TURBO PASCAL LIBRARY

Version 2.1

Paul Coxwell
30 Alford Road
Sutton-on-Sea
LN12 2HH
England

```
************************************************************
*                                                          *
*                    TURBO PASCAL LIBRARY                  *
*                        Version 2.1                       *
*                         June 1993                        *
*                                                          *
*    A library of routines for Borland's Pascal compiler   *
*                                                          *
************************************************************
```

TABLE OF CONTENTS

1. INTRODUCTION

Welcome to Turbo Pascal Library, a collection of routines to ease the development of application programs with Borland's Turbo Pascal compiler. This version of the library contains units covering string manipulation, date and time conversion and formatting, routines for the conversion of units of measurement, improved error handling, enhanced console input/output, windows, and an on-line help system.

Turbo Pascal Library requires an IBM PC or compatible, DOS version 2.0 or later, and Turbo Pascal version 5.0 or later. A fixed disk drive is not essential, but is highly recommended for any serious development work. Some portions of the library are written in assembler; to modify these you will need a standard 8086 assembler, such as Borland's Turbo Assembler. The intermediate object code files are included so that you can modify and recompile the Pascal source code without access to an assembler.

The files that comprise this package are stored on the distribution diskette in compressed form. To extract all the code enter the command

INSTALL C:\PATH

where "C:\PATH" is the drive and path where you want the files to be placed. Because of the incompatibility of TPU files compiled under different versions of Turbo Pascal, no TPU files are supplied. A batch file is included which will compile all the units using the command-line version of the compiler (TPC).

You should copy the TPU files to a directory that is searched by Turbo Pascal when compiling a program. You should also ensure that the OBJ files are in a directory that will be searched at link time if you intend to modify the units in any way.

The code in this package is released for public use without payment. You are free to use, copy, and distribute this package as widely as you wish. Your comments about this package and suggestions for future revisions are always welcome. The address to write to is shown on the cover of this manual.

HAPPY PROGRAMMING!

2.
UNIT STRINGS

Turbo Pascal provides a fairly standard selection of string-handling procedures and functions. This unit expands upon those routines defined by Turbo Pascal and offers routines for manipulating strings in a number of ways: replication, padding, truncation, justification, replacement, and so on.

GLOBAL DECLARATIONS

The following constant strings are defined in the interface section of the unit:

```
UCaseLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
LCaseLetters = 'abcdefghijklmnopqrstuvwxyz';
Letters      = UCaseLetters + LCaseLetters;
DecDigits    = '0123456789';
HexDigits    = '0123456789ABCDEF';
OctDigits    = '01234567';
BinDigits    = '01';
```

These will be found useful with many of the string manipulation routines.

The unit also defines a new type and a global variable which consists of four fields:

```
FormatConfigRec = RECORD
                    Fill,
                    Currency,
                    Overflow,
                    FracSep:        CHAR;
                  END;

FormatConfig: FormatConfigRec =
                    (Fill:     '*';
                     Currency: '$';
                     Overflow: '?';
                     FracSep:  '-');
```

This global variable is used by the Format function, and the fields are described in detail when appropriate.

```
FUNCTION LoCase (ch: CHAR): CHAR;
```

Turbo Pascal provides an UpCase function to convert any character to its upper-case equivalent. This function performs the converse function and converts a character to its lower-case equivalent.

The character supplied as parameter ch is converted to lower case and returned as the function's result. If the supplied character is already lower case or is not alphabetic, it is returned unchanged.

```
FUNCTION UpperCase (s: STRING): STRING;
FUNCTION LowerCase (s: STRING): STRING;
```

These two functions perform case conversion on an entire string of characters. The returned string is always equal in length to the supplied string, and all alphabetic characters are converted to upper or lower case as appropriate. Non-alphabetic characters, or characters which are already in the required case, are not changed.

```
FUNCTION DuplChar (ch: CHAR; count: BYTE): STRING;
```

DuplChar returns a string of characters of the length requested in parameter count. The character supplied in ch is used to build the string. If count is zero, a null string is returned.

```
FUNCTION DuplStr (s: STRING; count: BYTE): STRING;
```

This function is similar to DuplChar, but duplicates a string of characters instead of a single character. The returned string comprises the supplied string, s, concatenated to itself the number of times specified by count.

Note that the maximum length of a string in Turbo Pascal is 255 characters. If the length of s multiplied by count is greater than 255, then the returned string will be truncated after the 255th character. If count is zero, a null string is returned.

```
FUNCTION TrimL (s: STRING): STRING;
FUNCTION TrimR (s: STRING): STRING;
```

These two functions allow blanks to be trimmed from either the beginning or end of a string of characters. TrimL (Trim Left) returns the supplied string with all leading blanks removed. TrimR (Trim Right) returns the supplied string with all trailing blanks removed. If there are no blanks, the string is returned unchanged.

Both functions regard a space (ASCII code 32 decimal, 20 hex.) and a tab (ASCII code 9) as being blanks; trimming will stop as soon as any other character is encountered. Both TrimL and TrimR return a null string if s is a null string or consists of spaces and tabs only.

```
FUNCTION PadL (s: STRING; width: BYTE): STRING;
FUNCTION PadR (s: STRING; width: BYTE): STRING;
```

PadL (Pad Left) and PadR (Pad Right) allow a string to be padded out to a specified length by the addition of spaces to the start or end of the supplied string, s. PadL adds spaces (ASCII code 20 hex.) to the start of the string to make it up to the length supplied in parameter width. PadR adds spaces to the end of the string to make it up to the requested width.

If spaces or tabs are already present at the start or end of the supplied string they are not removed. To justify a string by removing existing blanks and then adding those required, use the justify functions described below.

If the supplied string is a null, both functions return a string of spaces of the specified length. If the length of the supplied string is already equal to or greater than the specified width it is returned unchanged. PadL and PadR will never return a string of less than the specified width, but may return one which is longer.

```
FUNCTION TruncL (s: STRING; width: BYTE): STRING;
FUNCTION TruncR (s: STRING; width: BYTE): STRING;
```

These functions truncate a string to the specified width. TruncL removes characters from the start of a string and TruncR removes them from the end.

Both functions return a null string if width is specified as zero, and both functions return the string unchanged if its length is already less than or equal to the

specified width. TruncL and TruncR will never return a string longer than the specified width, but may, therefore, return one which is shorter.

```
FUNCTION JustL (s: STRING; width: BYTE): STRING;
FUNCTION JustR (s: STRING; width: BYTE): STRING;
```

These justification functions combine the effects of the trim, pad, and truncate functions described above.

JustL ensures that a string is left justified within a specified field width. All blanks are removed from both the beginning and end of the string. The remaining string is truncated from the right if it is too long for the requested width or spaces are added to the end of the string if it is too short.

JustR ensures that a string is right justified within a specified field width. All blanks are removed from both the beginning and end of the string, then the remaining string is truncated from the left if it is too long for the requested width or spaces are added to the left of the string if it is too short.

JustL is equivalent to TrimL, TrimR, TruncR, and PadR being applied to a string, in that order. JustR is equivalent to TrimL, TrimR, TruncL, PadL. Both functions, therefore, will always return a string of the specified length by either truncating the supplied string or padding it as required. If width is zero, a null string is returned.

```
FUNCTION JustC (s: STRING; width: BYTE): STRING;
```

JustC (Justify Center) positions a string centrally within the specified field width.

First, leading and trailing blanks are trimmed and the string is truncated from the right if it is too long (this part is identical to the JustL function). If the remaining string is shorter than the requested length spaces are added equally to the left and right of the string to make it up to the required width.

If the trimmed and truncated string contains an odd number of characters and the requested width is an even number, or vice versa, it will not be possible to add an equal number of spaces to the left and right of the string in order to center it. In such cases, JustC places the

extra "odd" space at the right, leaving the string justified slightly to the left.

JustC, like JustL and JustR, will always return a string of the specified length. If width is specified as zero, a null string is returned.

FUNCTION Precede (s, target: STRING): STRING;

Precede searches for a sub-string, target, within a string, s. If the search is successful the function returns a string consisting of all the characters up to, but not including, the target string. If the target string is not found, the entire string is returned unchanged.

FUNCTION Follow (s, target: STRING): STRING;

Follow searches for a sub-string, target, within a string, s. If the target string is found the function returns a string consisting of all the characters that follow the search string. If the search is unsuccessful, Follow returns a null string.

FUNCTION Break (VAR s: STRING; d: STRING): STRING;

The Break function emulates the function of the same name found in the SNOBOL programming language.

A string to be parsed is supplied as parameter s and a set of delimiter characters is supplied as parameter d. Break scans string s for the first of any of the delimiter characters in d. The returned string consists of all characters in the supplied string up to, but not including, the delimiter character found. In addition, Break modifies the string s so as to remove the characters preceding the delimiter (i.e. the characters returned by Break are also removed from the source string).

Note that the order of the characters in the string of delimiters is unimportant; Break scans until it finds the first of any of the characters. If no delimiter is found during the search, Break returns the entire input string and sets the input string to a null.

FUNCTION Span (VAR s: STRING; d: STRING): STRING;

Span also emulates a SNOBOL function, and is similar in operation to Break. The difference is that Span searches for a character which is not specified in string d, rather than one that is.

Span searches the input string, s, checking that each character of s is also present in string d. The scan stops when a character is found in s which is not also present in d. Span then returns all the characters scanned and, like Break, removes them from the source string, s.

As with Break, the order in which the characters are specified in d has no effect on the outcome of the call to Span. If every character in the input string is listed in d, Span returns the entire string and sets s to a null string.

Break and Span provide a very powerful way to parse input and remove each element as it is dealt with. The string constants listed in the introduction to this unit will be found helpful for such applications.

FUNCTION Replace (s, srch, repl: STRING): STRING;

Replace also closely emulates a SNOBOL function; the source string, s, is searched for any of the characters listed in the string srch. When such a character is found, it is replaced by a character in the string repl. The search is continued until the end of the string is reached.

Strings srch and repl should be equal in length, and when a character in srch is located it is replaced by the character in the equivalent position in repl (e.g. if srch contains 'ABC' and repl contains 'XYZ' then an 'A' is replaced with an 'X', a 'B' with a 'Y', and a 'C' with a 'Z'). The function returns a string of the same length as s, but with all requested substitutions in place.

Note that if repl is longer than srch, then the extra characters in repl are ignored. Similarly, if the length of string srch is greater than that of repl, the extra characters specified in srch are not searched for. If either srch or repl is a null string, Replace will return the input string unchanged.

FUNCTION Remove (s, srch: STRING): STRING;

Remove searches the source string, s, for any of the characters listed in srch. The returned string is s with all such characters removed.

The order in which the characters are listed in srch does not matter; all characters matching those listed will be deleted. If srch is a null string or the supplied string contains none of the characters listed in srch, then s is returned unchanged.

FUNCTION StripBit7 (s: STRING): STRING;

This function returns the supplied string with the high-order bit of each character cleared. This converts ASCII codes 128 through 255 (decimal) to codes 0 through 127, thus changing symbols and foreign characters to standard 7-bit ASCII code. Another use would be to strip a parity bit from data received over a communication channel.

The returned string is always equal in length to the supplied string and characters which are already in the conventional ASCII set (0 through 127 decimal) are unaffected.

FUNCTION FileSpecDefault (s,path,name,extn: STRING): STRING;

This function is specifically designed to enable user's input of a file path and name to be processed, and allow any section which is absent from the input to be set to a predetermined string.

The user's input should be passed as parameter s, and defaults for the path, name, and file extension should be passed in path, name, and extn, respectively. Turbo Pascal's FSplit procedure is used to break the user's input into three sections. If any section is missing from the user's input it is taken from the supplied defaults; FileSpecDefault returns a string with the three sections re-assembled.

When specifying the default path, ensure that a final backslash is included if required. Similarly, the extension should include the leading period delimiter.

```
FUNCTION HexStr (n: WORD; count: BYTE): STRING;
FUNCTION OctStr (n: WORD; count: BYTE): STRING;
FUNCTION BinStr (n: WORD; count: BYTE): STRING;
```

These three functions allow an integer in the range 0 through 65,535 to be coverted to a string holding the hexadecimal, octal, or binary representation of the number.

The number to be converted should be supplied in parameter n and the number of hexadecimal, octal, or binary digits required in the string should be passed in parameter count. Each function returns a string of the length specified by count, with leading zeros if necessary. Note that if the number of digits specified is not sufficient to represent the number then the most-significant digits are lost.

```
FUNCTION Format (n: REAL; form: STRING): STRING;
```

Although the standard Write and WriteLn procedures allow a number to be output to any given field width and number of decimal places, more control over the display and printing of numbers is often required.

Format takes the number passed in parameter n and returns a string suitable for printing. The contents of the parameter form determine the exact way in which the output is formatted. The variable FormatConfig, defined in the interface section of the unit, also controls the way in which some features work. This variable contains four fields: Fill, Currency, Overflow, and FracSep. These are described below when appropriate.

The basic formatting string closely resembles that used with Write and WriteLn. Two numbers, separated by a colon, specify the total width of the returned string and the number of decimal places. The format string '10:2', for example, specifies a total width of 10 characters and that the number should be rounded to 2 decimal places. If either number is omitted, or invalid, the default values are a total width of 12 characters and zero decimal places. The format strings '8' and '8:0', for example, are equivalent. Format will always return a string equal in length to the width specified. If the width is insufficient for the number being formatted, the returned field is filled with an overflow warning character, which defaults to a question mark. You can change this character by making an assignment to the Overflow field of FormatConfig.

```
Format (25.631, '8:2')      returns '   25.63'
Format (-46.9, '4')         returns ' -47'
Format (1000, '3')          returns '???'
```

SIGN OPTIONS

Format offers several different ways to display the sign (positive or negative) of a number. The default is to prefix negative numbers with a minus sign. If the format string contains a leading "+" then a sign is added even if the number is positive.

```
Format (-8.3, '6:1')        returns '  -8.3'
Format (8.3, '6:1')         returns '   8.3'
Format (-20.4, '+6:1')      returns ' -20.4'
Format (20.4, '+6:1')       returns ' +20.4'
```

The sign may also be placed after the number, by putting a "+" or "-" at the end of the format string. The former option causes positive and negative numbers to be displayed with a sign; the latter formats positive numbers with a trailing space for ease of alignment of printed columns.

```
Format (-32, '5-')          returns '  32-'
Format (32, '5-')           returns '  32 '
Format (3.2, '5:1+')        returns ' 3.2+'
Format (-3.2, '5:1+')       returns ' 3.2-'
```

The final sign option allows negative numbers to be enclosed in parentheses and is specified with a "P" in front of the size specifiers. Positive numbers are formatted with a trailing space for ease of alignment. (Note that all alphabetic options in the format string may be in either upper or lower case.)

```
Format (123.45, 'P10:2')    returns '   123.45 '
Format (-678.9, 'P10:2')    returns '  (678.90)'
```

It is also possible to display the absolute value of a number by using the "A" option. This causes negative numbers to be converted to positive numbers before formatting.

```
Format (-68.4, 'A8:1')      returns '    68.4'
Format (-68.4, '+A8:1')     returns '   +68.4'
```

JUSTIFICATION AND FILL OPTIONS

There are controls for the justification of the number and for controlling how the remainder of the field is treated. Each of these options is controlled by adding the appropriate character to the front of the format string. When several options precede the size specifiers they may appear in any order.

