Writing 8087 Reals

8087 *Reals* are written on a format slightly different from the standard format, as described below.

R The decimal representation of the value of *R* is output in a field 23 characters wide, using floating point format. For $R >= 0.0$, the format is:

```
␣␣#.################E*##
```

For $R < 0.0$, the format is:

```
␣-#.################E*##
```

where ␣ represents a blank, # represents a digit, and * represents either plus or minus.

R:n The decimal representation of the value of *R* is output, right adjusted in a field *n* characters wide, using floating point format. For $R >= 0.0$:

```
blanks#.digitsE*##
```

For $R < 0.0$:

```
blanks-#.digitsE*##
```

where *blanks* represents zero or more blanks, *digits* represents from 1 to 14 digits, # represents a digit, and * represents either plus or minus.

Internal Data Format

The 8087 chip supports a range of data types. The one used by TURBO-87 is the *long real*; its 64-bits yielding 16 digits accuracy and a range of 4.19E-307 to 1.67E + 308.

This 8-byte *Real* is not compatible with TURBO standard or BCD *Reals*. This, however, should only be a problem if you develop programs in different versions of TURBO which must interchange data. The trick then is simply to provide an interchange-format between the programs in which you transfer *Reals* on ASCII format, for instance.

Appendix A
SUMMARY OF STANDARD PROCEDURES AND FUNCTIONS

This appendix lists all standard procedures and functions available in TURBO Pascal and describes their application, syntax, parameters, and type. The following symbols are used to denote elements of various types:

type	any type
string	any string type
file	any file type
scalar	any scalar type
pointer	any pointer type

Where parameter type specification is not present, it means that the procedure or function accepts variable parameters of any type.

Input/Output Procedures and Functions

The following procedures use a non-standard syntax in their parameter lists:

procedure

Read (**var** *F*: **file of** *type*; **var** V: *type*);
Read (**var** *F*: text; **var** *I*: Integer);
Read (**var** *F*: text; **var** *R*: Real);
Read (**var** *F*: text; **var** *C*: Char);
Read (**var** *F*: text; **var** *S*: *string*);
Readln (**var** *F*: text);
Write (**var** *F*: **file of** *type*; **var** V: *type*);
Write (**var** *F*: text; *I*: Integer);
Write (**var** *F*: text; *R*: Real);
Write (**var** *F*: text; *B*: Boolean);
Write (**var** *F*: text; *C*: Char);
Write (**var** *F*: text; *S*: *string*);
Writeln (**var** *F*: text);

Arithmetic Functions

function

Abs (I: Integer): Integer;
Abs (R: Real): Real;
ArcTan (R: Real): Real;
Cos (R: Real): Real;
Exp (R. Real): Real;
Frac (R: Real): Real;
Int (R: Real): Real;
Ln (R: Real): Real;
Sin (R: Real): Real;
Sqr (I: Integer): Integer;
Sqr (R: Real): Real;
Sqrt (R: Real): Real;

Scalar Functions

function

Odd (I: Integer): Boolean;
Pred (X: scalar): scalar;
Succ (X: scalar): scalar;

Transfer Functions

function

Chr (I: Integer): Char;
Ord (X: scalar): Integer;
Round (R: Real): Integer;
Trunc (R: Real): Integer;

String Procedures and Functions

The *Str* procedure uses a non-standard syntax for its numeric parameter.

procedure

Delete (var S: string; Pos, Len: Integer);
Insert (S: string; var D: string; Pos: Integer);
Str (I: Integer; var S: string);
Str (R: Real; var S: string);
Val (S: string; var R: Real; var P: Integer);
Val (S: string; var I, P: Integer);

function

Concat (S1,S2,...,Sn: string): string;
Copy (S: string; Pos, Len: Integer): string;
Length (S: string): Integer;
Pos (Pattern, Source: string): Integer;

File Handling Routines

procedure

Append (var F: file; Name: String);
Assign (var F: file; Name: string);
BlockRead (var F: file; var Dest: Type; Num: Integer);
BlockWrite (var F: file; var Dest:Type; Num: Integer);
Chain (var F: file);
Close (var F: file);
Erase (var F: file);
Execute (var F: file);
Rename (var F: file; Name: string);
Reset (var F: file);
Rewrite (var F: file);
Seek (var F: file of type; Pos: Integer);

function

Eof (var F: file): Boolean;
Eoln (var F: Text): Boolean;
FilePos (var F: file of type): Integer;
FilePos (var F: file): Integer;
FileSize (var F: file of type): Integer;
FileSize (var F: file): Integer;
SeekEof (var F: file): Boolean;
SeekEoln (var F: Text): Boolean;

Heap Control Procedures and Functions

procedure

Dispose(var P: Pointer);
FreeMem(var P: Pointer, I: Integer);
GetMem (var P: pointer; I: Integer);
Mark (var P: pointer);
New (var P: pointer);
Release (var P: pointer);

function

MaxAvail: Integer;
MemAvail: Integer;
Ord (P: pointer): Integer;
Ptr (I: Integer): Pointer;

Screen Related Procedures and Functions

procedure

CrtExit;
CrtInit;
ClrEol;
ClrScr;
DelLine;
GotoXY (X, Y: Integer);
InsLine;
LowVideo;
NormVideo;

function

WhereX: Integer; (IBM PC only)
WhereY: Integer; (IBM PC only)

Miscellaneous Procedures and Functions

procedure

Bdos (Func,Param: Integer); (CP/M only)
Bios (Func,Param: Integer); (CP/M only)
ChDir (Path: String);
Delay (mS: Integer);
FillChar (var Dest, Length: Integer; Data: Char);
FillChar (var Dest, Length: Integer; Data: byte);
Halt;
GetDir (Drv:integer; var Path: String);
MkDir (Path: String);
MsDos (Func: Integer; Param: record); (PC/MS-DOS only)
Move (var Source,Dest:type; Length: Integer);
Randomize;
RmDir (Path: String);

function

Addr (var Variable): Pointer; (PC/MS-DOS, CP/M-86)
Addr (var Variable): Integer; (CP/M-80)
Addr (< function identifier >): Integer; (CP/M-80)
Addr (< procedure identifier >): Integer;(CP/M-80)
Bdos (Func, Param: Integer): Byte;
BdosHL (Func, Param: Integer): Integer;
Bios (Func, Param: Integer): Byte;
BiosHL (Func, Param: Integer): Integer;
Hi (I: Integer): Integer;
IOresult : Boolean;
KeyPressed : Boolean;
Lo (I: Integer): Integer;
ParamCount : Integer;
ParamStr (N: Integer): String
ParamCount: Integer;
ParamStr (N: Integer): String;
Random (Range: Integer): Integer;
Random : Real;
SizeOf (var Variable): Integer;
SizeOf (< type identifier >): Integer;
Swap (I: Integer): Integer;
UpCase (Ch: Char): Char;

IBM PC Procedures and Functions

The following procedures and functions apply to the IBM PC implementations only.

Basic Graphics, Windows, and Sound

procedure

```
Draw(X1,Y1,X2,Y2,Color);
GraphBackground(Color:Integer);
GraphColorMode;
GraphMode;
GraphWindow(X1,Y1,X2,Y2:Integer);
HiRes;
HiResColor(Color:Integer);
NoSound;
Palette(Color:Integer);
Plot(X,Y,Color:Integer);
Sound(I: Integer);
TextBackground(Color:Integer);
TextColor(Color:Integer);
TextMode(Color:Integer);
Window(X1,Y1,X2,Y2:Integer);
```

function

```
WhereX:Integer;
WhereY:Integer;
```

constant

```
BW40:Integer;        = 0
C40:Integer;         = 1
BW80:Integer;        = 2
C80:Integer;         = 3
Black:Integer;       = 0
Blue:Integer;        = 1
Green:Integer;       = 2
Cyan:Integer;        = 3
Red:Integer;         = 4
Magenta:Integer;     = 5
Brown:Integer;       = 6
LightGray:Integer;   = 7
DarkGray:Integer;    = 8
LightBlue:Integer;   = 9
LightGreen:Integer;  = 10
```



```
LightCyan:Integer;   = 11
LightRed:Integer;    = 12
LightMagenta:Integer; = 13
Yellow:Integer;      = 14
White:Integer;       = 15
Blink:Integer;       = 16
```

