 19
by default 20
from integrated environment 20
used by CPP 462
wildcards, in OBJXREF command line
529
window (function) 406
windows
active 409
Edit 409
text mode, defining 406
zooming 563
word aligning 560
of integers 448
words
reading from hardware ports 205
writing to hardware ports 259
WordStar 409, 421
commands 409
Editor commands not in 421
working directory 107, 353
changing 71
returning 156, 157
write (function) 407
write access 38, 72, 87, 89, 147, 258,
360
write block command (TC editor) 417
write error 120

X

x aspect factor 151
x coordinate 187
maximum 173

Y

y aspect factor 151
y coordinate 188
maximum 173

Z

Zoomed Windows option, TCINST
563

Operator Precedence Table

Operator	Level	Name
[ ]	1	Array
( )		Function
->		Member
.		Member
!	2	Logical negation
~		Complement
++		Increment
--		Decrement
-		Arithmetic negation
(type)		Type cast
*		Indirection
&		Address of
sizeof		Size of object
*	3	Multiplication
/		Division
%		Remainder
+	4	Addition
-		Subtraction
<<	5	Left shift
>>		Right shift
<	6	Less than
<=		Less than or equal to
>		Greater than
>=		Greater than or equal to
==	7	Equal to
!=		Not equal
&	8	Bitwise AND
^	9	Bitwise XOR
|	10	Bitwise OR
&&	11	Logical AND
||	12	Logical OR
?:	13	Conditional
=	14	Assignment
,	15	Multiple expressions

Escape Sequences

\a	Bell
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal value
\xnn	Hex value

Color Table for Text Mode & EGA/VGA Graphics Mode

COLOR	VALUE
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

Imprimerie Hérissey - Évreux — N° 46145