Placing a "C" in the control string causes numbers of 1,000 or greater (or -1,000 or lower) to have embedded commas. This option may also be specified with a "," in the format string.

```
Format (2000, '7')          returns '   2000'
Format (2000, 'C7')         returns '  2,000'
```

It is also possible to have values of zero displayed as a blank field by using the "B" option. Any non-zero value is formatted as per the rest of the format string. Note that the zero test is performed on the number after it is rounded for display. This means, for example, that a value of 0.005 formatted to one decimal place would be considered to be zero and shown as a blank field.

```
Format (100, 'B6:2')        returns '100.00'
Format (-0.002, 'B6:2')     returns '      '
```

It is sometimes desirable to ensure that numbers fill the entire width of the field, such as when printing checks. Three options allow the blanks within a formatted number to be filled. If the format string contains an asterisk the left of the field is filled with asterisks. These are placed before any leading sign or currency symbol.

```
Format (-45, '*8:1')        returns '***-45.0'
Format (12345, '*8:1')      returns '*12345.0'
```

A similar effect can be achieved by using an "F" in the control string, but in this case Format uses the character assigned to the Fill field of FormatConfig. This field is initially set to an asterisk, giving the "*" and "F" options the same meaning unless altered by your program. Note that an assignment to FormatConfig.Fill has no effect on the "*" option.

Placing a "Z" in the control string causes a number to be printed with leading zeros, and these are placed after any sign or currency symbol. Note that the "B" option takes precedence over all three fill options if the rounded number is zero.

```
Format (2500, 'Z8:1')       returns '00002500'
Format (2500, '+Z8:1')      returns '+0002500'
Format (0.03, 'BZ8:1')      returns '        '
```

A floating currency symbol can be placed in front of the formatted amount by using a dollar sign ahead of the field width in the control string. When combined with a sign, the symbol is always placed after a plus, minus, or opening parenthesis. It will also always be placed after any fill characters.

```
Format (99.50, '$9:2')      returns '   $99.50'
Format (-47.25, 'P$9:2')    returns ' ($47.25)'
Format (1.25, 'P*$9:2')     returns '***$1.25 '
```

The currency symbol can be changed to any other character by an assignment to the Currency field of FormatConfig. This would typically be done during your program's initialization routine. Note that you must still use "$" in the string passed to Format; only the resulting character in the formatted string is affected by the assignment. Assigning the null code (#0) to FormatConfig.Currency will cause the "$" option to be ignored.

Output from Format is normally right-justified, with all leading positions either blank or filled with the specified fill character. This allows simple alignment when printing figures in columns. Use of the "L" option in a format string causes Format to left-justify the returned string, with trailing blanks to make up the correct field width. The "*" and "F" options may be used, in which case the specified fill character will pad the returned number to the right. Left-justification has no effect when the "Z" option is also selected, since the zeros must be added to the left of the number.

DISPLAY OF VULGAR FRACTIONS

The final group of formatting options allow the display of numbers in vulgar fraction form instead of decimal form. The width specifier value should be followed by ":/" and then the value of denominator required. For example, the format string '12:/8' will format numbers to a field width of 12 characters, rounded to the nearest eighth.

```
Format (1.125, '12:/8')     returns '     1-1/8  '
Format (3.623, '12:/8')     returns '     3-5/8  '
```

The maximum allowable value for the denominator is 99. Formatted output always allows five character positions for fractional part of the number, plus one extra space for the

"-" separator. This allows the integer part of the number to be easily aligned when printing numbers in columns. The character used to separate the integral and fractional parts of the number can be changed by an assignment to the FracSep field of FormatConfig.

Use of vulgar fractions automatically selects the "A" option to force absolute values and cancels any sign options. The "*" and "F" fill options may be used, and in addition to their usual function will also cause any space to the right of formatted number to be filled with the specified character. Zero-fill is not available with vulgar fractions, and a "Z" in the format control string is interpreted differently: Use of this option causes values of less than 1 to be shown with a leading zero.

```
Format (0.25, '8:/4')    returns '   1/4  '
Format (0.25, 'Z8:/4')   returns ' 0-1/4  '
```

The final option affecting vulgar fractions is the "R" option to disable fraction reduction. By default, Format will reduce any fraction to the lowest possible form; use of the "R" option causes Format to leave the fraction in the exact form specified, even if it could be reduced.

```
Format (0.5, '6:/8')    returns ' 1/2  '
Format (0.5, 'R6:/8')   returns ' 4/8  '
```

CONTROL STRING SUMMARY

```
    [Options]  [Width] [:Decimals] [:/Fraction]  [Options]
```

Width defaults to 12 characters if not specified. Decimals defaults to zero if not specified. Fraction may be any value in the range 2 through 99. Leading options may be placed in any order; upper and lower case are equivalent.

Trailing options:

+	Always add trailing plus or minus sign
-	Add trailing minus if negative

Leading options:

+	Always add leading plus or minus sign
-	Add leading minus if negative (default)
P	Format negative numbers in parentheses
A	Show absolute value of supplied number
C ,	Add commas for thousands
B	Return blank field if result is zero
*	Pad field with asterisks
F	Pad field with user-defined character
Z	Zero-fill or add zero before vulgar fraction
$	Add floating currency symbol
L	Left-justify formatted number
R	Do not reduce vulgar fractions

3.
UNIT MATH & MATH87

Conversion between various units of measurement is required in a great many application programs. The MATH and MATH87 units provide a wide range of functions for the conversion of linear, square, and cubic measurements, temperatures, weights, and so on.

All the routines require the passing of floating-point numbers, and to provide best versatility the units allow you to use regular REAL numbers or EXTENDED floating-point numbers. The latter use the 8087 co-processor or Turbo Pascal's emulation (see your reference manual for details). If you wish to use regular REAL numbers, declare the MATH unit in your program's USES statement; to use EXTENDED floating-point variables, declare the MATH87 unit. Both units define a new variable type, FLOAT, which is set to be equivalent to type REAL or type EXTENDED, respectively.

In programs which make extensive use of measurements, it is recommended that the FLOAT type variable is always used to hold these measurements in their basic form of inches for linear measurement, square inches for area, and cubic inches for volume. It is then easy to perform calculations on the variables without having to always check whether the units of measurement are the same. You can use the conversion routines described below when input and ouptut is required in feet, miles, square yards, and so on. The basic unit for liquid measure is assumed to be the fluid ounce and the basic unit for weight is assumed to be the ounce.

Metric conversions are also included in the MATH/MATH87 unit. The basic measurement units are taken to be the meter, square meter, cubic meter, and gram. Metric prefixes, such as for centimeters, square kilometers, kilograms, and so on are easily derived from these basic forms.

```
FUNCTION FahrToCent (FahrTemp: FLOAT): FLOAT;
FUNCTION CentToFahr (CentTemp: FLOAT): FLOAT;
FUNCTION KelvToCent (KelvTemp: FLOAT): FLOAT;
FUNCTION CentToKelv (CentTemp: FLOAT): FLOAT;
```

These four functions provide temperature conversions between Fahrenheit, Centigrade, and Kelvin measurements. The supplied temperature is converted as required and returned by the function. For example, FahrToCent takes a Fahrenheit temperature passed in parameter FahrTemp and returns the Centigrade equivalent.

Direct conversions between Fahrenheit and Kelvin can be made by using the return value of one function as the parameter to another. For example:

```
K := CentToKelv(FahrToCent(F));
```

will convert the Fahrenheit temperature F to its Kelvin equivalent K.

Note that no error checking is performed by these functions; it is left entirely to your own program to ensure that valid values (i.e. above absolute zero) are passed to the functions.

```
PROCEDURE InchToFtIn (Inches: FLOAT; VAR ft, ins: FLOAT);
FUNCTION  FtInToInch (ft, ins: FLOAT): FLOAT;
```

The FLOAT type variable, whether defined as type REAL or type EXTENDED, has adequate capacity to allow most measurements to be stored directly in inches. Some applications, however, are easier to use if input and output is separated into feet and inches; these two routines provide this conversion.

InchToFtIn takes the measurement passed as parameter Inches and separates it into feet and inches which are returned in parameters ft and ins respectively. An input value of 61, for example, will cause values of 5 and 1 to be returned.

FtInToInch performs the converse function and combines the two supplied parameters to return a single measurement in inches. Note that it is quite permissible for ins to be 12 or greater. For example, if ft is passed as 3 and ins is passed as 25, then the function will return a value of 61.

```
FUNCTION InchToYard (Inches: FLOAT): FLOAT;
FUNCTION YardToInch (Yards: FLOAT): FLOAT;
```

These two functions provide direct conversion between inches and yards, for situations where measurements are stored in inches but input or output must be in yards.

InchToYard takes the supplied parameter Inches and returns a value that represents the same distance in yards. YardToInch takes the parameter Yards and returns the equivalent measurement in inches.

Note that negative values are permissible with all

conversion functions such as these; the returned value will simply reflect the sign of the supplied parameter.

```
FUNCTION InchToMile (Inches: FLOAT): FLOAT;
FUNCTION MileToInch (Miles: FLOAT): FLOAT;
FUNCTION InchToNautMile (Inches: FLOAT): FLOAT;
FUNCTION NautMileToInch (NautMiles: FLOAT): FLOAT;
```

Four functions provide direct conversion between inches and miles. InchToMile and MileToInch take the supplied parameter and convert it using the standard statute mile of 5,280 feet. InchToNautMile and NautMileToInch provide similar conversions based on the nautical mile of 6,080 feet.

```
FUNCTION InchToMeter (Inches: FLOAT): FLOAT;
FUNCTION MeterToInch (Meters: FLOAT): FLOAT;
```

These functions provide conversion between English and metric linear measurements. InchToMeter takes the parameter supplied in Inches and returns a value that represents the metric equivalent in meters. MeterToInch performs the reverse function.

```
FUNCTION SqInchToSqFeet (SqInches: FLOAT): FLOAT;
FUNCTION SqFeetToSqInch (SqFeet: FLOAT): FLOAT;
FUNCTION SqInchToSqYard (SqInches: FLOAT): FLOAT;
FUNCTION SqYardToSqInch (SqYards: FLOAT): FLOAT;
FUNCTION SqInchToSqMile (SqInches: FLOAT): FLOAT;
FUNCTION SqMileToSqInch (SqMiles: FLOAT): FLOAT;
FUNCTION SqInchToAcre (SqInches: FLOAT): : FLOAT;
FUNCTION AcreToSqInch (Acres: FLOAT): FLOAT;
```

This group of functions provides conversion between units for measurements of area. Each function takes a measurement in square inches and returns the equivalent value in the specified unit of measurement, or takes the value in the specified unit and returns the equivalent in square inches. The square mile used in this conversion is based on the statute mile of 5,280 feet.

```
FUNCTION SqInchToSqMeter (SqInches: FLOAT): FLOAT;
FUNCTION SqMeterToSqInch (SqMeters: FLOAT): FLOAT;
```

SqInchToSqMeter and SqMeterToSqInch provide conversion of area measurements between English and metric units. Other metric units of square measurement, such as square centimeters or square kilometers can be easily derived from the square-meters value.

```
FUNCTION CuInchToCuFeet (CuInches: FLOAT): FLOAT;
FUNCTION CuFeetToCuInch (CuFeet: FLOAT): FLOAT;
FUNCTION CuInchToCuYard (CuInches: FLOAT): FLOAT;
FUNCTION CuYardToCuInch (CuYards: FLOAT): FLOAT;
```

These four functions provide conversions between cubic inches, cubic feet, and cubic yards for measurements of volume. The parameter passed to the function is converted to the required value and returned to the calling program.

```
FUNCTION CuInchToCuMeter (CuInches: FLOAT): FLOAT;
FUNCTION CuMeterToCuInch (CuMeters: FLOAT): FLOAT;
```

CuInchToCuMeter allows volumes given in English units to be converted to their metric equivalent. CuMeterToCuInch performs the converse function.

```
FUNCTION FluidOzToPint (FluidOz: FLOAT): FLOAT;
FUNCTION PintToFluidOz (Pints: FLOAT): FLOAT;
FUNCTION FluidOzToGals (FluidOz: FLOAT): FLOAT;
FUNCTION GalsToFluidOz (Gals: FLOAT): FLOAT;
FUNCTION FluidOzToImpPint (FluidOz: FLOAT): FLOAT;
FUNCTION ImpPintToFluidOz (ImpPints: FLOAT): FLOAT;
FUNCTION FluidOzToImpGals (FluidOz: FLOAT): FLOAT;
FUNCTION ImpGalsToFluidOz (ImpGals: FLOAT): FLOAT;
```

For liquid volume measurements, these functions provide conversion between fluid ounces, pints, and gallons.

The first four functions are based on the standard American system of 16 fluid ounces per pint. The remaining functions use the Imperial system of measurement in which a pint consists of 20 fluid ounces; this system is used in most English-speaking countries that were once part of the British Commonwealth, such as Australia and New Zealand.

```
FUNCTION FluidOzToCuMeter (FluidOz: FLOAT): FLOAT;
FUNCTION CuMetersToFluidOz (CuMeters: FLOAT): FLOAT;
```

These two functions convert between fluid ounces and the metric system of liquid measurement. Cubic centimeters and liters can be derived from cubic meters.

```
PROCEDURE OunceToLbOz (Ounces: FLOAT; VAR lb, oz: FLOAT);
FUNCTION  LbOzToOunce (lb, oz: FLOAT): FLOAT;
```

For applications where it is required to enter or print weights as pounds and ounces, these two routines provide conversion to or from the total number of ounces.