Extended Graphics

procedure

```
Arc(X,Y,Angle,Radius,Color: Integer);
Circle(X,Y,Radius,Color: Integer);
ColorTable(C1,C2,C3,C4: Integer);
FillScreen(Color: Integer);
FillShape(X,Y,FillColor,BorderColor: Integer);
FillPattern(X1,Y1,X2,Y2,Color: Integer);
GetPic(var Buffer: AnyType;X1,Y1,X2,Y2: Integer);
Pattern(P: array[0..7] of Byte);
PutPic(var Buffer: AnyType;X,Y: Integer);
```

function

```
GetDotColor(X,Y: Integer): Integer;
```

Turtlegraphics

procedure

```
Back(Dist: Integer);
ClearScreen;
Forward(Dist: Integer);
HideTurtle;
Home;
NoWrap;
PenDown;
PenUp;
SetHeading(Angle: Integer);
SetPenColor(Color: Integer);
SetPosition(X,Y: Integer);
ShowTurtle;
TurnLeft(Angle: Integer);
TurnRight(Angle: Integer);
TurtleWindow(X,Y,W,H: Integer);
Wrap;
```

```
function
    Heading: Integer;
    Xcor: Integer;
    Ycor: Integer;
    TurtleThere: Boolean;

constant
    North:Integer constant      = 0
    East:Integer constant       = 90
    South:Integer constant      = 180
    West:Integer constant       = 270
```


Appendix B
SUMMARY OF OPERATORS

The following table summarizes all operators of TURBO Pascal. The operators are grouped in order of descending precedence. Where *Type of operand* is indicated as *Integer, Real*, the result is as follows:

Operand	**Result**
Integer, Integer	Integer
Real, Real	Real
Real, Integer	Real

Operator	**Operation**	**Type of operand(s)**	**Type of result**
+ unary	sign identity	Integer, Real	as operand
- unary	sign inversion	Integer, Real	as operand
not	negation	Integer, Boolean	as operand
*	multiplication	Integer, Real	Integer, Real
	set intersection	any set type	as operand
/	division	Integer, Real	Real
div	Integer division	Integer	Integer
mod	modulus	Integer	Integer
and	arithmetical **and**	Integer	Integer
	logical **and**	Boolean	Boolean
shl	shift left	Integer	Integer
shr	shift right	Integer	Integer
+	addition	Integer, Real	Integer, Real
	concatenation	string	string
	set union	any set type	as operand
-	subtraction	Integer, Real	Integer, Real
	set difference	any set type	as operand
or	arithmetical **or**	Integer	Integer
	logical **or**	Boolean	Boolean
xor	arithmetical **xor**	Integer	Integer
	logical **xor**	Boolean	Boolean

Operator	Operation	Type of operand(s)	Type of result
=	equality	any scalar type	Boolean
	equality	string	Boolean
	equality	any set type	Boolean
	equality	any pointer type	Boolean
<>	inequality	any scalar type	Boolean
	inequality	string	Boolean
	inequality	any set type	Boolean
	inequality	any pointer type	Boolean
>=	greater or equal	any scalar type	Boolean
	greater or equal	string	Boolean
	set inclusion	any set type	Boolean
<=	less or equal	any scalar type	Boolean
	less or equal	string	Boolean
	set inclusion	any set type	Boolean
>	greater than	any scalar type	Boolean
	greater than	string	Boolean
<	less than	any scalar type	Boolean
	less than	string	Boolean
in	set membership	see below	Boolean

The first operand of the **in** operator may be of any scalar type, and the second operand must be a set of that type.

Appendix C
SUMMARY OF COMPILER DIRECTIVES

A number of features of the TURBO Pascal compiler are controlled through compiler directives. A compiler directive is introduced as a comment with a special syntax which means that whenever a comment is allowed in a program, a compiler directive is also allowed.

A compiler directive consists of an opening bracket immediately followed by a dollar-sign immediately followed by one compiler directive letter or a list of compiler directive letters separated by commas, ultimately terminated by a closing bracket.

Examples:

```
{$I-}
{$I INCLUDE.FIL}
{$B-,R+,V-}
(*$U+*)
```

Notice that no spaces are allowed before or after the dollar-sign. A + sign after a directive indicates that the associated compiler feature is enabled (active), and a minus sign indicates that is disabled (passive).

IMPORTANT NOTICE

All compiler directives have default values. These have been chosen to optimize execution speed and minimize code size. This means that e.g. code generation for recursive procedures (CP/M-80 only) and index checking has been disabled. Check below to make sure that your programs include the required compiler directive settings!

Common Compiler Directives

B - *I/O Mode Selection*

Default: B+

The **B** directive controls input/output mode selection. When active, {$B+}, the CON: device is assigned to the standard files *Input* and *Output*, i.e. the default input/output channel. When passive, {$B-}, the TRM: device is used. **This directive is global to an entire program block** and cannot be re-defined throughout the program. See pages 105 and 108 for further details.

C - *Control C and S*

Default: C+

The **C** directive controls control character interpretation during console I/O. When active, {$C+}, a Ctrl-C entered in response to a *Read* or *Readln* statement will interrupt program execution, and a Ctrl-S will toggle screen output off and on. When passive, {$C-}, control characters are not interpreted. The active state slows screen output somewhat, so if screen output speed is imperative, you should switch off this directive. **This directive is global to an entire program block** and cannot be re-defined throughout the program.

I - *I/O Error Handling*

Default: I+

The **I** directive controls I/O error handling. When active, {$I+}, all I/O operations are checked for errors. When passive, {$I-}, it is the responsibility of the programmer to check I/O errors through the standard function *IOresult*. See page 116 for further details.

I - *Include Files*

The **I** directive succeeded by a file name instructs the compiler to include the file with the specified name in the compilation. Include files are discussed in detail in chapter 17.

R - *Index Range Check*

Default: R-

The **R** directive controls run-time index checks. When active, {$R+}, all array indexing operations are checked to be within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. When passive, {$R-}, no checks are performed, and index errors may well cause a program to go haywire. It is a good idea to activate this directive while developing a program. Once debugged, execution will be speeded up by setting it passive (the default state).

V - *Var-parameter Type Checking*

Default: V+

The **V** compiler directive controls type checking on strings passed as **var**-parameters. When active, {$V+}, strict type checking is performed, i.e. the lengths of actual and formal parameters must match. When passive, {$V-}, the compiler allows passing of actual parameters which do not match the length of the formal parameter. See pages 203, 236, and 267 for further details.

U - *User Interrupt*

Default: U-

The **U** directive controls user interrupts. When active, {$U+}, the user may interrupt the program anytime during execution by entering a Ctrl-C. When passive, {$U-}, this has no effect. Activating this directive will significantly slow down execution speed.

PC-DOS and MS-DOS Compiler Directives

The following directives are unique to the PC/MS-DOS implementations:

G - *Input File Buffer*

Default: G0

The **G** (get) directive enables I/O re-direction by defining the standard *Input* file buffer. When the buffer size is zero (default), the *Input* file refers to the *CON:* or *TRM:* device. When non-zero (e.g. {$G256}), it refers to the MS-DOS standard input handle.

The **D** compiler directive applies to such non-zero-buffer input and output files. The **G** compiler directive must be placed before the declaration part.

P - *Output File Buffer*

Default: P0

The **P** (put) directive enables I/O re-direction by defining the standard *Output* file buffer. When the buffer size is zero (default), the *Output* file refers to the *CON:* or *TRM:* device. When non-zero (e.g. {$G512}), it refers to the MS-DOS standard output handle.

The **D** compiler directive applies to such non-zero-buffer input and output files. The **P** compiler directive must be placed before the declaration part.

D - *Device Checking*

Default: D+

When a text file is opened by *Reset*, *Rewrite* or *Append*, TURBO Pascal asks MS-DOS for the status of the file. If MS-DOS reports that the file is a device, TURBO Pascal disables the buffering that normally occurs on text files, and all I/O operations on the file are done on a character by character basis.

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

F - *Number of Open Files*

Default: F16

The **F** directive controls the number of files that may be open simultaneously. The default setting is {$F16}, which means that up to 16 files may be open at any one time. This directive is global to a program and must be placed before the declaration part. The **F** compiler directive does not limit the number of files that may be declared in a program; it only sets a limit to the number of files that may be open at the same time.

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

PC-DOS, MS-DOS, and CP/M-86 Compiler Directive

The following directive is unique to the 16-bit implementations:

K - *Stack Checking*

Default: K+

The **K** directive controls the generation of stack check code. When active, {$K+}, a check is made to insure that space is available for local variables on the stack before each call to a subprogram. When passive, {$K-}, no checks are made.

CP/M-80 Compiler Directives

The following directives are unique to the 8-bit implementation:

A - *Absolute Code*

Default: A+

The **A** directive controls generation of absolute, i.e. non-recursive, code. When active, {$A+}, absolute code is generated. When passive, {$A-}, the compiler generates code which allows recursive calls. This code requires more memory and executes slower.

W - *Nesting of With Statements*

Default: W2

The **W** directive controls the level of nesting of *With* statements, i.e. the number of records which may be 'opened' within one block. The **W** must be immediately followed by a digit between 1 and 9. For further details, please refer to page 81.