OunceToLbOz takes the value passed in Ounces and returns the equivalent weight as pounds and ounces in lb and oz respectively. LbOzToOunce takes separate pound and ounce measurements and returns a value that represents the total weight in ounces. Note that the function will correctly handle a value of oz that exceeds 16.

```
FUNCTION OunceToTon (Ounces: FLOAT): FLOAT;
FUNCTION TonToOunce (Tons: FLOAT): FLOAT;
FUNCTION OunceToLongTon (Ounces: FLOAT): FLOAT;
FUNCTION LongTonToOunce (LongTons: FLOAT): FLOAT;
```

The first two of these functions provide conversion between ounces and the standard American ton of 2,000 pounds. (This is also known as a short ton.) The second two functions convert between ounces and the long ton of 2,240 pounds. This is the "standard" ton in most British Commonwealth countries.

```
FUNCTION OunceToGram (Ounces: FLOAT): FLOAT;
FUNCTION GramToOunce (Grams: FLOAT): FLOAT;
```

OunceToGram and GramToOunce provide conversion between the basic measurements of weight in the English and metric systems. Other metric derivatives, such as kilograms, are easily obtained from the basic gram.

4.
UNIT TIME

Many programs require the use of dates and times in some form. Such programs range from simple disk utilities to full accounting systems.

Turbo Pascal's own date and time functions are limited to the setting or retrieving of the system clock and the date and time of a disk file. This unit provides many more routines which will be found useful for the input, output, and manipulation of dates and times.

GLOBAL DECLARATIONS

The unit defines four new types:

```
DateString      = STRING[9];
TimeString      = STRING[13];

DateRec         = RECORD
                        M, D:   BYTE;
                        Y:      WORD;
                  END;

TimeRec         =  RECORD
                        H, M, S:  BYTE;
                   END;
```

The first two definitions are used to return strings from the unit's functions. The second two provide a convenient way to store a date or time in a single, three-field variable. The M, D, and Y fields of type DateRec hold the month, day, and year, respectively. Note that the field for the year is defined as type word and the unit always expects the year to be specified in full (e.g. 1990 should be assigned as 1990, not just 90). The H, M, and S fields of type TimeRec are used to hold the hours, minutes, and seconds of a time, respectively.

The following constants and typed constants are also defined in the interface of the unit. Their use is described with each routine when necessary.

```
DateFormNumeric       = 0;
DateFormAlpha         = 1;
DateFormMDY           = 2;
DateFormDMY           = 3;
DateFormLower         = 4;
DateFormZeroFill      = 8;
```

```
        FullDateFormMDY       = 0;
        FullDateFormDMY       = 1;

        TimeFormNormal        = 0;
        TimeFormNormalSec     = 1;
        TimeFormShort         = 2;
        TimeFormShortSec      = 3;
        TimeFormMilitary      = 4;
        TimeFormMilitarySec   = 5;
        TimeFormMilitaryHHMM  = 6;

        TimeFormat:       BYTE    = TimeFormNormal;
        DateFormat:       BYTE    = DateFormNumeric;
        FullDateFormat:   BYTE    = FullDateFormMDY;

        TimeDelimiter:    CHAR    = ':';
        DateDelimiter:    CHAR    = '/';

        TimeParseDelims:  TimeString  = ':., ' + #9;
        DateParseDelims:  DateString = '/-., ' + #9;

        TimeParseNow:     BOOLEAN = FALSE;
        DateParseToDay:   BOOLEAN = FALSE;
        DateParseCurYear: BOOLEAN = FALSE;
        DateParseCent21:  BYTE    = 0;
```

```
PROCEDURE CombineDateTime (VAR DtTm: DateTime;
                               Dt:   DateRec;
                               Tm:   TimeRec);

PROCEDURE SplitDateTime   (DtTm:   DateTime;
                           VAR Dt: DateRec;
                           VAR Tm: TimeRec);
```

The DOS unit supplied with Turbo Pascal defines a DateTime type which can be used in conjunction with the file date and time routines. These two procedures provide a means for combining separate date and time variables into one common record and for splitting the combined record into separate date and time records.

CombineDateTime takes the supplied date and time in Dt and Tm and returns the variable DtTm with all six fields set. SplitDateTime takes the values supplied in DtTm and returns with Dt and Tm set to the specified date and time respectively.

Note that no error checking is performed for out of range values. Since the fields of the DateTime type are defined as being of type WORD, then any data in the high-order byte will be lost when passed to SplitDateTime.

```
PROCEDURE GetToDay (VAR Dt: DateRec);
```

GetToDay simply returns the current system date in the record Dt. If, for example, the system date was December 15, 1990, then Dt would have its three fields, M, D, and Y, set to 12, 15, and 1990 respectively.

```
PROCEDURE GetTimeNow (VAR Tm: TimeRec);
```

This procedure sets the three fields of Tm the the current system time. All the time-handling routines assume that the hours field, H, holds a value of 0 (midnight) through 23 (11 p.m.). The minutes and seconds fields, M and S, may hold a value of up to 59.

```
PROCEDURE GetDateTime (VAR DtTm: DateTime);
```

This procedure allows the current date and time to be obtained from the system clock with a single subroutine call. The six fields of DtTm are set to the current values for year, month, day, hours, minutes, and seconds.

```
FUNCTION DateValid (Dt: DateRec): BOOLEAN;
```

DateValid examines the three fields within the supplied parameter Dt and returns true if they form a valid date.

Only dates in the range January 1, 1900 through June 6, 2079 are accepted as valid (i.e. the range of dates that can be represented as a two-byte word). The month field, M, must be 1 through 12, and the day field, D, must be 1 through 28, 29, 30, or 31 as appropriate to the month. A date of February 29 will only be accepted as valid if the year field also specifies a leap year.

```
FUNCTION TimeValid (Tm: TimeRec): BOOLEAN;
```

TimeValid checks the three fields of the variable passed as parameter Tm and returns true if they form a valid time. If the time in Tm is in any way invalid (minutes or seconds greater than 59 or hours greater than 23) then the function returns a result of false.

```
FUNCTION DateTimeValid (DtTm: DateTime);
```

DateTimeValid combines the actions of the DateValid and TimeValid functions, allowing a validity check to be performed on a variable of type DateTime. If the high-order byte of any supplied field is not zero, the function returns false, otherwise the date and time are passed to the DateValid and TimeValid functions for checking. A value of true will be returned only if both the date and time are valid.

```
PROCEDURE WordToDate (w: WORD; VAR Dt: DateRec);
```

WordToDate performs the opposite conversion to DateToWord. The date represented by the value supplied in w is converted to a three-field date in parameter Dt.

```
FUNCTION DateToWord (Dt: DateRec): WORD;
```

This function allows a date in the form of a record to be converted to a simple integer. DateToWord assumes that 0 represents January 1, 1900, 1 represents January 2, 1900, and so on. The two-byte value can, therefore, represent any date from January 1, 1900 to June 6, 2079.

There are two advantages to representing a date in this way. First, storage requirements are reduced when many dates must be handled because a variable of type DateRec requires four bytes. Second, it is easy to perform calculations by simple addition and subtraction. To do this with a variable of type DateRec would require careful consideration with regard to month end, year end, and leap years. The word representation of the date accounts for all 28-, 29-, 30-, or 31-day months (e.g. 33,052 is June 30, 1990 and 33,053 is July 1, 1990).

Note that no error checking is performed on the supplied date. If the date in parameter Dt is invalid in some way DateToWord will still return a value but it will represent some other date entirely.

FUNCTION LeapYear (Y: WORD): BOOLEAN;

This function checks the year supplied as parameter Y and returns true if it is a leap year or false if it is not. As with the other date functions, Y must specify the year in full (e.g. 1990 must be passed as 1990, not 90).

FUNCTION TimeAP (Tm: TimeRec): TimeString;

This function simply returns a four-character string of 'a.m.' if the time in Tm is before noon or 'p.m.' if the time in Tm is after noon.

PROCEDURE AdjustDate (VAR Dt: DateRec; n: INTEGER);

In order to add or subtract a given number of days from a date specified as a variable of type DateRec, three steps are necessary. First, the date must be converted using the DateToWord function. Second, the number of days adjustment required is added or deducted. Third, the integer must be converted back to a three-field record using WordToDate.

This procedure combines these three steps into one routine, making such adjustments simpler in your main program. The date variable to be adjusted should be passed as Dt and the number of days adjustment, plus or minus, should be passed a parameter n.

Note that no range checking is performed on the date, so trying to adjust to before January 1, 1900 or after June 6, 2079 will produce undesired results. Given the range of dates that this unit can handle, this should not be a problem in most applications.

PROCEDURE AdjustTime (VAR Tm: TimeRec; n: LongInt);

AdjustTime takes the time passed in Tm and adjusts it by the number of seconds passed in parameter n. Positive values adjust the time forward; negative values adjust the time backward.

Note that no indication is given if the adjustment causes the time to pass midnight in either direction. If you must account for the date as well as the time, use the AdjustDateTime function.

PROCEDURE AdjustDateTime (VAR DtTm: DateTime; n: LongInt);

This routine works in a similar way to AdjustTime, but also provides for adjustment of the date if the time adjustment causes midnight to be passed in either direction. The date and time are returned in DtTm after the necessary alteration has been made.

Note that no range checking is performed on any of the fields, so invalid dates and times supplied to AdjustDateTime will give meaningless results. The date range limitations listed under AdjustDate also apply.

PROCEDURE SetLastDay (VAR Dt: DateRec);

SetLastDay will be found useful in many accounting programs which require an end-of-month date.

The month and year fields of the date variable passed as parameter Dt should be set to the required month. The day field, may have any value upon entry. SetLastDay will return with the month and year fields unchanged, but will have adjusted the day field to the last day of the specified month. (The year field is required only for determining whether February should have 28 or 29 days.)

FUNCTION DayOfWeek (w: WORD): BYTE;

This function should be passed a date value as parameter w. The value must be in the form provided by the DateToWord function.

DayOfWeek returns a number that represents the day of the week for the specified date. Only a value of 0 through 6 will be returned:

0 = Sunday
1 = Monday
2 = Tuesday
3 = Wednesday
4 = Thursday
5 = Friday
6 = Saturday

FUNCTION DayOfWeekStr (d: BYTE): DateString;

This function converts the day-of-week value (described above) to a string containing the name of the day. The returned string is variable in length, to match the number of characters in the name, and each name is in mixed case.

If the value supplied in parameter d is greater than 6, the function returns a null string.

FUNCTION MonthStr (M: BYTE): DateString;

This function works in a similar way to the DayOfWeekStr function and returns the name of a month when given its number. The returned string is variable in length, to match the number of characters in the full name of the month, and the names are in mixed case.

If the supplied parameter is out of range, the function returns a null string.

FUNCTION DayOfMonthStr (D: BYTE): DateString;

This function returns a three- or four-character string based upon the value passed in parameter D. The string consists of the number followed by the appropriate suffix (i.e. 1st, 2nd, 3rd, 4th, etc.).

If D is out of range, a null string is returned.

FUNCTION DateStr (Dt: DateRec): DateString;

DateStr takes a date in the parameter Dt and formats it into a string suitable for display or printing.

There are two basic formats for the string - numeric and alphanumeric. The default setting is numeric and DateStr will return a string of eight characters in the form 'mm/dd/yy' or 'dd-mm-yy' depending on whether your system configuration is set to American or British format with a COUNTRY line in CONFIG.SYS. For example, a date of December 15, 1990 will be returned as '12/15/90' or '15-12-90'. Note that the year appears in abbreviated form as just the last two digits.

The delimiter separating the month, day, and year may be changed by assigning a character to the global variable DateDelimiter. Assigning '.' to DateDelimiter, for example, would cause future calls to DateStr to return '12.15.90' or '15.12.90' as appropriate. The character assigned to DateDelimiter may be changed at any time; the assignment stays in effect until another assignment to the variable is made. The unit initializes the delimiter to a "/" or "-" for American and British formats respectively. Note that the country setting for DateStr is determined at run-time, not compile-time, so any program using DateStr will always use the current country setting.

Another global variable, DateFormat, is used to determine the way in which DateStr builds the string it returns. The values used are listed in the introduction to this unit.

DateFormat is initialized to DateFormNumeric, causing DateStr to return a string of the form described above. Assigning DateFormMDY will cause DateStr to always return a date in American format, even if the system configuration is set to British date format. Similarly, assigning DateFormDMY will cause DateStr to always return a date in British format, regardless of the country setting in the system's configuration file.

Setting DateFormat to the constant DateFormAlpha will cause further calls to DateStr to return a nine-character string of the form 'dd/mmm/yy' or 'dd-mmm-yy'. The date of December 15, 1990 shown above, for example, might be formatted as '15-DEC-90'. (The separators would be replaced with whatever character, if any, had been assigned to DateDelimiter.)

Two further options may be used in conjunction with the numeric and alphanumeric formats. The values DateFormZeroFill and DateFormLower may be added to any of the other values. The former causes a leading blank in the string to be replaced by a zero. For example, a date of January 1, 1991 would be returned as '01/01/91' instead of ' 1/01/91'. Note that without the zero-fill option single digit numbers start with a space. This is done to ease alignment of printed columns. A similar result is obtained by adding DateFormZeroFill to the alphanumeric format: ' 1-JAN-91' would be replaced with '01-JAN-91'.

The second option is applicable only to the alphanumeric format and causes the month abbreviation to be printed in mixed upper and lower case instead of all capitals. DateStr would return '01-Jan-91' instead of '01-JAN-91'.

A typical program sets DateFormat and DateDelimiter to the required values once and then leaves them alone.

FUNCTION FullDateStr (Dt: DateRec): STRING;

FullDateStr provides a similar facility to that of DateStr, but returns a longer string containing the name of the month and the year in full.

FullDateStr may be set to format a date in two different ways. The default setting is 'month dd, yyyy'; a typical example of formatted output is 'January 2, 1991'.

Just as the global variable DateFormat is used to control the way in which DateStr presents its output, so FullDateFormat controls the way in which FullDateStr presents output. The constant FullDateFormMDY selects the default option described above. Assigning the constant FullDateFormDMY to FullDateFormat causes all future calls to FullDateStr to return a string of the form 'ddxx month yyyy', where "xx" represents the date suffix, as returned by the DayOfMonthStr function. The date shown in the above example would be formatted as '2nd January 1991'.