X - *Array Optimization*

Default: X+

The **X** directive controls array optimization. When active, {$X+}, code generation for arrays is optimized for maximum speed. When passive, {$X-}, the compiler minimizes the code size instead. This is discussed further on page 75.

Appendix D
TURBO VS. STANDARD PASCAL

The TURBO Pascal language follows the Standard Pascal defined by Jensen & Wirth in their **User Manual and Report**, with only minor differences introduced for the sheer purpose of efficiency. These differences are described in the following. Notice that the *extensions* offered by TURBO Pascal are not discussed.

Dynamic Variables

The procedure *New* will not accept variant record specifications. This restriction, however, is easily circumvented by using the standard procedure *GetMem*.

Recursion

CP/M-80 version only: Because of the way local variables are handled during recursion, a variable local to a subprogram must not be passed as a **var**-parameter in recursive calls.

Get and Put

The standard procedures *Get* and *Put* are not implemented. Instead, the *Read* and *Write* procedures have been extended to handle all I/O needs. The reason for this is threefold: Firstly, *Read* and *Write* give much faster I/O; secondly, variable space overhead is reduced, as file buffer variables are not required, and thirdly, the *Read* and *Write* procedures are far more versatile and easier to understand that *Get* and *Put*.

Goto Statements

A **goto** statement must not leave the current block.

Page Procedure

The standard procedure *Page* is not implemented, as the CP/M operating system does not define a form-feed character.

Packed Variables

The reserved word **packed** has no effect in TURBO Pascal, but it is still allowed. This is because packing occurs automatically whenever possible. For the same reason, standard procedures *Pack* and *Unpack* are not implemented.

Procedural Parameters

Procedures and functions cannot be passed as parameters.

Appendix E
COMPILER ERROR MESSAGES

The following is a listing of error messages you may get from the compiler. When encountering an error, the compiler will always print the error number on the screen. Explanatory texts will only be issued if you have included error messages (answer **Y** to the first question when you start TURBO).

Many error messages are totally self-explanatory, but some need a little elaboration as provided in the following.

01 **';' expected**
02 **':' expected**
03 **',' expected**
04 **'(' expected**
05 **')' expected**
06 **' = ' expected**
07 **': = ' expected**
08 **'[' expected**
09 **']' expected**
10 **'.' expected**
11 **'..' expected**
12 **BEGIN expected**
13 **DO expected**
14 **END expected**
15 **OF expected**
16 **PROCEDURE or FUNCTION expected**
17 **THEN expected**
18 **TO or DOWNTO expected**
20 **Boolean expression expected**
21 **File variable expected**
22 **Integer constant expected**
23 **Integer expression expected**
24 **Integer variable expected**
25 **Integer or real constant expected**
26 **Integer or real expression expected**
27 **Integer or real variable expected**
28 **Pointer variable expected**
29 **Record variable expected**

30 Simple type expected
Simple types are all scalar types, except real.
31 Simple expression expected
32 String constant expected
33 String expression expected
34 String variable expected
35 Textfile expected
36 Type identifier expected
37 Untyped file expected
40 Undefined label
A statement references an undefined label.
41 Unknown identifier or syntax error
Unknown label, constant, type, variable, or field identifier, or syntax error in statement.
42 Undefined pointer type in preceding type definitions
A preceding pointer type definition contains a reference to an unknown type identifier.
43 Duplicate identifier or label
This identifier or label has already been used within the current block.
44 Type mismatch
1) Incompatible types of the variable and the expression in an assignment statement **2)** Incompatible types of the actual and the formal parameter in a call to a subprogram. **3)** Expression type incompatible with index type in array assignment. **4)** Types of operands in an expression are not compatible.
45 Constant out of range
46 Constant and CASE selector type does not match
47 Operand type(s) does not match operator
Example: 'A' div '2'
48 Invalid result type
Valid types are all scalar types, string types, and pointer types.
49 Invalid string length
The length of a string must be in the range 1..255.
50 String constant length does not match type
51 Invalid subrange base type
Valid base types are all scalar types, except real.
52 Lower bound > upper bound
The ordinal value of the upper bound must be greater than or equal to the ordinal value of the lower bound.
53 Reserved word
These may not be used as identifiers.
54 Illegal assignment

55 String constant exceeds line
String constants must not span lines.
56 Error in integer constant
An *Integer* constant does not conform to the syntax described in page 43, or it is not within the *Integer* range -32768..32767. Whole *Real* numbers should be followed by a decimal point and a zero, e.g. 123456789.0.
57 Error in real constant
The syntax of *Real* constants is defined on page 43.
58 Illegal character in identifier
60 Constants are not allowed here
61 Files and pointers are not allowed here
62 Structured variables are not allowed here
63 Textfiles are not allowed here
64 Textfiles and untyped files are not allowed here
65 Untyped files are not allowed here
66 I/O not allowed here
Variables of this type cannot be input or output.
67 Files must be VAR parameters
68 File components may not be files
file of file constructs are not allowed.
69 Invalid ordering of fields
70 Set base type out of range
The base type of a set must be a scalar with no more than 256 possible values or a subrange with bounds in the range 0..255.
71 Invalid GOTO
A GOTO cannot reference a label within a FOR loop from outside that FOR loop.
72 Label not within current block
A GOTO statement cannot reference a label outside the current block.
73 Undefined FORWARD procedure(s)
A subprogram has been **forward** declared, but the body never occurred.
74 INLINE error
75 Illegal use of ABSOLUTE
1) Only one identifier may appear before the colon in an **absolute** variable declaration. **2)** The **absolute** clause may not be used in a record.
76 Overlays can not be forwarded
The FORWARD specification cannot not be used in connection with overlays.
77 Overlays not allowed in direct mode
Overlays can only be used from programs compiled to a file.

90 File not found
The specified include file does not exist.

91 Unexpected end of source
Your program cannot end the way it does. The program probably has more **begin**s than **end**s.

92 Unable to create overlay file

93 Invalid compiler directive

97 Too many nested WITHs
Use the W compiler directive to increase the maximum number of nested WITH statements. Default is 2. (CP/M-80 only).

98 Memory overflow
You are trying to allocate more storage for variables than is available.

99 Compiler overflow
There is not enough memory to compile the program. This error may occur even if free memory seems to exist; it is, however, used by the stack and the symbol table during compilation. Break your source text into smaller segments and use include files.

Appendix F.
RUN-TIME ERROR MESSAGES

Fatal errors at run-time result in a program halt and the display of the message:

```
Run-time error NN, PC=addr
   Program aborted
```

where *NN* is the run-time error number, and *addr* is the address in the program code where the error occurred. The following contains explanations of all run-time error numbers. Notice that the numbers are hexadecimal!

01 Floating point overflow.

02 Division by zero attempted.

03 Sqrt argument error.
The argument passed to the Sqrt function was negative.

04 Ln argument error.
The argument passed to the Ln function was zero or negative.

10 String length error.
1) A string concatenation resulted in a string of more than 255 characters. **2)** Only strings of length 1 can be converted to a character.

11 Invalid string index.
Index expression is not within 1..255 with *Copy*, *Delete* or *Insert* procedure calls.

90 Index out of range.
The index expression of an array subscript was out of range.

91 Scalar or subrange out of range.
The value assigned to a scalar or a subrange variable was out of range.

92 Out of integer range.
The real value passed to *Trunc* or *Round* was not within the *Integer* range −32768..32767.

F0 Overlay file not found.

FF Heap/stack collision.
A call was made to the standard procedure *New* or to a recursive subprogram, and there is insufficient free memory between the heap pointer (HeapPtr) and the recursion stack pointer (RecurPtr).

Notes:

Appendix G
I/O ERROR MESSAGES

An error in an input or output operation at run-time results in in I/O error. If I/O checking is active (I compiler directive active), an I/O error causes the program to halt and the following error message is displayed:

```
I/O error NN, PC=addr
Program aborted
```

Where *NN* is the I/O error number, and *addr* is the address in the program code where the error occurred.

If I/O error checking is passive ({$I-}), an I/O error will not cause the program to halt. Instead, all further I/O is suspended until the result of the I/O operation has been examined with the standard function *IOresult*. If I/O is attempted before *IOresult* is called after en error, a new error occurs, possibly hanging the program.

The following contains explanations of all run-time error numbers. Notice that the numbers are hexadecimal!

01 **File does not exist.**
The file name used with *Reset, Erase, Rename, Execute,* or *Chain* does not specify an existing file.

02 **File not open for input.**
1) You are trying to read (with *Read* or *Readln*) from a file without a previous *Reset* or *Rewrite*. **2)** You are trying to read from a text file which was prepared with *Rewrite* (and thus is empty). **3)** You are trying to read from the logical device LST:, which is an output-only device.

03 **File not open for output.**
1) You are trying to write (with *Write* or *Writeln*) to a file without a previous *Reset* or *Rewrite*. **2)** You are trying to read from a text file which was prepared with *Reset*. **3)** You are trying to read from the logical device KBD:, which is an input-only device.

04 **File not open.**
You are trying to access (with *BlockRead* or *BlockWrite*) a file without a previous *Reset* or *Rewrite*.

10 **Error in numeric format.**
The string read from a text file into a numeric variable does not conform to the proper numeric format (see page 43).

20 **Operation not allowed on a logical device.**
You are trying to *Erase, Rename, Execute,* or *Chain* a file assigned to a logical device.

21 **Not allowed in direct mode.**
Programs cannot be *Executed* or *Chained* from a program running in direct mode (i.e. a program activated with a **R**un command while the **M**emory compiler option is set).

22 **Assign to std files not allowed.**

90 **Record length mismatch.**
The record length of a file variable does not match the file you are trying to associate it with.

91 **Seek beyond end-of-file.**

99 **Unexpected end-of-file.**
1) Physical end-of-file encountered before EOF-character (Ctrl-Z) when reading from a text file. **2)** An attempt was made to read beyond end-of-file on a defined file. **3)** A *Read* or *BlockRead* is unable to read the next sector of a defined file. Something may be wrong with the file, or (in the case of *BlockRead*) you may be trying to read past physical EOF.

F0 **Disk write error.**
Disk full while attempting to expand a file. This may occur with the output operations *Write, WriteLn, BlockWrite,* and *Flush*, but also *Read, ReadLn,* and *Close* may cause this error, as they cause the write buffer to be flushed.

F1 **Directory is full.**
You are trying to *Rewrite* a file, and there is no more room in the disk directory.

F2 **File size overflow.**
You are trying to *Write* a record beyond 65535 to a defined file.

F3 **Too many open files.**

FF **File disappeared.**
An attempt was made to *Close* a file which was no longer present in the disk directory, e.g. because of an unexpected disk change.

Appendix H
TRANSLATING ERROR MESSAGES

The compiler error messages are collected in the file *TURBO.MSG*. These messages are in English but may easily be translated into any other language as described in the following.

The first 24 lines of this file define a number of text constants for subsequent inclusion in the error message lines; a technique which drastically reduces the disk and memory requirements of the error messages. Each constant is identified by a control character, denoted by a ^ character in the following listing. The value of each constant is anything that follows on the same line. All characters are significant, also leading and trailing blanks.

The remaining lines each contain one error message, starting with the error number and immediately followed by the message text. The message text may consist of any characters and may include previously defined constant identifiers (control characters). Appendix E lists the resulting messages in full.

When you translate the error messages, the relation between constants and error messages will probably be quite different from the English version listed here. Start therefore with writing each error message in full, disregarding the use of constants. You may use these error messages, but they will require excessive space. When all messages are translated, you should find as many common denominators as possible. Then define these as constants at the top of the file and include only the constant identifiers in subsequent message texts. You may define as few or as many constants as you need, the restriction being only the number of control characters.

As a good example of the use of constants, consider errors 25, 26, and 27. These are defined exclusively by constant identifiers, 15 in total, but would require 101 characters if written in clear text.

The TURBO editor may be used to edit the *TURBOMSG.OVR* file. Control characters are entered with the Ctrl-P prefix, i.e. to enter a Ctrl-A (^A) into the file, hold down the <CONTROL> key and press first P, then A. Control characters appear dim on the screen (if it has any video attributes).

Notice that the TURBO editor deletes all trailing blanks. The original message therefore does not use trailing blanks in any messages.

Error Message File Listing

```
^A are not allowed
^B can not be
^C constant
^D does not
^E expression
^F identifier
^G file
^H here
^KInteger
^LFile
^NIllegal
^O or
^PUndefined
^Q match
^R real
^SString
^TTextfile
^U out of range
^V variable
^W overflow
^X expected
^Y type
^[Invalid
^] pointer
01';'^X
02':'^X
03',^X
04'('^X
05')'^X
06'='^X
07':='^X
08'['^X
09']'^X
10'.'^X
11'..'^X
12BEGIN^X
13DO^X
14END^X
```



```
15OF^X
17THEN^X
18TO^O DOWNTO^X
20Boolean^E^X
21^L^V^X
22^K^C^X
23^K^E^X
24^K^V^X
25^K^O^R^C^X
26^K^O^R^E^X
27^K^O^R^V^X
28Pointer^V^X
29Record^V^X
30Simple^Y^X
31Simple^E^X
32^S^C^X
33^S^E^X
34^S^V^X
35^T^X
36Type^F^X
37Untyped^G^X
40^P label
41Unknown^F^O syntax error
42^P^]^Y in preceding^Y definitions
43Duplicate^F^O label
44Type mismatch
45^C^U
46^C and CASE selector^Y^D^Q
47Operand^Y(s)^D^Q operator
48^[ result^Y
49^[ ^S length
50^S^C length^D^Q^Y
51^[ subrange base^Y
52Lower bound > upper bound
53Reserved word
54^N assignment
55^S^C exceeds line
56Error in integer^C
57Error in^R^C
58^N character in^F
60^Cs^A^H
61^Ls and^]s^A^H
62Structured^Vs^A^H
63^Ts^A^H
```

```
64^Ts and untyped^Gs^A^H
65Untyped^Gs^A^H
66I/O^A
67^Ls must be^V parameters
68^L components^B^Gs
69^[^Ordering of fields
70Set base^Y^U
71^[ GOTO
72Label not within current block
73^P FORWARD procedure(s)
74INLINE error
75^N use of ABSOLUTE
90^L not found
91Unexpected end of source
97Too many nested WITH's
98Memory^W
99Compiler^W
```


Appendix I
TURBO SYNTAX

The syntax of the TURBO Pascal language is presented here using the formalism known as the *Backus-Naur Form*. The following symbols are meta-symbols belonging to the BNF formalism, and not symbols of the TURBO Pascal language:

::=	Means "is defined as".
|	Means "or".
{ }	Enclose items which may be repeated zero or more times.

All other symbols are part of the language. Each syntactic construct is printed in *italics*, e.g.: *block* and *case-element*. reserved words are printed in **boldface**, e.g.: **array** and **for**.

actual-parameter ::= *expression* | *variable*
adding-operator ::= + | - | **or** | **xor**
array-constant ::= (*structured-constant* { , *structured-constant* })
array-type ::= **array** [*index-type* { , *index-type* }] **of** *component-type*
array-variable ::= *variable*
assignment-statement ::= *variable* := *expression* |
function-identifier ::= *expression*
base-type ::= *simple-type*
block ::= *declaration-part statement-part*
case-element ::= *case-list* : *statement*
case-label ::= *constant*
case-label-list ::= *case-label* { , *case-label* }
case-list ::= *case-list-element* { , *case-list-element*}
case-list-element ::= *constant* | *constant* .. *constant*
case-statement ::= **case** *expression* **of** *case-element* { ; *case-element* } **end** |
case *expression* **of** *case-element* { ;*case-element* }
otherwise *statement* { ; *statement* } **end**
complemented-factor ::= *signed-factor* | **not** *signed-factor*
component-type ::= *type*
component-variable ::= *indexed-variable* | *field-designator*
compound-statement ::= **begin** *statement* { ; *statement* } **end**
conditional-statement ::= *if-statement* | *case-statement*

constant :: = *unsigned-number* | *sign unsigned-number* | *constant-identifier*
| *sign constant-identifier* | *string*
constant-definition-part :: = **const** *constant-definition*
{ ; *constant-definition* } ;
constant-definition :: = *untyped-constant-definition* |
typed-constant-definition
constant-identifier :: = *identifier*
control-character :: = # *unsigned-integer* | ^ *character*
control-variable :: = *variable-identifier*
declaration-part :: = { *declaration-section* }
declaration-section :: = *label-declaration-part* | *constant-definition-part* |
type-definition-part | *variable-declaration-part* |
procedure-and-function-declaration-part
digit :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit-sequence :: = *digit* { *digit* }
empty :: =
empty-statement :: = *empty*
entire-variable :: = *variable-identifier* | *typed-constant-identifier*
expression :: = *simple-expression* { *relational-operator simple-expression* }
factor :: = *variable* | *unsigned-constant* | (*expression*) |
function-designator | *set*
field-designator :: = *record-variable* . *field-identifier*
field-identifier :: = *identifier*
field-list :: = *fixed-part* | *fixed-part* ; *variant-part* | *variant-part*
file-identifier :: = *identifier*
file-identifier-list :: = *empty* | (*file-identifier* { , *file-identifier* }
file-type :: = **file of** *type*
final-value :: = *expression*
fixed-part :: = *record-section* { ; *record-section* }
for-list :: = *initial-value* **to** *final-value* | *initial-value* **downto** *final-value*
for-statement :: = **for** *control-variable* : = *for-list* **do** *statement*
formal-parameter-section :: = *parameter-group* | **var** *parameter-group*
function-declaration :: = *function-heading block* ;
function-designator :: = *function-identifier* | *function-identifier*
(*actual-parameter* { , *actual-parameter* })
function-heading :: = **function** *identifier* : *result-type* ; |
function *identifier* (*formal-parameter-section*
{ , *formal-parameter-section* }) : *result-type* ;
function-identifier :: = *identifier*
goto-statement :: = **goto** *label*
hexdigit :: = *digit* | A | B | C | D | E | F
hexdigit-sequence :: = *hexdigit* { *hexdigit* }
identifier :: = letter { *letter-or-digit* }
identifier-list :: = *identifier* { , *identifier* }