Note that the settings of DateFormat, DateDelimiter, and country selection have no effect on the FullDateStr function.

FUNCTION TimeStr (Tm: TimeRec): TimeString;

TimeStr converts the time passed as parameter Tm to a string suitable for printing or display.

There are several ways in which the output of TimeStr may be formatted and these are determined by the value set in the global variable TimeFormat. Several constants are provided by the unit to enable this setting to be made.

The default format, TimeFormNormal, causes the time to be formatted as 'hh:mm x.m.', where "x.m." is either "a.m." or "p.m." as appropriate. The hours field is converted to conventional 12-hour format and numbers less than 10 have a leading space to ease alignment of columns. Typical returned strings would be ' 2:35 p.m.' and '12:06 a.m.'. Note that the seconds field is ignored. If the system running the program is set to British configuration with a COUNTRY line in CONFIG.SYS, then the colon separator is replaced with a period, giving output such as ' 2.35 p.m.' and '12.06 a.m.'.

The seconds field can be added to the formatted string by the assignment of the constant TimeFormNormalSec to TimeFormat. A typical returned string would be ' 8:05:30 p.m.'.

A shortened form of the time is available by assigning TimeFormShort or TimeFortShortSec to TimeFormat. These options are similar to TimeFormNormal and TimeFormNormalSec but omit the "a.m." or "p.m." indicator. Typical returned strings would be ' 8:05' and ' 8:05:30' respectively.

Setting TimeFormat to the constant TimeFormMilitary causes TimeStr to return a string in military time format 'hh:mm'. If the constant TimeFormMilitarySec is assigned to TimeFormat, the seconds field is added. Examples of military format with and without seconds display would be '20:05' and '20:05:30' respectively.

The final option, TimeFormMilitaryHHMM, causes TimeStr to return a four-character string without the hours/minutes separator. For example, 8:25 p.m. would be returned as '2025' and 4:45 a.m. would be formatted as '0445'.

To clarify the way in which TimeStr works, the following examples show the output for each format when the function is given the time 9:10:50 p.m.:

```
' 9:10 p.m.'        (TimeFormNormal)
' 9:10:50 p.m.'     (TimeFormNormalSec)
' 9:10'             (TimeFormShort)
' 9:10:50'          (TimeFormShortSec)
'21:10'             (TimeFormMilitary)
'21:10:50'          (TimeFormMilitarySec)
'2110'              (TimeFormMilitaryHHMM)
```

One other adjustment to the formatted output is possible. The separating colon or period can be replaced with another character by making an assignment to the global variable TimeDelimiter. This separator appears between the hours and minutes fields unless TimeFormMilitaryHHMM is selected. The separator is also used between minutes and seconds if the selected format displays seconds.

```
FUNCTION DateParse (s: STRING; VAR Dt: DateRec): BOOLEAN;
```

Many programs require the user to input a date using the console and this function is intended to make such input easier.

DateParse attempts to convert the string supplied as parameter s to a date in record Dt. If the conversion succeeds a result of true is returned; if the conversion fails in any way (including invalid dates such as February 29, 1991) a value of false is returned.

DateParse is very flexible with regard to the format of the input string. The string should consist of three distinct parts: month, day, and year. The month may be a number (1 through 12) or an abbreviation of the month's name. DateParse only requires the first three letters of the month and ignores the rest of the word. Either upper or lower case may be used. The day and year must be numeric; the year may be specified in full or abbreviated form (e.g. 1990 may be "1990" or just "90"). Note that the date must be within the range January 1, 1900 through June 6, 2079 for DateParse to return true.

Each part of the date must have a delimiter to separate it from the other parts, and when using numeric representation of the month the parts must be in the correct sequence. This sequence must be month, day, year if DateFormMDY is set or day, month, year if DateFormDMY is set. When using the default DateFormNumeric, the sequence is determined by the country setting in the system's configuration file, as with DateStr. The following are all acceptable strings using the American date format:

```
1/2/91          (January 2, 1991)
12.17.90        (December 17, 1990)
10-23-1990      (October 23, 1990)
7 4 91          (July 4, 1991)
09, 30,  90     (September 30, 1990)
11/20. 1990     (November 20, 1990)
```

If the British date format is selected with DateFormDMY or from the CONFIG.SYS file, the following strings would be accepted:

```
30 12 90        (December 30, 1990)
7/4/90          (April 7, 1990)
2, 1, 1991      (January 2, 1991)
17-12/1990      (December 17, 1990)
```

Note that several delimiters are permissible and may even be mixed, as the last example shows. Spaces or tabs may follow a delimiter and may themselves be used as delimiters.

If the alphabetic representation of the month is used, the string may be in the order month, day, year, or day, month, year. The following are all acceptable dates, whichever date format is currently selected:

```
January 2,91     (January 2, 1991)
DEC/17/90        (December 17, 1990)
23-Oct-1990      (October 23, 1990)
4 July, 91       (July 4, 1991)
sep.30.90        (September 30, 1990)
20 nov 90        (November 20, 1990)
```

To clarify the way in which DateParse operates, the following examples show an invalid string which will cause DateParse to return a result of false (American MDY format is assumed):

```
9/31/90          (There is no September 31)
1jan 90          (No day/month delimiter)
fe 23 91         (Month abbreviation too short)
14,7,1990        (There is no fourteenth month)
6 /30/91         (Spaces cannot precede delimiter)
1990 Apr 3       (Year must be last item)
```

The examples show that several delimiters may be used to separate the month, day, and year fields. The global variable DateParseDelims contains the acceptable delimiter characters and is initialized to the string shown in the introduction to this unit. The accepted delimiters may be changed by assigning a new string to DateParseDelims. The assignment of '/- ' would cause DateParse to accept only the three characters listed as delimiters. The strings '1/2/90', '12-30-90', and '17 DEC 90' would, therefore, be acceptable, but '1.2.90' and 'Dec 30, 90' would not. Note that DateParse will always accept any number of blanks (spaces or tabs) after a delimiter; any number of blanks may also precede or follow the string as a whole.

It is possible to have DateParse automatically supply the current year if none is specified by setting the global variable DateParseCurYear to true. Strings supplied to the function can then consist of just the month and day, and DateParse will automatically obtain the year from the system clock. Input which specifies a year will be treated in the usual way. DateParseCurYear is initialized to false.

Setting the global variable DateParseToDay to true also modifies the way in which DateParse interprets the input string. With this setting, the function will accept a null-string input (or a string which consists entirely of blanks) and return true with the current system date. This feature

is useful in applications where the date being entered by the user is frequently the current date. DateParseToDay is initialized to false.

The final option available with DateParse uses the global variable DateParseCent21 and relates to the use of dates in the 21st century. By default, any year entered as two digits is assumed to be in the 20th century, so that year "xx" becomes "19xx." Assigning a value to DateParseCent21 causes DateParse to assume that two-digit years less than the specified value are to be treated as "20xx." The following examples show how DateParse interprets input when DateParseCent21 is set to 45:

```
1/1/99          (January 1, 1999)
7/10/62         (July 10, 1962)
1 1 00          (January 1, 2000)
7 Dec 41        (December 7, 2041)
7 Dec 1941      (December 7, 1941)
```

The last example shows that a year entered as four digits will always be interpreted exactly as entered. DateParseCent21 is initialized to zero; if no assignment is made by your program then all dates from 2000 onward must be entered as four digits.

```
FUNCTION TimeParse (s: STRING; VAR Tm: TimeRec): BOOLEAN;
```

It is sometimes necessary for a program to accept the input of a time from the console or from the command line that started the program. This function will parse such input and attempt to convert it into a variable of type TimeRec.

TimeParse converts a string supplied as parameter s to a time value in the three fields of parameter Tm. If the conversion succeeds a result of true is returned; if the conversion fails due to invalid input, a value of false is returned.

TimeParse is flexible with regard to the format of the input string. Two distinct parts are required to specify the hours and minutes and a delimiter must separate the fields. Times may be entered in normal or military format; in the case of the former an "a" or "p" after the minutes field indicates a.m. or p.m. The following are examples of acceptable input:

```
10:20           (10:20 a.m.)
5.50            (5:50 a.m.)
7 25a           (7:25 a.m.)
```

3.10 P	(3:10 p.m.)
11. 0 pm	(11:00 p.m.)
21:40	(9:40 p.m.)
0.15	(12:15 a.m.)

TimeParse will reject any input which has an "a" or "p" if the hours field is less than 1 or greater than 12. Similarly, if the minutes field is out of range, TimeParse will return a value of false to the calling routine. Note that there may be any number of blanks (spaces or tabs) after the delimiter and that blanks themselves may be a delimiter.

The above examples show the input of just hours and minutes; the seconds field of Tm is set to zero in such cases. It is also possible to specify the seconds field in the input string by the addition of a second delimiter. Examples of acceptable input are:

3:50:30	(3:50:30 a.m.)
20.10.15	(8:10:15 p.m.)
7, 4, 55P	(7:04:55 p.m.)
9,0,58 a	(9:00:58 a.m.)

The seconds field must come before any a.m. or p.m. indicator or it will not be recognized (TimeParse does not scan beyond an "a" or "p").

To clarify the way in which TimeParse operates, the following examples show an invalid string which will cause the function to return a result of false:

3:62	(Minutes out of range)
0130	(No hours/minutes separator)
14.00p	(Mixed p.m. and military time)
2a	(No minutes field)
5 :35	(Space before delimiter)

The characters acceptable as delimiters may be changed by assigning a new string to the global variable TimeParseDelims. The initial string is shown in the introduction to this section. Assigning ':' would set TimeParse to accept only the two characters specified as delimiters. The strings '3:45p' and '15 21' would, therefore, be acceptable, but '3.45p' and '15,21' would not.

Normally, TimeParse will not accept a null input and will return false. Setting the global variable TimeParseNow to true causes TimeParse to accept a null-string (or a string of blanks) and return true with the current time from the system clock. TimeParseNow is initially set to false when your program starts.

5.
UNIT STDERR

Turbo Pascal's Write and WriteLn procedures normally send output to the operating system's standard output device. This allows the pipe and redirection commands to be used on the command line to send output to a file, printer, or another program.

Many utilities can be written to take advantage of this redirection, which provides versatility without the need to complicate the program. Most utility programs have the need to produce an error message on occasions, however, and if this message is sent to the standard output device it is possible that it will be "lost." Although the message will appear in the redirected output (either on the printer, in a file, or as input to another program), it is usually far better to have the message sent to the console.

When a Turbo Pascal program uses the CRT unit, a standard Write or WriteLn procedure no longer sends output to the standard output device, but always sends it the the display console. This would provide one solution to the problem, but requires that the program specifically assigns and opens an output channel for all its regular output which must go to the standard output device (for full details consult your Turbo Pascal reference guide).

This unit offers a simple solution which avoids the need to use the CRT unit in such cases. Only one procedure is provided in the STDERR unit and it is used to send output to the standard error device, which is always the console display.

PROCEDURE WriteStdErr (s: STRING);

WriteStdErr takes the string passed as parameter s and sends it to the standard error device. A carriage return and line feed are also sent after the string. Because the standard error device is always the screen, any error messages presented using this procedure cannot be redirected and "lost" in a file.

Note that, unlike the standard Write and WriteLn procedures, WriteStdErr can only take a single string argument, so any numeric output must be converted to string format first.

While the function's primary purpose is that of presentation of error messages, WriteStdErr can be used to send any text to the screen. For example, a sign-on and copyright notice could be displayed using this procedure, ensuring that when the program's standard output is redirected the sign-on message is still sent to the console.

6.
UNIT CRTCLERR

The standard IBM PC operating system incorporates an error handler which is called whenever a critical error occurs. Critical errors are caused by such things as a printer out of paper, a communication port not ready to accept data, or a disk drive with an open door.

The critical-error handler presents an error message on the console, followed by "Abort, Retry, Ignore?" or "Abort, Retry, Fail?" depending on the version of DOS in use. The user may then decide whether to retry the operation (after correcting the cause of the error, by loading a printer with paper or inserting a disk, for example), abort the program, or allow the program to continue by informing it that the operation failed.

Turbo Pascal installs its own critical-error handler which overrides these DOS messages. If a Turbo Pascal program attempts to open a file on a drive which has no disk in it, the operation fails and returns a suitable result code in IOResult. In many programs this action is ideal, especially when the screen is not used in conventional teletype mode (i.e. the appearance of the DOS error message would disturb the screen layout).

For some programs, however, it is more desirable that the usual critical-error message is presented when such an error is encountered. Most disk utility programs would fall into this category. This unit enables the original critical-error handler to be restored by a Turbo Pascal program.

The CRTCLERR unit defines one new type: ErrorString. This is a string of up to 20 characters and is used to return an error message from the function CriticalErrorMsg.

If all that is required is for a program to use the operating system's critical-error handler throughout, then the only step necessary is to declare the CRTCLERR unit in the program's USES statement.

```
PROCEDURE CriticalErrorDOS;
PROCEDURE CriticalErrorTP;
```

These two procedures allow switching between Turbo Pascal's critical-error handler (CriticalErrorTP) and the operating system's handler (CriticalErrorDOS).

Any critical error which occurs after a call to one of these procedures will be handled by the specified routine. Note that declaring the CRTCLERR unit in a program's USES statement automatically sets critical-error handling to DOS, so it is only necessary to use these procedures if different parts of the same program must use different critical-error handlers.

```
PROCEDURE CriticalErrorOwn (ErrAddr: POINTER);
```

In some programs neither Turbo Pascal nor DOS can provide the correct type of critical-error handling. An example would be where the screen is divided into windows and critical errors must be presented to the user by way of a window.

In such cases, this procedure allows control to be passed to any routine when a critical error is detected. The value passed in parameter ErrAddr should be the address of your own critical-error handler routine.

The exact details of how to write your own critical-error handler are outside the scope of this manual; this procedure is provided for those programmers who are fully conversant with such matters. You should carefully study a good DOS manual before attempting to use this routine.

```
FUNCTION CriticalErrorMsg (n: BYTE): ErrorString;
```

This function is provided for use in your own critical-error handlers and may be freely called from within such a handler.

When DOS makes a call to the critical-error handler using interrupt 24 (hex.), the DI register holds a value which indicates the nature of the error. If the value in DI is passed to this function as parameter n, a string containing the appropriate error message is returned. If the number is not recognized, the message "Unknown error" is returned.

The codes and messages supported are:

00	Write-protect error
01	Unknown unit
02	Drive not ready
03	Unknown command
04	Data error
05	Bad request
06	Seek error
07	Unknown disk format
08	Sector not found
09	Printer out of paper
0A	Write fault
0B	Read fault
0C	General error

7.
UNIT ENHCON

Many programs written for the IBM PC need to manipulate the display in some way that is not catered for by the standard Pascal language. Turbo Pascal's CRT unit provides features designed specifically to take advantage of the PC's console, such as cursor positioning and reading a single key-stroke.

The ENHCON unit expands upon the facilities offered by Borland's CRT unit in several ways. First, a few extra routines are provided for controlling some low-level aspects of the display, such as adjusting the size of the cursor. Second, several functions are provided that allow strings and numbers to be edited on the display; these will be found to be useful again and again in many interactive programs that require input from the user.

Third, a complete set of routines for handling display windows is provided. An application program can define up to 255 different windows, each of which may set its own text attribute and cursor, and may define a border with header and footer text. The area of screen which is occupied by an opened window may be saved, thus enabling the original contents to be restored when the window is closed. It is also possible to remove a window from the screen but preserve its contents. These routines allow the simple development of programs which require status lines, "pop-up" menus, and so on.

Finally, the unit provides a simple way to implement on-line help systems for application programs. The help text may consist of up to 255 sections, each of which may have up to 1,000 lines of text. Each section name appears in an index, allowing the user to select the required topic.

This unit requires Borland's CRT unit in order to operate, but it is not necessary for your program to specifically include CRT in its USES statement, unless it also accesses routines in the CRT unit directly. If your program does use CRT, it is most important that you declare ENHCON after CRT in your USES statement (i.e. "USES CRT, ENHCON;" is acceptable but "USES ENHCON, CRT;" is not). The reason for this is that ENHCON "hooks" into some of CRT's routines in order to modify their operation.

You should be aware of the fact that the ENHCON unit accesses memory directly and makes extensive use of BIOS routines. This means that any program using this unit may not work on systems that do not have full PC compatibility. This is also true of any program that uses Turbo Pascal's own

CRT unit. As the programmer, you must make a trade-off between compatibility and the features you would like to implement.

DISPLAY ATTRIBUTE CONSTANTS

Turbo Pascal's CRT unit defines several text-attribute constants for color displays, but overlooks those for monochrome systems. The following constants may be be assigned to the CRT unit's TextAttr variable when using a monochrome display.

MonoNormal	Normal text
MonoUnderline	Underlined text
MonoIntense	Intensified text
MonoIntenseUL	Intensified and underlined
MonoReverse	Reverse video (dark on light)
MonoNone	Blank, or hidden, text

EXTENDED KEY HANDLING AND READKEY FUNCTION

The standard IBM PC keyboard provides a comprehensive set of extended keys - keys which do not have a regular ASCII code, such as Home, End, PgUp, PgDn, etc. Programs can make use of these function keys in many ways, but use of the standard ReadKey function in the CRT unit can make this application somewhat complex, because of the way in which it returns information about these extended keys.

Normally, keys which have a standard ASCII code, such as letters, numbers, carriage-return, and so on, cause ReadKey to return the appropriate ASCII code. Extended keys cause a zero, or null, to be returned and it is then necessary to call ReadKey a second time to get the scan code for the key.

To simplify keyboard input, ENHCON was designed to modify the behavior of ReadKey so that it only returns a single value for either normal ASCII characters or extended keys. All 256 possible values of a character-type variable are assigned in the IBM PC, the values 80 through FF hex. representing mathematical symbols, graphics, and foreign alphabets. Some programs may need to send one of these characters to the display, but it is highly unlikely that they would need to input such a character from the keyboard; standard keyboards can only generate these codes by the Alt-keypad method anyway.

ENHCON modifies ReadKey to use the values 80 through FF hex. to represent extended key-strokes. The assigned values are defined as constants in the interface section of ENHCON, and the identifiers may be used in your program. Note that an S, A, or C after "Key" represents Shift, Alt, or Ctrl, respectively (e.g. "KeyCHome" is the key-stroke Ctrl-Home, and "KeyAX" is the key-stroke Alt-X).

Constant	Value
KeyIns	80
KeyDel	81
KeyUp	82
KeyDown	83
KeyLeft	84
KeyRight	85
KeyHome	86
KeyEnd	87
KeyPgUp	88
KeyPgDn	89
KeyCLeft	8A
KeyCRight	8B
KeyCHome	8C
KeyCEnd	8D
KeyCPgUp	8E
KeyCPgDn	8F
KeyA0	90
KeyA1	91
KeyA2	92
KeyA3	93
KeyA4	94
KeyA5	95
KeyA6	96
KeyA7	97
KeyA8	98
KeyA9	99
KeyAHyphen	9A
KeyAEquals	9B
PoundSign	9C
KeySTab	9D
KeyCPrtSc	9E

Constant	Value
KeyF1	A0
KeyF2	A1
KeyF3	A2
KeyF4	A3
KeyF5	A4
KeyF6	A5
KeyF7	A6
KeyF8	A7
KeyF9	A8
KeyF10	A9
KeySF1	B0
KeySF2	B1
KeySF3	B2
KeySF4	B3
KeySF5	B4
KeySF6	B5
KeySF7	B6
KeySF8	B7
KeySF9	B8
KeySF10	B9
KeyCF1	C0
KeyCF2	C1
KeyCF3	C2
KeyCF4	C3
KeyCF5	C4
KeyCF6	C5
KeyCF7	C6
KeyCF8	C7
KeyCF9	C8
KeyCF10	C9
KeyAF1	D0
KeyAF2	D1
KeyAF3	D2
KeyAF4	D3
KeyAF5	D4
KeyAF6	D5
KeyAF7	D6
KeyAF8	D7
KeyAF9	D8
KeyAF10	D9

Constant	Value
KeyAA	E1
KeyAB	E2
KeyAC	E3
KeyAD	E4
KeyAE	E5
KeyAF	E6
KeyAG	E7
KeyAH	E8
KeyAI	E9
KeyAJ	EA
KeyAK	EB
KeyAL	EC
KeyAM	ED
KeyAN	EE
KeyAO	EF
KeyAP	F0
KeyAQ	F1
KeyAR	F2
KeyAS	F3
KeyAT	F4
KeyAU	F5
KeyAV	F6
KeyAW	F7
KeyAX	F8
KeyAY	F9
KeyAZ	FA

A call to ReadKey, therefore, causes a single value to be returned - either a standard ASCII code or one of the values listed above. For example, the Backspace key returns a value of 08 hex. (the ASCII code for Ctrl-H) and the F1 function key returns A0 hex. The Alt-keypad method for entering characters will only work for standard ASCII characters in the range 01 through 7E, and for the code 9C hex. (156 decimal).

Note that a constant, PoundSign, has been defined as the value 9C hex. British keyboards have the pounds-sterling sign where American keyboards have the "#" symbol, so this code is left unchanged to enable this symbol to be entered correctly on such systems.

The IBM PC keyboard offers four toggles: capitals, number, scroll, and insert lock, each changed by pressing the appropriate lock key. The first three simply change the status of the appropriate lock, but do not return any value to ReadKey. For some curious reason, pressing the Ins, or Insert, key not only toggles the lock status but also returns an extended key code.

A good many programs make use of the Ins key to switch between insert and overwrite modes, and use the appropriate lock-status flag to determine the current mode. This means that the program must disregard calls to ReadKey which return the Ins key's code. The enhanced version of ReadKey in ENHCON causes depressions of the Ins or Insert key to be ignored, relieving the programmer of this extra work. This facility can be overridden if desired, causing ReadKey to return a code of 80 hex. when Ins is pressed. The global boolean variable InsKeyEnable should be set true to cause the Ins key to be treated as a regular extended key, or false to cause ReadKey to ignore it. InsKeyEnable is set to false by default.

FUNCTION ColorDisplay: BOOLEAN;

This function returns true if the program is running on a system which has a color display or false if running on a monochrome system. If the computer has both a monochrome and a color display, the value returned indicates the type of the currently active display.

```
PROCEDURE GetMaxXY (VAR x, y: BYTE);
```

GetMaxXY sets the variables specified as parameters x and y to the number of columns and rows, respectively, of the current display mode. A standard monochrome display will return x as 80 and y as 25.

This routine will be found useful for console input and output routines which must work with a variety of different display adapters and modes.

```
FUNCTION GetDisplayPage: BYTE;
```

All of IBM's display adapters, except the basic MDA (monochrome), support multiple display pages. Turbo Pascal does not provide full support for this facility, but some applications may benefit from using two or more pages.

The current version of ENHCON does not provide page-switching support, but this function is used by some other routines and was included in the unit's interface on the grounds that it could be useful for developing such applications.

GetDisplayPage returns the number of the currently selected page, which will usually be zero.

```
FUNCTION GetDisplayBase: WORD;
```

Some programs need to access the display in a way which can only be achieved by directly storing values in the display memory. This function aids calculation of the display address by returning the base segment address for the currently selected display adapter and page.

GetDisplayBase will usually return B000 hex. for a monochrome display and B800 hex. for a color display, unless a page other than zero is selected. A description of the way in which display memory is used is beyond the scope of this manual; consult your system's documentation for details.

```
FUNCTION MaxCursorSize: BYTE;
```

The display adapters employed by the IBM PC allow the size of the cursor to be adjusted by software. The cursor size is set by specifying the starting and ending scan lines. The top line of a character position is line zero; the number

of lines per position is dependent upon the type of display adapter in use. The CGA color display has 8 scan lines; the EGA color and MDA monochrome adapters both have 14 lines.

This function checks the display type in use and returns a value that represents the highest-numbered scan line for the current mode. A CGA system causes the returned value to be 07 hex. and a monochrome or EGA color system sets the returned value to 0D hex.

Note that the new VGA color adapters have a mode which emulates the cursor of the CGA; the ENHCON unit automatically sets a VGA display to this mode upon initialization.

PROCEDURE SetCursor (size: WORD);

This procedure adjusts the cursor size to that specified by the parameter. Size is a two-byte value - the upper byte represents the starting scan line and the lower byte the ending scan line (the standard monochrome cursor of two scan lines, for example, is represented by the value 0B0C hex.).

If your program switches to a different display mode at any point in its operation, then the cursor size is reset to the default for the new display mode.

Note that if the cursor is hidden, SetCursor does not automatically make it visible again - it just sets the new size to be used when the cursor is re-enabled. Similarly, SetCursor cannot be used to set to disable-cursor bit - the procedure HideCursor should be used instead. You should consult your system's documentation for details before using this procedure if you are unfamiliar with the way in which the cursor can be manipulated.

FUNCTION GetCursor: WORD;

GetCursor provides the converse function to SetCursor and returns the size of the current cursor. Like SetCursor, it ignores the state of the cursor-hide bit and will return the cursor size as though the cursor were actually enabled (you should use the CursorHidden function to test the cursor-hide bit).

```
PROCEDURE HideCursor (hide: BOOLEAN);
```

The SetCursor and GetCursor routines allow the size of the cursor to be handled; HideCursor allows it to be hidden and restored.

Calling this procedure with parameter hide set true causes the cursor to be disabled by setting the hide-bit in the cursor control word. Calling with hide set false causes the cursor to be restored. A typical use would be to remove the cursor from the screen while presenting the user with a menu display.

Note that calling HideCursor does not affect the size of the cursor; the cursor will be restored to the same size it was before it was hidden, unless SetCursor was called in between.

If your program switches display modes by calling TextMode, then the cursor is automatically made visible and reset to the default size for the new mode.

```
FUNCTION CursorHidden: BOOLEAN;
```

This function checks the cursor-control word and returns true if the cursor is currently hidden or false if it is visible.

Manuals for the IBM BIOS sometimes list ways in which the cursor may be hidden, other than by setting the disable-cursor flag in the control word. All the cursor routines in this unit expect that the cursor will be hidden only by using the appropriate flag and CursorHidden looks only at this flag. So long as you only control the cursor by using the facilities provided by ENHCON, there will be no conflict.

```
PROCEDURE LineCursor;
PROCEDURE BlockCursor;
```

The cursor-control routines above allow for maximum flexibility when manipulating the cursor, but SetCursor requires careful handling of the cursor's size specification. Many programs require nothing more than the ability to select a line cursor, a block cursor, or no cursor.

These two procedures allow simple switching between a line or block cursor. A program may call LineCursor or BlockCursor without any consideration as to the type of display in use; ENHCON automatically determines the correct

values to pass to SetCursor for the requested size.

As with SetCursor, calling either of these routines does not automatically enable the cursor if it is hidden; the procedures simply adjust the size so that the new cursor will appear after a call to HideCursor to re-enable it.

INSERT/OVERWRITE CURSOR SWITCHING

It has become common practice to change the size of the cursor when the insert mode status is changed. The usual arrangement is to have a line cursor for overwrite mode and a block cursor for insert mode.

ENHCON allows this cursor switching to be easily accomplished; all that is necessary is to set the global boolean variable CursorInsert to true. The size of the cursor is updated whenever a call to ReadKey is made and it does not matter whether InsKeyEnable is set to true or false (see Extended key handling, above). You should note that the cursor switching will not take place if you call the standard input procedures Read or ReadLn. One of the main aims of the ENHCON unit was to simplify console input and output for the type of display-oriented programs that are so common today, so it is unlikely that the standard input procedures would be used.

Should you wish to disable the cursor switching for certain parts of your program, you may accomplish this by setting CursorInsert back to false. The cursor may be left as a line or a block, depending upon the state of the insert toggle at the time, so if you want the cursor to be set to a specific size you should call SetCursor, LineCursor, BlockCursor, or OrigCursor immediately after.

PROCEDURE OrigCursor;

The unit automatically saves the cursor size and hide status when your program is run and restores them when your program terminates. This ensures that cursor changes in your program are not carried back to the DOS prompt.