if-statement :: = **if** *expression* **then** *statement* { **else** *statement* }
index-type :: = *simple-type*
indexed-variable :: = *array-variable* [*expression* { , *expression* }]
initial-value :: = *expression*
inline-list-element :: = *unsigned-integer* | *constant-identifier* |
variable-identifier | *location-counter-reference*
inline-statement :: = **inline** *inline-list-element* { , *inline-list-element* }
label :: = *letter-or-digit* { *letter-or-digit* }
label-declaration-part :: = **label** *label* { , *label* } ;
letter :: = 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 | _
letter-or-digit :: = *letter* | *digit*
location-counter-reference :: = * | * *sign constant*
multiplying-operator :: = * | / | **div** | **mod** | **and** | **shl** | **shr**
parameter-group :: = *identifier-list* : *type-identifier*
pointer-type :: = ^ *type-identifier*
pointer-variable :: = *variable*
procedure-and-function-declaration-part :: =
{ *procedure-or-function-declaration* }
procedure-declaration :: = *procedure-heading block* ;
procedure-heading :: = **procedure** *identifier* ; | **procedure** *identifier*
(*formal-parameter-section*
{ , *formal-parameter-section* }) ;
procedure-or-function-declaration :: = *procedure-declaration* |
function-declaration
procedure-statement :: = *procedure-identifier* | *procedure-identifier*
(*actual-parameter* { , *actual-parameter* })
program-heading :: = *empty* | **program** *program-identifier*
file-identifier-list
program :: = *program-heading block* .
program-identifier :: = *identifier*
record-constant :: = (*record-constant-element*
{ ; *record-constant-element* })
record-constant-element :: = *field-identifier* : *structured-constant*
record-section :: = *empty* | *field-identifier* { , *field-identifier* } : *type*
record-type :: = **record** *field-list* **end**
record-variable :: = *variable*
record-variable-list :: = *record-variable* { , *record-variable* }
referenced-variable :: = *pointer-variable* ^
relational-operator :: = = | < > | < = | > = | < | > | **in**
repeat-statement :: = **repeat** *statement* { ; *statement* } **until** *expression*
repetitive-statement :: = *while-statement* | *repeat-statement* | *for-statement*

result-type :: = *type-identifier*
scalar-type :: = (*identifier* { , *identifier* })
scale-factor :: = *digit-sequence* | *sign digit-sequence*
set :: = [{ *set-element* }]
set-constant :: = [{ *set-constant-element* }]
set-constant-element :: = *constant* | *constant* .. *constant*
set-element :: = *expression* | *expression* .. *expression*
set-type :: = **set of** *base-type*
sign :: = + | -
signed-factor :: = *factor* | *sign factor*
simple-expression :: = *term* { *adding-operator term* }
simple-statement :: = *assignment-statement* | *procedure-statement* |
goto-statement | *inline-statement* | *empty-statement*
simple-type :: = *scalar-type* | *subrange-type* | *type-identifier*
statement :: = *simple-statement* | *structured-statement*
statement-part :: = *compound-statement*
string :: = { *string-element* }
string-element :: = *text-string* | *control-character*
string-type :: = **string** [*constant*]
structured-constant :: = *constant* | *array-constant* | *record-constant* |
set-constant
structured-constant-definition :: = *identifier* : *type* = *structured-constant*
structured-statement :: = *compound-statement* | *conditional-statement* |
repetitive-statement | *with-statement*
structured-type :: = *unpacked-structured-type* |
packed *unpacked-structured-type*
subrange-type :: = *constant* .. *constant*
tag-field :: = *empty* | *field-identifier* :
term :: = *complemented-factor* { *multiplying-operator complemented-factor* }
text-string :: = ' { *character* } '
type-definition :: = *identifier* = *type*
type-definition-part :: = **type** *type-definition* { ; *type-definition* } ;
type-identifier :: = *identifier*
type :: = *simple-type* | *structured-type* | *pointer-type*
typed-constant-identifier :: = *identifier*
unpacked-structured-type :: = *string-type* | *array-type* | *record-type* |
set-type | *file-type*
unsigned-constant :: = *unsigned-number* | *string* | *constant-identifier* | **nil**
unsigned-integer :: = *digit-sequence* | $ *hexdigit-sequence*
unsigned-number :: = *unsigned-integer* | *unsigned-real*
unsigned-real :: = *digit-sequence* . *digit-sequence* |
digit-sequence . *digit-sequence* E *scale-factor* |
digit-sequence E *scale-factor*
untyped-constant-definition :: = *identifier* = *constant*

variable :: = *entire-variable* | *component-variable* | *referenced-variable*
variable-declaration :: = *identifier-list* : *type* |
identifier-list : *type* **absolute** *constant*
variable-declaration-part :: = **var** *variable-declaration*
{ ; *variable-declaration* } ;
variable-identifier :: = *identifier*
variant :: = *empty* | *case-label list* : (*field-list*)
variant-part :: = **case** *tag-field type-identifier* **of** *variant* { ; *variant* }
while-statement :: = **while** *expression* **do** *statement*
with-statement :: = **with** *record-variable-list* **do** *statement*

Notes:

Appendix J
ASCII TABLE

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	^@ NUL	32	20	SPC	64	40	@	96	60	`
1	01	^A SOH	33	21	!	65	41	A	97	61	a
2	02	^B STX	34	22	"	66	42	B	98	62	b
3	03	^C ETX	35	23	#	67	43	C	99	63	c
4	04	^D EOT	36	24	$	68	44	D	100	64	d
5	05	^E ENQ	37	25	%	69	45	E	101	65	e
6	06	^F ACK	38	26	&	70	46	F	102	66	f
7	07	^G BEL	39	27	'	71	47	G	103	67	g
8	08	^H BS	40	28	(	72	48	H	104	68	h
9	09	^I HT	41	29	)	73	49	I	105	69	i
10	0A	^J LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P DLE	48	30	0	80	50	P	112	70	p
17	11	^Q DC1	49	31	1	81	51	Q	113	71	q
18	12	^R DC2	50	32	2	82	52	R	114	72	r
19	13	^S DC3	51	33	3	83	53	S	115	73	s
20	14	^T DC4	52	34	4	84	54	T	116	74	t
21	15	^U NAK	53	35	5	85	55	U	117	75	u
22	16	^V SYN	54	36	6	86	56	V	118	76	v
23	17	^W ETB	55	37	7	87	57	W	119	77	w
24	18	^X CAN	56	38	8	88	58	X	120	78	x
25	19	^Y EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^] GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_ US	63	3F	?	95	5F	_	127	7F	DEL

Notes:

Appendix K
KEYBOARD RETURN CODES

This appendix lists the codes returned from all combinations of keys on the entire IBM PC keyboard, as they are seen by TURBO Pascal. Actually, function keys and 'Alt-ed' keys generate 'extended scan codes', but these are turned into 'escape sequences' by TURBO.

To read the escape sequences, you let your read routine check for ESC, and if detected see if there is another character in the keyboard buffer. If there is, an escape code was received, so you read the next character and set a flag to signal that what you got is not a normal character, but the second part of an 'escape sequence'

```
if KeyPressed then
begin
  Read(Kbd,Ch)                          { ch is char     }
  if (ch = #27) and KeyPressed then { one more char? }
  begin
    Read(Kbd,Ch)
    FuncKey := True;              { FuncKey is boolean }
  end
end;
```

The following table lists the return codes as decimal ASCII values. Normal keys only return a single code; extended codes return an ESC (27) followed by one more character.

Key	Unshifted	Shift	Ctrl	Alt
F1	27 59	27 84	27 94	27 104
F2	27 60	27 85	27 95	27 105
F3	27 61	27 86	27 96	27 106
F4	27 62	27 87	27 97	27 107
F5	27 63	27 88	27 98	27 108
F6	27 64	27 89	27 99	27 109
F7	27 65	27 90	27 100	27 110
F8	27 66	27 91	27 101	27 111
F9	27 67	27 92	27 102	27 112
F10	27 68	27 93	27 103	27 113

Key	Unshifted	Shift	Ctrl	Alt
LArr	27 75	52	27 115	27 178
RArr	27 77	54	27 116	27 180
UArr	27 72	56	27 160	27 175
DArr	27 80	50	27 164	27 183
Home	27 71	55		27 174
End	27 79	49	27 117	27 182
PgUp	27 73	57	27 132	27 176
PgDn	27 81	51	27 118	27 184
Ins	27 82	48	27 165	27 185
Del	27 83	46	27 166	27 186
Esc	27	27	27	
BackSp	8	8	127	
Tab	9	27 15		
RETURN	13	13	10	
A	97	65	1	27 30
B	98	66	2	27 48
C	99	67	3	27 46
D	100	68	4	27 32
E	101	69	5	27 18
F	102	70	6	27 33
G	103	71	7	27 34
H	104	72	8	27 35
I	105	73	7	27 23
J	106	74	10	27 36
K	107	75	11	27 37
L	108	76	12	27 38
M	109	77	13	27 50
N	110	78	14	27 49
O	111	79	15	27 24
P	112	80	16	27 25
Q	113	81	17	27 16
R	114	82	18	27 19
S	115	83	19	27 31
T	116	84	20	27 20
U	117	85	21	27 22
V	118	86	22	27 47
W	119	87	23	27 17
X	120	88	24	27 45
Y	121	89	25	27 21
Z	122	90	26	27 44

Key	Unshifted	Shift	Ctrl	Alt
[	91	123	27	
\	92	124	28	
]	93	125	29	
`	96	126		
0	48	41		27 129
1	49	33		27 120
2	50	64	27 3	27 121
3	51	35		27 122
4	52	36		27 123
5	53	37		27 124
6	54	94	30	27 125
7	55	38		27 126
8	56	42		27 127
9	57	40		27 128
*	42		27 114	
+	43	43		
-	45	95	31	27 130
=	61	43		27 131
,	44	60		
/	47	63		
;	59	58		

Table K-1: Keyboard Return Codes

Notes:

Appendix L
INSTALLATION

Terminal Installation

Before you use TURBO Pascal, it must be installed to your particular terminal, i.e. provided with information regarding control characters required for certain functions. This installation is easily performed using the program *TINST* which is described in this chapter.

After having made a work-copy, please store your distribution diskette safely away and work only on the copy.

Now start the installation by typing *TINST* at your terminal. Select **S**creen installation from the main menu. Depending on your version of TURBO Pascal, the installation proceeds as described in the following two sections.

IBM PC Display Selection

If you use TURBO Pascal without installation, the default screen set-up will be used. You may override this default by selecting another screen mode from this menu:

```
Choose one of the following displays:

  0) Default display mode
  1) Monochrome display
  2) Color display 80x25
  3) Color display 40x25
  4) b/w   display 80x25
  5) b/w   display 40x25

Which display (enter no. or ^X to exit) ■
```

Figure L-1: IBM PC Screen Installation Menu

Each time TURBO Pascal runs, the selected mode will be used, and you will return to the default mode on exit.

Non-IBM PC Installation

A menu listing a number of popular terminals will appear, inviting you to choose one by entering its number:

```
Choose one of the following terminals:

 1) ADDS 20/25/30          15) Lear-Siegler ADM-31
 2) ADDS 40/60             16) Liberty
 3) ADDS Viewpoint-1A      17) Morrow MDT-20
 4) ADM 3A                 18) Otrona Attache
 5) Ampex D80              19) Qume
 6) ANSI                   20) Soroc IQ-120
 7) Apple/graphics         21) Soroc new models
 8) Hazeltine 1500         22) Teletext 3000
 9) Hazeltine Esprit       23) Televideo 912/920/925
10) IBM PC CCP/M b/w       24) Visual 200
11) IBM PC CCP/M color     25) Wyse WY-100/200/300
12) Kaypro 10              26) Zenith
13) Kaypro II and 4        27) None of the above
14) Lear-Siegler ADM-20    28) Delete a definition

Which terminal? (Enter no. or ^X to exit):
```

Figure L-2: Terminal Installation Menu

If your terminal is mentioned, just enter the corresponding number, and the installation is complete. Before installation is actually performed, you are asked the question:

```
Do you want to modify the definition before installation?
```

This allows you to modify one or more of the values being installed as described in the following. If you do not want to modify the terminal definition, just type **N**, and the installation completes by asking you the operating frequency of your CPU (see last item in this appendix).

If your terminal is **not** on the menu, however, you must define the required values yourself. The values can most probably be found in the manual supplied with your terminal.

Enter the number corresponding to **None of the above** and answer the questions one by one as they appear on the screen.

In the following, each command you may install is described in detail. Your terminal may not support all the commands that can be installed. If so, just pass the command not needed by typing RETURN in response to the prompt. If *Delete line, Insert line,* or *Erase to end of line* is not installed, these functions will be emulated in software, slowing screen performance somewhat.

Commands may be entered either simply by pressing the appropriate keys or by entering the decimal or hexadecimal ASCII value of the command. If a command requires the two characters 'ESCAPE' and '=', may:

either: press first the **Esc** key, then the =. The entry will be echoed with appropriate labels, i.e. `<ESC> =`.

or: enter the decimal or hexadecimal values separated by spaces. Hexadecimal values must be preceded by a dollar-sign. Enter e.g. `27 61` or `$1B 61` or `$1B $3D` which are all equivalent.

The two methods cannot be mixed, i.e. once you have entered a non-numeric character, the rest of that command must be defined in that mode, and vise versa.

A hyphen entered as the very first character is used to delete a command, and echoes the text *Nothing*.

Terminal type:

Enter the name of the terminal you are about to install. When you complete *TINST* , the values will be stored, and the terminal name will appear on the initial list of terminals. If you later need to reinstall TURBO Pascal to this terminal, you can do that by choosing it from the list.

Send an initialization string to the terminal?
If you want to initialize your terminal when TURBO Pascal starts (e.g. to download commands to programmable function keys), you answer **Y** for yes to this question. If not, just hit RETURN.

Send a reset string to the terminal?
Define a string to be sent to the terminal when TURBO Pascal terminates. The description of the initialization command above applies here.

CURSOR LEAD-IN command:
Cursor Lead-in is a special sequence of characters which tells your terminal that the following characters are an address on the screen on which the cursor should be placed.

When you define this command, you are asked the following supplementary questions:

CURSOR POSITIONING COMMAND to send between line and column:
Some terminals need a command between the two numbers defining the row- and column cursor address.

CURSOR POSITIONING COMMAND to send after line and column:
Some terminals need a command after the two numbers defining the row- and column cursor address.

Column first?
Most terminals require the address on the format: first ROW, then COLUMN. If this is the case on your terminal, answer **N**. If your terminal wants COLUMN first, then ROW, then answer **Y**.

OFFSET to add to LINE
Enter the number to add to the LINE (ROW) address.

OFFSET to add to COLUMN
Enter the number to add to the COLUMN address.

Binary address?
Most terminals need the cursor address sent on binary form. If that is true for your terminal, enter **Y**. If your terminal expects the cursor address as ASCII digits, enter **N**. If so, you are asked the supplementary question:

2 or 3 ASCII digits?
Enter the number of digits in the cursor address for your terminal.

CLEAR SCREEN command:
Enter the command that will clear the entire contents of your screen, both foreground and background, if applicable.

Does CLEAR SCREEN also HOME cursor?
This is normally the case; if it is not so on your terminal, enter **N**, and define the cursor HOME command.

DELETE LINE command:
Enter the command that deletes the entire line at the cursor position.

INSERT LINE command:
Enter the command that inserts a line at the cursor position.

ERASE TO END OF LINE command:
Enter the command that erases the line at the cursor position from the cursor position through the right end of the line.

START OF 'LOW VIDEO' command:
If your terminal supports different video intensities, then define the command that initiates the **dim** video here. If this command is defined, the following question is asked:

START OF 'NORMAL VIDEO' command:
Define the command that sets the screen to show characters in 'normal' video.

Number of rows (lines) on your screen:
Enter the number of horizontal lines on your screen.

Number of columns on your screen:
Enter the number of vertical column positions on your screen.

Delay after CURSOR ADDRESS (0-255 ms):
Delay after CLEAR, DELETE, and INSERT (0-255 ms):
Delay after ERASE TO END OF LINE and HIGHLIGHT On/Off (0-255 ms):
Enter the delay in milliseconds required after the functions specified. RETURN means 0 (no delay).

Is this definition correct?

If you have made any errors in the definitions, enter **N**. You will then return to the terminal selection menu. The installation data you have just entered will be included in the installation data file and appear on the terminal selection menu, but installation will **not** be performed. When you enter **Y** in response to this question, you are asked:

Operating frequency of your microprocessor in MHz (for delays):

As the delays specified earlier are depending on the operating frequency of your CPU, you must define this value.