OrigCursor may be called to accomplish the same thing part way through your program. If you have switched display modes by calling TextMode, OrigCursor recalls the default cursor that was set up for the new mode.

```
FUNCTION CapsLock: BOOLEAN;
FUNCTION NumLock: BOOLEAN;
FUNCTION ScrollLock: BOOLEAN;
FUNCTION InsertLock: BOOLEAN;
```

These four functions allow your program to check the current state of each of the four keyboard toggles: capitals, number, scroll, and insert lock. Each function returns true if the appropriate toggle is set or false if it is clear.

InsertLock is used to check whether typed text should be inserted in a string or overwrite the current contents and will be found useful in developing editing routines (the source code for the EditString routine in this unit is a typical example). Scroll lock is used by some programs to change the action of cursor movement keys, a spreadsheet being a case in point. CapsLock and NumLock will probably be used only very occasionally, but were included for completeness.

```
PROCEDURE ForceInsert (Ins: BOOLEAN);
```

ForceInsert may be used to set the insert toggle to a known state, at the start of a program or editing routine, for example. If parameter Ins is true the toggle is set on (insert mode); if Ins is false it is set off (overwrite mode).

If the automatic insert/overwrite cursor switching is enabled, the cursor will be updated at the next call to ReadKey.

The inclusion of procedures to force the capitals, number, and scroll lock toggles to a known state was considered during the development of this unit. The decision was made not to provide them, however, because of the difficulty of keeping keyboard indicators synchronized with the toggle. Many keyboards provide three lock indicators which show the current state of the capitals, number, and scroll toggles. Forcing specific values into these toggles usually does not update the indicators, which results in the light being on when the lock is off, and vice versa.

```
PROCEDURE FlushKB;
```

The IBM PC buffers keyboard input to allow type-ahead, although the size of the buffer is extremely limited. There are many times in a program when it is not desirable to allow such buffering; a typical example is a check-point which asks

"Are you sure?"

Calling FlushKB causes the contents of the keyboard buffer to be flushed, thereby preventing previous key-strokes from being read by the next call for input. FlushKB should be called immediately before the input routine that must not allow buffered input to be used.

EDITING ROUTINES

The ENHCON unit provides five functions that allow different types of data to be displayed and edited on the screen: EditString, EditInt, EditReal, EditDate, and EditTime.

Each routine allows the programmer to position the cursor at the start of an editing field on the display, highlight the field, and then accept user input. A parameter, form, is passed to the appropriate routine and this controls the format of the edit. Each field of this record (of type EditFormatRec) is described in detail in the following sections.

```
EditFormatRec  =  RECORD
                    Attribute:      BYTE;
                    StartChar,
                    EndChar:        CHAR;
                    MarkerAttr:     BYTE;
                    AllowChars,
                    ExitKeys:       CharSet;
                    EditKey,
                    RestoreKey,
                    AbortKey:       CHAR;
                    NumFormat:      STRING[12];
                    SignalError:    SignalErrorProc;
                    Flags:          WORD;
                  END;
```

```
FUNCTION EditString (form:      EditFormatRec;
                     VAR s:     STRING;
                     width:     BYTE): CHAR;
```

This function may be called to edit any string, subject to the string having a maximum length of 80 characters (i.e. the largest field that will fit across a standard display screen).

Before calling EditString the cursor should be positioned at the first character of the desired position of the editing

field on the display. The string to be edited should be supplied as parameter s and its maximum length should be passed as parameter width. This value determines the number of characters that will be used on the display for the editing field. It is the programmer's responsibility to ensure that the cursor position and field width are consistent, so that the field does not run off the edge of the display.

MARKING THE FIELD

There are two ways in which the editing field may be highlighted on the screen. If the Attribute field of form is set to a non-zero value, then that value is used as a text attribute. If, for example, form.Attribute is set to MonoReverse, then the editing field is displayed in reverse video. Note that the whole field is highlighted to the specified field width, even if the string, s, is shorter in length. If Attribute is set to zero, EditString does not change the text attribute and the field will be displayed in whatever value was in TextAttr at the time EditString was called.

The alternative way to mark the editing field on the screen is to have a marker character at each end of the field. Setting StartChar in form to a character other than NUL (i.e. #0) causes the specified character to be displayed immediately to the left of the first character of the editing field. In a similar way, EndChar controls the display of a character immediately after the last position of the field. A typical program would use arrow characters to point to the field being edited (e.g. character codes #16 and #17). If you use a start- or end-of-field marker character, you must ensure that space exists on the screen for these characters. (You should not try to use StartChar if the editing field starts in the first display column, for instance.) Whenever form specifies that a start- or end-of-field marker is to be used, EditString uses the value supplied in field MarkerAttr as the text attribute for the markers. If MarkerAttr is zero, the characters are displayed with whatever value was set in TextAttr when EditString was called.

Most programs will use only one of the above methods to mark the field being edited, but EditString allows both to be used simultaneously if desired. If no highlighting of the field is required, just set StartChar and EndChar to NUL and Attribute to zero. This disables the marker characters and causes the field to be displayed in the current text attribute, as described above.

BASIC EDITING

EditString automatically uses six keys to permit editing: Home, End, cursor left, cursor right, Backspace, and Delete. The use of these keys is fairly obvious, but they are summarized here for sake of completeness.

Home	Move to start of string
End	Move to end of string
Cursor Left	Move left one character
Cursor Right	Move right one character
Backspace	Delete character to left of cursor
Delete	Delete character at cursor

The exact action of some of these keys may vary depending upon the configuration set up in form. Such details are listed where appropriate in the following sections.

Five of the fields in the parameter form allow the various editing keys to be defined. These fields are AllowChars, ExitKeys, RestoreKey, AbortKey, and EditKey. AllowChars is of type CharSet, a set of characters, and is used to set which characters may be entered in the string. For example, if the string being edited should only be allowed to hold capital letters and numbers, the set ['A'..'Z', '0'..'9'] should be assigned to form.AllowChars; any other keys pressed during the edit will then be ignored by EditString. The interface section of ENHCON defines a set called StandardChars, which will be found useful for general string editing. Assigning this set to AllowChars allows any of the standard printable characters to be entered (i.e. ASCII characters 32 through 126 decimal).

ExitKeys is also defined as type CharSet and is used to hold the keys which will terminate the edit and cause EditString to return to its calling program. In many programs, assigning the simple set [CR] will be sufficient, allowing the carriage-return, or Enter, key to signal that editing has been completed. Other applications may require that several keys are available to exit from EditString. A data entry screen which uses the cursor keys to move from one field to another, for example, would require ExitKeys to be set to a larger set (e.g. [CR, KeyLeft, KeyRight, KeyUp, KeyDown, KeyF10]). Note that EditString uses the enhanced version of ReadKey, so all keys are defined as a single character value.

Whenever a key that is listed in ExitKeys is pressed, EditString updates the edited string (parameter s) to the current contents of the edit field and returns to the calling program. The character-type result of EditString is the character code of the key that was pressed to terminate the

edit. Using the example shown in the last paragraph, a code of 0D hex. would be returned if the edit was terminated by the Enter key, but A9 hex. would be returned if the user finished the edit by pressing F10. This feature allows the calling routine to determine which key terminated the edit so that it may take the appropriate action (moving to the next or previous field, for instance).

Note that it is quite permissible to define a key in both AllowChars and ExitKeys. If this is done, pressing the key will first enter the character into the string being edited, then cause EditString to terminate as usual.

Before returning, EditString removes any highlighting of the editing field. The field itself is re-written in whatever text attribute was set when EditString was called. If start- or end-of-field marker characters were used, they are replaced by whatever character was on the screen previously. Finally, the cursor is left at the first position of the editing field (i.e. the same place as when the edit function was called).

RESTORE AND ABORT

EditString makes a copy of the string s upon entry and allows it to be restored at any time during the edit by the user pressing a single key. The key to be used for this purpose should be assigned to the RestoreKey field of form. If, for example, RestoreKey is set to KeyF2 (value A1 hex.), then pressing F2 will throw away any changes made to the string since EditString was called; the cursor is returned to the first character in the field so the user may start the edit again. The restore feature may be disabled by assigning NUL to form.RestoreKey.

In a similar way, a key may be assigned to AbortKey. Pressing the abort key during an edit causes the original string to be restored and the edit routine to terminate. EditString returns the character code for the abort key so that the calling routine may detect the abort and act accordingly. If the abort facility is not required, it may be disabled by assigning NUL to form.AbortKey.

It is acceptable for RestoreKey and AbortKey to hold the same value so that the same key may be used for both purposes. The restore action is performed whenever the key is pressed, but EditString only terminates if the key was pressed when the string had not been edited. For example, assume that RestoreKey and AbortKey are both set to ESC. If EditString is called and the Escape key is pressed straight away, the string has not been edited so the routine terminates and returns the ESC code. If the string is edited

and then Escape is pressed, the original string is restored and the edit is allowed continue. (Pressing Escape a second time, with no other intervening keys, would then terminate the edit.)

FLAGS

The Flags field of form contains 16 flags, not all of which are used in the current version of ENHCON. The interface section of the source code lists several constants which may be used to build the flags value for a format. These values are summarized here and each flag is described in detail below.

EdFlagFlushKB	Flush keyboard buffer before edit
EdFlagInsert	Select insert mode
EdFlagForceIns	Force insert toggle to selected state
EdFlagInsStat	Allow insert/overwrite switching
EdFlagFirstClr	First character clears field
EdFlagEdKeyExit	Allow standard edit key to terminate
EdFlagHideCursor	Hide cursor before edit
EdFlagTrimL	Remove leading blanks
EdFlagTrimR	Remove trailing blanks
EdFlagPadL	Pad with leading blanks
EdFlagPadR	Pad with trailing blanks
EdFlagUpper	Force upper case

The constants for the required flags should be added together when setting the Flags field of form. Example:

```
Flags := EdFlagFirstClr + EdFlagHideCursor + EdFlagInsStat;
```

sets the three flags specified, leaving all other flags clear.

INSERT AND OVERWRITE MODES

The three flags EdFlagInsert, EdFlagForceIns, and EdFlagInsStat are used to control the way in which EditString uses insert and overwrite modes. If all three flags are clear, overwrite mode is selected. In this mode typed characters overwrite existing characters in the string; pressing Backspace moves the cursor to the left and replaces the character there with a space - it does not "close up" the string.

If EdFlagInsert is set, insert mode is selected. Typed characters are inserted at the cursor, pushing everything beyond the cursor one position to the right (the last character of the field is lost). Pressing Backspace while in

insert mode deletes the character to the left of the cursor and moves everything beyond the cursor back one position.

The two settings described above fix EditString to one mode or the other; they do not allow the Insert key to be used to switch modes. If EdFlagInsStat is set, EditString checks the current Insert lock status each time a character is typed. This allows the Insert key to be used to toggle between insert and overwrite modes.

Setting EdFlagForceIns causes the insert lock to be forced to a known state at the start of the edit, the state used being that set in EdFlagInsert (i.e. EdFlagForceIns and EdFlagInsert forces insert mode, but EdFlagForceIns alone forces overwrite mode). EdFlagInsStat must still be set for the Insert key to be used to toggle insert mode; EdFlagForceIns simply ensures that the toggle is set to a specific value at each call to EditString.

Summary:

No flags
    Fix in overwrite mode

EdFlagInsert
    Fix in insert mode

EdFlagInsStat
    Allow Ins key to toggle mode

EdFlagInsStat + EdFlagForceIns
    Allow Ins key toggle, but start in overwite mode

EdFlagInsStat + EdFlagForceIns + EdFlagInsert
    Allow Ins key toggle, but start in insert mode

FIELD CLEARING AND EDIT KEYS

The flag EdFlagFirstClr determines the action taken by EditString when the first key is pressed to start the edit. By default, the first key pressed is added to the string in the usual way, either by overwriting the first character or by being inserted before the first character.

If EdFlagFirstClr is set, the first character typed causes the field to be cleared. Any cursor movement made before the first allowable character is typed causes this blanking to be disabled. This arrangement lets the user edit the existing string by moving the cursor and making the changes, but also gives a "clean slate" when entering an entirely new string.

In the section Basic editing an example was given which showed the cursor left and right keys used as exit keys. This would appear to create a conflict: Both keys are exit keys and both keys perform cursor movement within the editing field. EditString usually handles such conflicts in a very simple way. If the key is the first key pressed after EditString was called (or the first key pressed after a restore action has been performed), then the key is treated as an exit key and terminates the edit accordingly. If the key is pressed after an edit has started, it is treated as a cursor-movement key.

Setting the flag EdFlagEdKeyExit causes edit/exit key conflicts to be handled in quite a different way - the editing function of any edit key which is defined as an exit key is simply disabled. The cursor left and right keys in the above example, therefore, would terminate the edit whenever they were pressed; it would not be possible to use the cursor keys to move around the field itself. It is not advisable to define the Backspace key as an exit key if this flag is set; doing so would leave the user no easy way to correct typing errors.

The final item regarding field clearing concerns the EditKey field in the form record. When EdFlagFirstClr is set it is possible to edit the existing contents of the field by moving the cursor with the left or right arrow keys, or by pressing Home or End. This removes the "not yet edited" signal and prevents the first allowable character typed from clearing the field. If the cursor movement keys have all been defined as exit keys this is not possible, because pressing any one of them to start editing will cause EditString to terminate. Pressing a regular character, however, will cause the existing contents of the edit field to be deleted.

You may define a key that will allow the user to edit the existing string in such circumstances by assigning a value to form.EditKey. Pressing the edit key tells EditString that the first allowable character typed should not cause the field to be cleared. You may use EditKey even if the regular cursor movement keys are not defined as exit keys, and you can avoid defining an edit key by assigning a code of NUL.

CURSOR CONTROL

By default, EditString does not change the display's cursor in any way; whatever size cursor is selected when EditString is called is used throughout the edit.

If you have enabled insert/overwrite switching of the cursor by setting CursorInsert to true, then the Ins key will

still change the size of the cursor accordingly. Whether EditString recognizes this switching depends upon the state of the appropriate flags in the format record. If your edit format has fixed insert or overwrite mode and you do not want the Ins key to change the cursor for the duration of the edit, you should disable the cursor switching before calling EditString.

The flag EdFlagHideCrsr in form.Flags allows EditString to manipulate the cursor to a limited degree. Setting this flag causes the edit routine to hide the cursor when first called. As soon as editing starts, by the user moving the cursor or entering text, the cursor is re-enabled. This arrangement may be used to keep the cursor hidden while the user moves up or down to the required field on the screen; it is turned on only when needed to edit the selected field. When EditString terminates, the cursor's hide status is reset to that upon entry.

STRING FORMATTING

Normally, EditString returns with the variable specified as parameter s set to the exact string that the user typed, with any leading or trailing spaces included. The string may vary in length from zero to the value specified in parameter width.

The flags EdFlagTrimL and EdFlagTrimR allow leading and trailing spaces, respectively, to be removed from the edited string. You may set either or both of these flags and when EditString terminates it will re-write the adjusted string in the editing field. These flags are useful for trimming strings entered by the user where extra spaces may cause alignment problems, for example.

EdFlagPadL and EdFlagPadR perform the converse function and ensure that the edited string is of the length specified in parameter width by adding spaces to the beginning (PadL) or end (PadR) of the string. You may set both flags but EdFlagPadL takes precedence, so spaces will be added to the start of the string only.

The trim and pad options may be combined to justify the edited string. Combining EdFlagTrimL and EdFlagPadR, for example, ensures that the final string is left-justified by trimming leading spaces and then adding trailing spaces to the required field width. A detailed description of the trim, pad, and justify functions will be found in the STRINGS unit's documentation.

Finally, the flag EdFlagUpper may be set to force all alphabetic characters to upper case. EditString usually

treats upper- and lower-case letters as different; if only one type is defined in form.AllowChars then the other type will be rejected. Setting EdFlagUpper causes all lower-case letters to be converted to capitals before being processed any further.

MISCELLANEOUS CONFIGURATION CONTROL

The final flag defined by the current version of ENHCON is EdFlagFlushKB. Setting this flag causes the keyboard buffer to be flushed at each call to EditString, thus preventing key-strokes left in the buffer from before the call from being accepted.

The fields NumFormat and SignalError in form are not used by EditString. Their use is described in the number-editing routines.

```
FUNCTION EditReal (form:      EditFormatRec;
                   VAR r:     REAL;
                   min, max:  REAL): CHAR;
```

This function provides a way to conveniently display and edit real numbers on the console's display. The real variable to be edited should be supplied as parameter r and the cursor should be moved to the first position of the required field on the screen. The parameters min and max should be set to numbers representing the minimum and maximum allowable value for r.

EditReal takes a format record as a parameter, just as EditString does, and each field of this record controls one aspect of the way in which EditReal operates. The fields Attribute, StartChar, EndChar, MarkerAttr, ExitKeys, EditKey, RestoreKey, and AbortKey work in exactly the same way (see EditString for details). The characters defined in AllowChars are ignored; EditReal will accept the characters necessary to form a valid number. Almost all of the flags also perform the same functions as in EditString, but the PadL, PadR, TrimL, and TrimR flags are ignored.

EditReal does, however, use the NumFormat field of form. NumFormat should be set to a string suitable for passing to the Format function in the STRINGS unit. This string is used to determine the way in which the real number is displayed and follows the rules laid down by the Format function. You should be thoroughly familiar with the way in which Format works before attempting to use EditReal.

The total width of the editing field is determined by the width specified in NumFormat (i.e. the length of a string returned by STRINGS.Format). Format is called to convert the supplied number into a displayable form and then the user may enter a new number or edit the existing one. For example, if the string in form.NumFormat was '+10:3' and the initial value of r was 470.58, then EditReal would start by displaying a field 10 characters wide containing " +470.580". Note that you should not attempt to use vulgar-fraction notation in a format string passed to EditReal, as the function will not recognize this format.

Once the user has entered a new value, EditReal adjusts it to the format specified in NumFormat and terminates. The existing value may be accepted, of course, by just pressing one of the exit keys, just as with EditString. (The numerical edit routines actually convert the number to a string using Format and then call EditString to perform the editing.)

Because of the way in which EditReal is written, a handy programming trick using the plus and minus keys is possible. It is sometimes useful to be able to enter a number and terminate its entry by pressing plus or minus. If the plus and minus keys are defined in form.ExitKeys, EditReal allows this - the number will be positive or negative as appropriate and EditReal will return a character of "+" or "-".

EXPONENTIAL NOTATION

There is one case in which the number initially displayed by EditReal does not conform to the way in which STRINGS.Format handles real numbers. If the decimals specifier in NumFormat is zero (e.g. '12:0' or '12'), then r is shown in exponential form. The above example of 470.580 would be displayed as " 4.70580E+02" if the field width was 12 characters. When exponential format is used, any other options listed in NumFormat are also ignored; it is most unlikely that zero-fill, floating dollar signs, and so on would be required with such formats anyway.

It is not a good idea to try to use the plus and minus keys as exit keys when working in exponential format; real numbers displayed in this form may need two signs, one for the mantissa and one for the exponent.

If EditReal cannot successfully convert the number entered from a string to a real value or if the converted value is outside the range specified by parameters min and max, a call to form.SignalError is made. SignalError is a field which is defined as a procedural variable - a call to SignalError results in a call to whatever procedure name was stored in SignalError.

You can use ENHCON's default routine by assigning StdSignalError in your format record, e.g.

```
MyFormat.SignalError := StdSignalError;
```

Under these conditions a conversion or range error causes EditReal to generate a beep and re-display the original number so that the user may try again.

In some programs it may be desirable for the computer to display a short error message on a status line or open a window on the screen to provide assistance. This can be done by assigning your own error-handler to SignalError. There are several conditions which must be strictly adhered to if your own error routine is not to upset the operation of EditReal. First, your procedure must take a single value of type byte as a paramater. Your procedure heading should look something like this:

```
PROCEDURE MyRealErrorSignal (width: BYTE);
```

Second, the procedure must be compiled in the far model by placing it after a $F+ directive or by having it appear in the interface section of a unit. This is a requirement of Turbo Pascal's procedural variables.

Third, your procedure must return with certain aspects of the display exactly the same as when your routine was called. Any of the following items which are changed by your routine should be restored to their original values before returning: the position of the cursor; the size and hide-status of the cursor; the value of TextAttr.

When your routine is called the cursor will be left positioned at the start of the editing field and paramater width will hold the width of the field. This enables you to put a short message in the editing field should you so desire (note that when your routine returns, however, the edit field will revert back to its original contents).

Note that a format record which is to be used by EditReal must have some valid procedure assigned to its SignalError field. If a range or conversion error is detected and this field is uninitialized, a call will be made to some random address in memory, causing your program to fail.

```
FUNCTION EditInt (form:        EditFormatRec;
                  VAR i:       LongInt;
                  min, max:    LongInt): CHAR;
```

EditInt works in a very similar way to EditReal, but allows integers to be edited. All of the facilities available in EditString and EditReal are also available in EditInt, including the use of form.SignalError to install an error-handling routine for conversion and range errors.

The NumFormat field of form is handled in a slightly different way by EditInt, however. If the format string specifies no decimal places, as you would expect for an integer value, then it is passed to STRINGS.Format in the usual way. Any of the valid options of Format may be used. For example, the format string '6+' would cause the value 45 to be displayed as " 45+". Note that the total field width is that specified in the format string, just as with EditReal.

EditInt also provides for an assumed decimal point, thus allowing an integer variable to hold fixed-decimal-point values. The format string in NumFormat specifies the number of assumed decimal places (e.g. '12:3' specifies three assumed decimal places) and EditInt displays the value with a decimal point inserted at the appropriate place. For example, a value of 1,258 is displayed as "12.58" if two decimal places are specified. Similarly, numbers entered by the user are converted in a converse manner, so that "3.47" would be returned as an integer value of 347 (again assuming that two decimal places were specified).

A typical example of using assumed decimal places is that of an accounts program. It is common practice to use a LongInt-type variable to store amounts of money - this enables faster integer arithmetic to be employed and eliminates rounding errors that may appear if real values were used. An amount of $12.75 would be stored as 1,275, for example (i.e. the value gives the amount directly in cents). To display a monetary value and allow the user to edit it, EditInt may be called with two decimal places specified. The value is thus presented in the familiar manner and the user may enter a new value in the same format.

The following examples should help clarify the use of assumed decimal places:

Value	Format string	Displayed as
1,234	10:2	12.34
475	10:1	47.5
2,700	10:4	0.2700

User's entry	Format string	Returned value
37.05	10:2	3,705
50	10:5	5,000,000
4.6	10:3	4,600

Any of the display options that STRINGS.Format accepts may also be used when specifying assumed decimal places. For instance, the format "$8:2+" would cause a value of 500 to be displayed as " $5.00+".

```
FUNCTION EditDate (form:EditFormatRec; VAR Dt:DateRec): CHAR;
FUNCTION EditTime (form:EditFormatRec; VAR Tm:TimeRec): CHAR;
```

These two functions are closely associated with the TIME unit and allow console editing of a date or time. You should be thoroughly familiar with the TIME unit before attempting to use either of these routines.

The values passed in parameter form control the edit in the same way as they do for EditString, with the exception that characters specified in AllowChars are ignored (both routines automatically accept the necessary characters to form a date or time). The cursor should be positioned at the start of the required editing field before calling EditDate or EditTime and the date or time to be edited should be suplied as parameter Dt or Tm, respectively.

EditDate converts parameter Dt into a displayable date by calling the DateStr function in the TIME unit. The exact format of the string is, therefore, determined by the current configuration of the unit. DateStr returns a string with a maximum length of nine characters, so EditDate automatically uses an editing field of this width. The new, or edited, string entered by the user is converted back to a date record by calling the TIME unit's DateParse function. Again, the way in which DateParse processes the string is dependent upon the current settings in the

TIME unit itself.

In a similar way, EditTime converts parameter Tm into a displayable string by calling the TimeStr function in the TIME unit and processes the user's input by calling TimeParse. As with EditDate, the exact format used is dependent upon the current settings in the TIME unit, but EditTime always uses an editing field 13 characters wide (the maximum length of a variable of type TimeString).

The SignalError field of parameter form is used the same way as in the edit number routines. You should assign either StdSignalError or your own error-handler to this field; the specified procedure is called whenever the entered date or time cannot be successfully converted. All rules regarding the use of this facility are the same as for the number routines (see Range and conversion errors in EditReal). The NumFormat field of form is ignored by EditDate and EditTime.

USING DISPLAY WINDOWS

Up to 255 user-defined windows are available; window zero is pre-defined and refers to the whole screen.

The data type WindowDefinition is used to hold all the basic information required to define a window. WindowDefinition is dependent upon two other data types: WindowBorder and WindowJustify.

```
WindowBorder      =  ARRAY[1..8] OF CHAR;
WindowJustify     =  (WJustLeft, WJustCenter, WJustRight);

WindowDefinition  =  RECORD
                       X1, Y1, X2, Y2:  BYTE;
                       DefaultAttr:     BYTE;
                       DefaultCrsrHide: BOOLEAN;
                       DefaultCrsrSize: BYTE;
                       Border:          WindowBorder;
                       BorderAttr:      BYTE;
                       HdrText,
                       FtrText:         ConsoleStr;
                       HdrAttr,
                       FtrAttr:         BYTE;
                       HdrPos,
                       FtrPos:          WindowJustify;
                       Flags:           BYTE:
                     END;
```

Fields X1 and Y1 specify the top-left corner of the required window (as column and row respectively), numbered as for the GotoXY procedure. The standard 80-by-25 display, therefore, has rows numbered 1 through 25 and columns numbered 1 through 80. Fields X2 and Y2 specify the bottom-right corner of the required window.

DefaultAttr determines the text attribute that will be set when the window is opened. Assigning a value of 01 hex., for example, will cause text written into the window to be underlined on a monochrome display or blue on a color display.

The two fields DefaultCrsrHide and DefaultCrsrSize are used to control the cursor. The first determines whether the particular window has a visible cursor or not. Setting DefaultCrsrHide to true causes the cursor to be hidden when the window is opened; setting it to false ensures that a cursor is visible, even if it was hidden before the window was opened.

DefaultCrsrSize is used to control the size of the cursor, and may be set to any valid values for the display adapter in use. The value used will be passed to the SetCursor routine when the window is opened; whether the cursor will actually be visible or not depends upon the setting of DefaultCrsrHide. There are three key values which may also be assigned to DefaultCrsrSize and these are defined as constants:

```
WCrsrDefault    =  $FF00;
WCrsrLine       =  $FE00;
WCrsrBlock      =  $FD00;
```

WCrsrLine and WCrsrBlock are provided to allow you to define a line or block cursor for the window without having to calculate the actual values (which will vary from one type of display to another). Assigning one of these values to DefaultCrsrSize causes the appropriate cursor size to be set when the window is opened. The constant WCrsrDefault may be used if the size of the cursor should remain unchanged; opening the window will still set the cursor to hidden or visible, but its size will remain unaltered.

Two fields control the way in which a border around the window may be shown on the display. The field Border allows eight characters to be defined:

Index	Position
1	Top left corner
2	Top line
3	Top right corner
4	Right side
5	Bottom right corner
6	Bottom line
7	Bottom left corner
8	Left side

Typed-constants are defined to enable four commonly used borders to be assigned to Border easily. These are:

Border1	A single-line border
Border2	A double-line border
BorderV1H2	Single vertical, double horizontal lines
BorderH1V2	Single horizontal, double vertical lines

BorderAttr defines the text attribute to be used for the display of each border character and determines how the border will appear on the screen. If a window is defined as not having a border (see description of flags, below) then the Border and BorderAttr fields are ignored.

Header text can also be placed in the top border of a window. This text may be used to provide a title for a menu or other window. The string to be displayed should be stored in HdrText and the required text attribute for the title should be assigned to HdrAttr. The HdrPos field controls the positioning of the text and may be set to one of three values: WJustLeft, WJustCenter, or WJustRight. These cause the title to be left-justified, centered, or right-justified, respectively. If no header is required, HdrText should be set to a null string and the HdrAttr and HdrPos fields are then ignored. The maximum length of the header text is less than the width of the window's border, because the two corners are not used; if HdrText is too long it is truncated.

In a similar way, the fields FtrText, FtrAttr, and FtrPos allow text to overwrite the bottom border of a window. As with the header definition, setting FtrPos to a null string prevents any footer from being displayed.

Note that the header and footer options are used only if the window is defined as having a border. If the window does not have a border, all six header and footer

control fields are ignored.

The final field of WindowDefinition contains a collection of flags. Six are used in the current version of ENHCON and they are defined as constants. A more detailed description of each flag is provided in later sections, but the following summary provides a quick overview.