The installation is finished, installation data is written to TURBO Pascal, and you return to the outer menu (see section 12). Installation data is also saved in the installation data file and the new terminal will appear on the terminal selection list when you run *TINST* in future.

Editing Command Installation

The built-in editor responds to a number of commands which are used to move the cursor around on the screen, delete and insert text, move text etc. Each of these functions may be activated by either of two commands: a primary command and a secondary command. The secondary commands are installed by Borland and comply with the 'standard' set by *WordStar*. The primary commands are un-defined for most systems, and using the installation program, they may easily be defined to fit your taste or your keyboard. IBM PC systems are supplied with the arrows and dedicated function keys installed as primary commands as described in chapter 19.

When you hit **C** for Command installation, the first command appears:

```
CURSOR MOVEMENTS:

1:  Character left     Nothing -> ■
```


This means that no primary command has been installed to move the cursor one character left. If you want to install a primary command (in **addition** to the secondary *WordStar*-like Ctrl-S, which is not shown here), you may enter the desired command following the -> prompt in either of two ways:

1) Simply press the key you want to use. It could be a function key (for example a left-arrow-key, if you have it) or any other key or sequence of keys that you choose (max. 4). The installation program responds with a mnemonic of each character it receives. If you have a left-arrow-key that transmits an <ESCAPE> character followed by a lower case **a**, and you press this key in the situation above, your screen will look like this:

```
CURSOR MOVEMENTS:

1:  Character left     Nothing -> <ESC> a ■
```

2) Instead of pressing the actual key you want to use, you may enter the ASCII value(s) of the character(s) in the command. The values of multiple characters are entered separated by spaces. Decimal values are just entered: 27; hexadecimal values are prefixed by a dollar-sign: $1B. This may be useful to install commands which are not presently available on your keyboard, for example if you want to install the values of a new terminal while still using the old one. This facility has just been provided for very few and rare instances, because there is really no idea in defining a command that cannot be generated by pressing a key. But it's there for those who wish to use it.

In both cases terminate your input by pressing <RETURN>.Notice that the two methods cannot be mixed within one command. If you have started defining a command sequence by pressing keys, you must define all characters in that command by pressing keys and vise versa.

You may enter a - (minus) to remove a command from the list, or a **B** to back through the list one item at a time.

The editor accepts a total of 45 commands, and they may all be installed to your specification. If you make an error in the installation, like defining the same command for two different purposes, an self-explanatory error message is issued, and you must correct the error before terminating the installation. A primary command, however, may conflict with one of the *WordStar*-like secondary commands; that will just render the secondary command inaccessible.

The following table lists the secondary commands, and allows you to mark any primary commands installed by yourself:

	CURSOR MOVEMENTS:		
1:	Character left	Ctrl-S	
2:	Alternative	Ctrl-H	
3:	Character right	Ctrl-D	
4:	Word left	Ctrl-A	
5:	Word right	Ctrl-F	
6:	Line up	Ctrl-E	
7:	Line down	Ctrl-X	
8:	Scroll up	Ctrl-W	
9:	Scroll down	Ctrl-Z	
10:	Page up	Ctrl-R	
11:	Page down	Ctrl-C	
12:	To left on line	Ctrl-Q Ctrl-S	
13:	To right on line	Ctrl-Q Ctrl-D	
14:	To top of page	Ctrl-Q Ctrl-E	
15:	To bottom of page	Ctrl-Q Ctrl-X	
16:	To top of file	Ctrl-Q Ctrl-R	
17:	To end of file	Ctrl-Q Ctrl-C	
18:	To beginning of block	Ctrl-Q Ctrl-B	
19:	To end of block	Ctrl-Q Ctrl-B	
20:	To last cursor position	Ctrl-Q Ctrl-P	

	INSERT & DELETE:		
21:	Insert mode on/off	Ctrl-V	
22:	Insert line	Ctrl-N	
23:	Delete line	Ctrl-Y	
24:	Delete to end of line	Ctrl-Q Ctrl-Y	
25:	Delete right word	Ctrl-T	
26:	Delete character under cursor	Ctrl-G	
27:	Delete left character	<DEL>	
28:	Alternative:	Nothing	
	BLOCK COMMANDS:		
29:	Mark block begin	Ctrl-K Ctrl-B	
30:	Mark block end	Ctrl-K Ctrl-K	
31:	Mark single word	Ctrl-K Ctrl-T	
32:	Hide/display block	Ctrk-K Ctrl-H	
33:	Copy block	Ctrl-K Ctrl-C	
34:	Move block	Ctrl-K Ctrl-V	
35:	Delete block	Ctrl-K Ctrl-Y	
36:	Read block from disk	Ctrl-K Ctrl-R	
37:	Write block to disk	Ctrl-K Ctrl-W	
	MISC. EDITING COMMANDS:		
38:	End edit	Ctrl-K Ctrl-D	
39:	Tab	Ctrl-I	
40:	Auto tab on/off	Ctrl-Q Ctrl-I	
41:	Restore line	Ctrl-Q Ctrl-L	
42:	Find	Ctrl-Q Ctrl-F	
43:	Find & replace	Ctrl-Q Ctrl-A	
44:	Repeat last find	Ctrl-L	
45:	Control character prefix	Ctrl-P	

Table L-1: Secondary Editing Commands

Items 2 and 28 let you define alternative commands to *Character Left* and *Delete left Character* commands. Normally <BS> is the alternative to Ctrl-S, and there is no defined alternative to <DEL>. You may install primary commands to suit your keyboard, for example to use the <BS> as an alternative to <DEL> if the <BS> key is more conveniently located. Of course, the two alternative commands must be unambiguous like all other commands.

Notes:

Appendix M
CP/M PRIMER

How to use TURBO on a CP/M system

When you turn on your computer, it reads the first couple of tracks on your CP/M diskette and loads a copy of the CP/M operating system into memory. Each time you re-boot your computer, CP/M also creates a list of the disk space available for each disk drive. Whenever you try to save a file to the disk, CP/M checks to make sure that the diskettes have not been changed. If you have changed the diskette in Drive A without re-booting, for example, CP/M will generate the following error message when a disk-write is attempted:

```
BDOS ERROR ON A:  R/O
```

Control will return to the operating system and your work was NOT saved! This can make copying diskette a little confusing for the beginner. If you are new to CP/M, follow these instructions:

Copying Your TURBO Disk

To make a working copy of your TURBO MASTER DISK, do the following:

1. Make a blank diskette and put a copy of CP/M on it (see your CP/M manual for details). This will be is your TURBO **work disk**.

2. Place this disk in Drive A:. Place a CP/M diskette with a copy of PIP.COM in Drive B (PIP.COM is CP/M's file copy program that should be on your CP/M diskette. See your CP/M manual for details).

3. Re-boot the computer. Type `B:PIP` and then press <RETURN>

4. Remove the diskette from Drive B: and insert your TURBO MASTER DISK.

5. Now type: `A:=B:*.*[V]` and then press <RETURN>

   You have instructed PIP it to copy all the files from the diskette in Drive B: onto the diskette in Drive A:. Consult your CP/M manual if any errors occur.

The last few lines on your screen should look like this:

```
A> B:PIP

*A:=B:*.*[V]

COPYING -
FIRSTFILE
:
:
LASTFILE
*
```

6. Press <RETURN>, and the PIP program will end.

Using Your TURBO Disk

Store your TURBO MASTER DISK in a safe place. To use TURBO PASCAL, place your new TURBO **work disk** in drive A: and re-boot the system. Unless your TURBO came pre-installed for your computer and terminal, you should install TURBO (see 12). When done, type

```
TURBO
```

and TURBO Pascal will start.

If you have trouble copying your diskette, please consult your CP/M user manual or contact your hardware vendor for CP/M support.

Appendix N
HELP!!!

This appendix lists a number of the most commonly asked questions and their answers. If you don't find the answer to *your* question here, you can either call Borland's technical support staff, or you can access CompuServe's Consumer Information 24 hours a day and 'talk' to the Borland Special Interest Group. See insert in the front of this manual for details.

Q: How do I use the system?
A: Please read the manual, specifically chapter 1 . If you must get started immediately do the following:
1) Boot up your operating system
2) If you have a computer other than an IBM PC, run Tinst to install Turbo for your equipment.
3) Run Turbo
4) Start programming!

Q: I am having trouble installing my terminal!
A: If your terminal is not one that is on the installation menu you must create your own. All terminals come with a manual containing information on codes that control video I/O. You must answer the questions in the installation program according to the information in your hardware manual. The terminology we use is the closest we could find to a standard. Note: most terminals do not require an initialization string or reset string. These are usually used to access enhanced features of a particular terminal; for example on some terminals you can send an initialization string to make the keypad act as a cursor pad. You can put up to 13 characters into the initialization or reset strings.

Q: I am having disk problems. How do I copy my disks?
A: Most disk problems do not mean you have a defective disk. Specifically, if you are on a CP/M-80 system you may want to look up the brief CP/M primer on page 355 . If you can get a directory of your distribution disk, then chances are that it is a good disk.

To make a backup copy of Turbo you should use a file-by-file copy program like *COPY* for PC/MS-DOS or *PIP* for CP/M-80/86. The reason is that for those of you who have quad density disk drives, you may have trouble using a DISKCOPY type program. These programs are expecting the exact same format for the Source diskette as well as the Destination diskette.

Q: Do I need an 8087 chip to use Turbo-87?
A: Yes, if you want to compile programs for the 8087 chip, that chip must be in your machine. The standard TURBO compiler, however, is included on the Turbo-87 disk, so you can have it both ways!

Q: Do I need any special equipment to use TURBO-BCD?
A: No, but the BCD reals package works on 16 bit implementations of Turbo only.

Q: Do I need Turbo to run programs I developed in Turbo?
A: No, Turbo can make .COM or .CMD files.

Q: How do I make .COM or .CMD files?
A: Type O from the main menu for Compiler Options and then select "C" for .COM or .CMD file.

Q: What are the limits on the compiler as far as code and data?
A: The compiler can handle up to 64K of code, 64K of data, 64K of stack and unlimited heap. The object code, however, cannot exceed 64K.

Q: What are the limits of the editor as far as space?
A: The editor can edit as much as 64K at a time. If this is not enough, you can split your source into more than one file using the *$I* compiler directive. This is explained in chapter 17.

Q: What do I do when I get error 99 (Compiler overflow)?
A: You can do two things: break your code into smaller segments and use the *$I* compiler directive (explained in chapter 17) or compile to a .COM or .CMD file.

Q: What do I do if my object code is going to be larger than 64K?
A: Either use the chain facility or use overlays.

Q: How do I read from the keyboard without having to hit return (duplicate BASIC's INKEY$ function)?
A: Like this: `read(Kbd,Ch)` where *Ch:Char*.

Q: How do I get output to go to the printer?
A: Try: `Writeln(Lst, ...)`.

Q: How can I get a listing of my source code to my printer?
A: You can use the following program. If you wish to have a listing that underlines or highlights reserved words, puts in page breaks, and lists all Include files, there is one included free (including source) on the Turbo Tutor diskette.

```
program TextFileDemo;

var
  TextFile : Text;
  Scratch  : String[128];

begin
  Write('File to print: ');          { Get file name    }
  Readln(Scratch);
  Assign(TextFile, Scratch);         { Open the file    }
  {$I-}
  Reset(TextFile);
  {$I+}
  if IOresult <> 0 then
    Writeln('Cannot find ', Scratch) { File not found   }
  else                               { Print the file.. }
  begin
    while not Eof(TextFile) do
    begin
      Readln(TextFile, Scratch);     { Read a line      }
      Writeln(Lst, Scratch)          { Print a line     }
    end; { while }
    Writeln(Lst)                   { Flush printer buffer }
  end { else }
end.
```

Q: How do I get output to and input from COM1:?
A: Try: writeln(AUX, ...) after setting up the port using MODE from MSDOS or an equivalent ASSIGN type program from CP/M. To read try read(AUX, ...). You must remember that there is no buffer set up automatically when reading from AUX.

Q: How do I read a function key?
A: Function keys generate 'extended scan codes' which are turned into 'escape sequences' by TURBO, that is, **two** characters are sent from the keyboard: first an Esc (decimal ASCII value 27), then some other character. You'll find a table of all values on page 341.

To read these extended codes, you check for ESC and if detected see if there is another character in the keyboard buffer. If there is, a function key was pressed, so you read the next character and set a flag to signal that what you got is not a normal character, but the second part of an 'escape sequence'

```
if KeyPressed then
begin
  Read(Kbd,Ch)                          {ch is char}
  if (ch = #27) and KeyPressed then {one more char?}
  begin
    Read(Kbd,Ch)
    FuncKey := True;              {FuncKey is boolean}
  end
end;
```

Q: I am having trouble with file handling. What is the correct order of instructions to open a file?
A: The correct manner to handle files is as follows:

To create a new file:

```
Assign(FileVar,'NameOf.Fil');
Rewrite(FileVar);
:
:
Close(FileVar);
```

To open an existing file:

```
Assign(fileVar,'NameOf.Fil');
Reset(FileVar);
:
:
Close(FileVar);
```


Q: Why do my recursive procedures not work?
A: Set the A compiler directive off:{$A-}(CP/M-80 only)

Q: How can I use EOF and EOLN without a file variable as a parameter?
A: Turn off buffered input:{$B-}

Q: How do I find out if a file exists on the disk?
A: Use {$I-} and {I+}. The following function returns *True* if the file name passed as a parameter exists, otherwise it returns *False*:

```
type
 Name=string[66];
:
:
function Exist(FileName: Name): Boolean;
Var
  Fil: file;
begin
  Assign(Fil, FileName);
  {$I-}
  Reset(Fil);
  {$I+}
  Exist := (IOresult = 0)
end;
```

Q: How do I disable CTRL-C?
A: Set compiler directive: {$C-}.

Q: I get a Type Mismatch error when passing a string to a function or procedure as a parameter.
A: Turn off type checking of variable parameters: {$V-}.

Q: I get file not found error on my include file when I compile my program - even though the file is in the directory.
A: When using the include compiler directive *{$I filename.ext}* there must be a space separating the filename from the terminating brace, if the extension is not three letters long: {$ISample.F }. Otherwise the brace will be interpreted as part of the file name.

Q: Why does my program behave differently when I run it several times in a row?
A: If you are running programs in **M**emory mode and use typed constants as initialized variables, these constants will only be initialized right after a compilation, not each time you **R**un the program as they reside in the code segment. With .COM files, this problem does not exist, but if you still experience different results when using arrays and sets, turn on range checking {$R+}.

Q: I don't get the results I think I should when using *Reals* and *Integers* in the same expression.
A: When assigning an Integer expression to a Real variable, the expression is converted to Real. However, the expression itself is calculated as an integer, and you should therefore be aware of possible integer overflow in the expression. This can lead to surprising results. Take for instance:

```
RealVar := 40 * 1000;
```

First, the compiler multiplies integers 40 and 1000, resulting in 40,000 which gives integer overflow. It will actually come out to -25536 as Integers wrap around. Now it will be assigned to the RealVar as -25536. To prevent this, use either:

```
RealVar := 40.0 * 1000;
```

or

```
RealVar := 1.0 * IntVar1 * IntVar2;
```

to ensure that the expression is calculated as a Real.

Q: How do I get a disk directory from my TURBO program?
A: Sample procedures for accessing the directory are included in the TURBO Tutor package (see how to order the TURBO Tutor on page 3).

Q: My program works well with TURBO 2.0, but now it keeps getting I/O Error F3 (or TURBO Access error 243)
A: TURBO 3.0 uses DOS file handles. When booting your computer, you should have a CONFIG.SYS file in the root directory of your boot drive. Place the statement:

```
FILES=16
```

in this file and re-boot your system. For more information about file handles, please refer to your DOS reference manual.

NOTE: If you distribute your programs, you should include similar instructions in the documentation that you provide.

Appendix O. SUBJECT INDEX

A

A-command, 192, 229
A-compiler directive, 286
Abort command, 34
Abs, 139
Absolute address function, 204, 237
Absolute value, 139
Absolute variable, 203, 236, 261, 267
Adding operator, 51, **53**
Addr, 204, 237
Addr Function, 268
Allocating variable (New), 120
Append procedure, 200
Arc, 173
ArcTan, 139
Arithmetic functions, 139, 304
Array component, 75
Array constant, 90
Array definition, 75
Array of characters, 112
Array subscript optimization, 269
Array, 75, 219, 224, 254, 285, 249, 281
Assign, 94
Assignment operator, 37
Assignment statement, 55
Auto indent on/off switch, 31
Auto indentation, 35
Automatic overlay management, 155
AUX:, 104

B

Back, 178
Background color, 161
Backslash, 188
Backspace, 109
Backup, 17
BAK file, 17
Basic data types, 216, 246, 278
Basic graphics, 171
    Windows, and sound, 308
Basic movement commands, 22
Basic symbols, 37
BCD range, 293
BDOS, 261
Bdos procedure and function, 271
BdosHL function, 271
Begin block, 28
Bios procedure and function, 272
BiosHL function, 272
Blanks, 39
Blink, 161
Block, 127
Block size, 235
Block commands, 28
    Begin block, 28
    Copy block, 29
    Delete block, 29
    End block, 28
    Hide/display block, 29
    Mark single word, 28
    Move block, 29
    Read block from disk, 29
    Write block to disk, 30
BlockRead, 114