WFlagClrOpen	Clear window on opening
WFlagClrClose	Clear window on closing
WFlagClrHide	Clear window on hiding
WFlagRestore	Restore original when closed/hidden
WFlagShowBrdr	Window has border
WFlagWriteBrdr	Border can be overwritten

The values for the required flags should be added together when assigning a value to Flags.

WINDOW ZERO

Window numbers 1 through 255 are available for use by your application program. Window zero is pre-defined and represents the entire screen, which will usually be 80 columns by 25 rows.

Window zero is automatically made active at the start of any program using the ENHCON unit and the cursor and text attribute will be those in effect when the program was loaded. It is quite permissible for window zero to be selected in order to use the full screen as a window, but no attempt should be made to close, hide, or move window zero; doing so will generate an error.

PROCEDURE DefineWindow (WindowID: BYTE; d: WindowDefinition);

This procedure should be called to define each window that your application program requires. A window definition should be passed as parameter d and the window number as WindowID. The window number specified may not be a window that has already been defined and may not be zero.

DefineWindow performs several error checks on the supplied definition before allowing a new window. Possible error codes are 01 (Invalid co-ordinates), 02 (Invalid border co-ordinates), 08 (Illegal window zero operation), 09 (Window already defined), and 10 (Out of memory). The two errors relating to invalid co-ordinates require some clarification.

The co-ordinates (X1, Y1, X2, and Y2) must be within range for the current display; X1 and Y1 must be less than or equal to X2 and Y2, respectively. The minimum window size possible, therefore, is one row by one column and failure to satisfy these conditions results in error 01. If the window has a border, the co-ordinates specify the top-left and bottom-right corners of the border and there must be at least one row and one column inside the border. Failure to satisfy this requirement results in error code 02 being returned.

Note that DefineWindow simply assigns a number to a specific window definition; it does not cause the window to be opened on the display.

PROCEDURE OpenWindow (WindowID: BYTE);

Calling this procedure causes a previously defined window to be opened on the screen. The window number required should be passed as parameter WindowID. OpenWindow calls Turbo Pascal's Window procedure to set the co-ordinates of a new display window and all further output is then sent to that window instead of the whole screen. The top row and left column of a window are numbered as one.

Opening a new window causes the text attribute to be set to that specified by DefaultAttr in the definition, and the cursor size and hide status to be set to the values found in DefaultCrsrSize and DefaultCrsrHide (see Defining a window). This process ensures that the default cursor and text colors are used whenever a window is opened.

If the flag WFlagClrOpen is set, the newly opened window is cleared to the new text attribute. If this flag is clear the current screen contents are left in place; this gives rise to a "see through" effect. Most programs will use the former option.

WFlagShowBrdr controls whether a border is drawn around the new window. Setting this flag to true causes the border, header, and footer to be written to the screen, as described earlier. If a border is enabled the flag WFlagWriteBrdr is also used. A value of false causes the window's co-ordinates to be set one row and column inside the border, so that the area inside the border scrolls, leaving the border intact. A true value for this flag sets the new co-ordinates on top of the border, allowing the program to write characters over part of the border. The latter option is useful for such things as menu option arrows.

The cursor, whether visible or hidden, is positioned at the top-left corner of the new window. With a bordered window this will be on the top-left border character if WFlagWriteBrdr is true or one column and row inside the border if WFlagWriteBrdr is false. Finally, OpenWindow makes the new window active so that further output to the display is sent to the new window.

Possible error codes returned by OpenWindow are 04 (Window is already open), 08 (Illegal window zero operation), 0A (Window not defined), and 10 (Out of memory).

PROCEDURE SelectWindow (WindowID: BYTE);

This procedure is used to move from one open window to another. The required window number should be passed as parameter WindowID. SelectWindow does not change the contents of any window; it simply allows movement from one to another.

The ENHCON unit maintains a record of the cursor and text attribute for each open window. When SelectWindow is called, the details of the current window are saved, the requested window is made active, and its cursor and text attribute are restored. The cursor and text-attribute save feature ensures that when a window is re-selected everything is exactly as it was when that particular window was last used.

It is important to realize that DefaultAttr, DefaultCrsrSize, and DefaultCrsrHide are used only by OpenWindow. SelectWindow automatically restores whatever the last values for the window were; it does not restore the defaults specified in the window's definition. If a window is closed and then later re-opened, the defaults in its definition will be used once again.

Possible error codes returned by SelectWindow are 05 (Window is not open), 07 (Window is hidden), and 0A (Window not defined).

PROCEDURE CloseWindow (WindowID: BYTE);

This procedure performs the converse function to OpenWindow and removes a window from the screen. The window to close should be passed as parameter WindowID.

Two flags in a window's definition affect the way in which the window is closed: WFlagClrClose and WFlagRestore.

If both flags are clear the window is considered closed but the area of the screen it occupied is left unchanged. Setting WFlagClrClose causes the area previously occupied by the window to be cleared; the value in DefaultAttr is used for this operation. Finally, if WFlagRestore is set the original screen contents are restored. This allows a window to be "popped up" temporarily over an existing display.

A list of the order in which windows were opened is maintaned by the unit. If the window closed by CloseWindow was not active, the currently active window is not changed. If the closed window was also the current window, however, an attempt is made to select whichever window was active at the time the closed window was opened. This facility simplifies nested "pop up" menus, and similar constructions, by avoiding the need to explicitly re-select windows.

Note that if it is not possible to re-select the previous window (because it has been closed, hidden, or purged) an error will be generated. If you do not require this "trace back" facility and want to avoid errors, you should ensure that when a window is closed it is not currently active.

It is quite permissible to close a window which is hidden. Doing so will have no effect on the display and simply throws away the stored text of the hidden window (see HideWindow).

Possible error codes returned by CloseWindow are 05 (Window not open), 08 (Illegal window zero operation), 0A (Window not defined), and 0B (Cannot return to previous window).

```
PROCEDURE HideWindow (WindowID: BYTE);
PROCEDURE ShowWindow (WindowID: BYTE);
```

A window can be removed from the display by calling the CloseWindow procedure, but its contents will be lost. HideWindow allows a window to be removed from the screen but saves its contents so that they may be restored later by ShowWindow.

Two flags in a window's definition affect the operation of HideWindow and ShowWindow: WFlagClrHide and WFlagRestore. The former works in a similar way to WFlagClrClose with CloseWindow, and the latter is identical. If both flags are clear, HideWindow stores the window's contents but leaves the screen unchanged. If WFlagClrHide is set, the area previously occupied by the window is cleared.

If WFlagRestore is set, the original screen contents are restored. If the window to be hidden is currently active, then window zero is automatically selected.

When a window is recalled with ShowWindow its entire area is re-drawn - border, header, footer, and text. If the screen restore feature is in use (i.e. WFlagRestore is set), then ShowWindow updates the saved area of the original screen so that CloseWindow will restore the correct text.

Possible error codes returned by HideWindow and ShowWindow are 05 (Window not open), 08 (Illegal window zero operation), 0A (Undefined window), and 10 (Out of memory). HideWindow can also return code 06 (Window already hidden) and ShowWindow can return 07 (Window not hidden).

```
PROCEDURE RelocateWindow (WindowID: BYTE; X, Y: BYTE);
PROCEDURE MoveWindow (WindowID:  BYTE;
                      Direction: WindowMovement);
```

These two routines allow a window to be re-positioned on the screen. Relocate window takes the co-ordinates supplied as parameters X and Y and uses them as the new top-left corner of the window. The new bottom-right co-ordinates are calculated automatically from the window's width and height. MoveWindow takes a value of type WindowMovement which specifies whether the window should be shifted up or down by one row or left or right by one column. WindowMovement is declared in the interface as an enumerated type:

```
  WindowMovement = (WMoveLeft,WMoveRight,WMoveUp,WMoveDown);
```

Either routine may be called for any window which is defined. If the window is closed or hidden the changes take effect when it is next opened or re-displayed. If the window is open and on display, the move is made immediately.

Possible error codes are 01 (Invalid co-ordinates), 03 (Cannot move window in specified direction), 08 (Illegal window zero operation), 0A (Window is undefined), and 10 (Out of memory).

```
PROCEDURE WriteWindow (s: ConsoleStr);
```

Although Turbo Pascal's standard Write and WriteLn procedures can be used to write text into a window, they do cause one problem: When an attempt is made to fill the window down to the last row and column, the whole window scrolls.

WriteWindow is provided to avoid this problem. The string supplied as parameter s is written into the window at the current cursor position and using the current text attribute, but the cursor is not moved, thus preventing the window from scrolling. In addition, if the string is too long to fit between the current cursor position and the edge of the window, it is truncated.

The only error code that can be returned by WriteWindow is 10 (Out of memory).

```
FUNCTION CurrentWindow: BYTE;
```

This function simply returns the identification number of the currently active window. If CurrentWindow is called before any window has been opened, it will return zero, indicating that window zero (the whole screen) is active.

```
FUNCTION WindowStat (WindowID: BYTE): WindowStatus;
```

WindowStatus is an enumerated type declared in the interface section of the unit:

```
WindowStatus = (Undefined, Closed, Hidden, Open, Active);
```

A call to WindowStat returns the status of the specified window. Note that window zero can only return Open or Active, because it cannot be closed or hidden, or have its definition purged.

```
PROCEDURE PurgeWindow (WindowID: BYTE);
```

PurgeWindow is used to remove a window's definition from the ENHCON unit. This facility will not be required by most programs, but is provided for two reasons: First, if a window must be re-defined it must be purged before attempting to call DefineWindow again. Second, in programs which use a great many windows and must run on a small

system it may be necessary to reclaim as much working store as possible when a window is not in use. (Each window definition takes up almost 200 bytes of heap storage.)

The window number passed to PurgeWindow should represent a closed, defined window, and possible error codes are 04 (Window is open), 08 (Illegal window zero operation), and 0A (Undefined window).

```
PROCEDURE GetWindowDef (WindowID: BYTE;
                        VAR d:    WindowDefinition);
```

GetWindowDef is unlikely to be required very often, but is included for completeness. Calling this routine causes the definition of the specified window to be returned in record d.

The only possible error code for GetWindowDef is 0A (Undefined window).

WINDOWS ERROR HANDLING

The ENHCON unit supports a comprehensive error-handling system for dealing with errors in the windowing routines.

By default, any windows-related error will terminate the program with an appropriate message. For example:

```
ENHCON unit run-time error 0A
Undefined window in OpenWindow
```

The above message represents an attempt to open a window which has not been defined. The global variable EnhConHaltError can be set to a program return code; when a program is aborted due to an error this value is passed back to the operating system. (The program return code is accessible from a batch file using ERRORLEVEL.) The default value of EnhConHaltError is zero.

Program termination due to error can be avoided by setting the global variable WindowCheck to false. Under this configuration each windows routine sets a result code (very much like the IOResult code in Turbo Pascal). A program may call the function WindowResult after any other windows routine in order to check the result of the operation. The windows routine descriptions, above, indicate the possible error return codes from each

routine. A value of zero indicates that the routine was successful.

The interface section of ENHCON defines several constants which specify the possible error codes in the current version:

Constant	Value	Error
ConErrXY	01	Invalid co-ordinates
ConErrBorderXY	02	Invalid border co-ordinates
ConErrMove	03	Invalid direction
ConErrOpen	04	Window is open
ConErrClose	05	Window not open
ConErrHidden	06	Window is hidden
ConErrNotHidden	07	Window not hidden
ConErrZero	08	Illegal window zero operation
ConErrDefined	09	Window already defined
ConErrUndefined	0A	Undefined window
ConErrReturn	0B	Cannot return to previous window
ConErrHeap	10	Out of memory
ConErrHelpRead	11	Cannot access help file
ConErrHelpInit	12	Help system already initialized
ConErrNoHelpFile	13	Help file not found
ConErrHelpFormat	14	Invalid format in help file
ConErrHelpIndex	15	Invalid format for help index
ConErrHelpInvalid	16	Help record invalid
ConErrHelpStkFull	17	Help context stack full
ConErrHelpStkEmpty	18	Help context stack empty

Note that errors 11 through 18 hex. relate to the on-line help system, described in a later section of this manual.

```
FUNCTION WindowResult: BYTE;
```

This function is used to obtain the result code of the last windows routine that was called. WindowResult returns zero if the last operation was successful or one of the error codes listed above if it failed for some reason.

All window-related routines cause the result code to be set. It is not mandatory to call WindowResult after every windowing operation, but errors will go undetected if a call is not made. As with Turbo Pascal's own IOResult, calling WindowResult also resets it to zero. A call to this function when internal error checking is enabled (i.e. WindowCheck is true) serves no useful purpose - errors will cause a program to abort before it can call WindowResult.

```
FUNCTION ConErrorMsg (ErrNum: BYTE): ConsoleStr;
```

This function is provided so that application programs can display a suitable error message when internal error trapping is disabled. ConErrorMsg should be given the result code returned by WindowResult or the help error-handler as its parameter ErrNum. The returned string will be one of the error messages listed above.

A parameter value of zero causes the string "Successful" to be returned and a code which does not specify one of the errors listed returns "Undefined error".

THE ON-LINE HELP SYSTEM

The help system relies upon the windows routines to provide "pop up" displays and will most likely be used in programs which make extensive use of windows. One window identification number must be set aside for the exclusive use of the help system.

CREATING THE HELP FILE

The help system requires the help file to be stored on disk in a special format and a conversion utility is provided so that the help text can be created using a standard ASCII editor.

The first line of the text file must contain just a single number that specifies the number of help sections required. This number must be in the range 1 through 255. Following lines may contain any comments you wish to add to the file for future reference; these comments do not form part of the help text and will not be stored in the final help file.

The text for each help section follows the first line or any comment lines. Each section must start with a header which includes the section number (represented by "n") and name:

```
#n:Section name
```

Note that the "#" symbol must be in the first column and be immediately followed by the section number and a colon. The text following the colon is used as the name for the section and will appear in the help index. Section names are limited to a maximum length of 30 characters and will be truncated if longer.

The text for the section should immediately follow the header and be in the form that you wish it to appear in the help window. There are three restrictions that apply to this text. First, you should ensure that no help text has a "#" in the first column, as this will cause the conversion program to start a new section. Second, no help section may have more than 1,000 lines of text. Third, any line which is too long to fit across the help window will be truncated when it is displayed; you should ensure that your text fits within the margins of the intended window.

The final section's text should be followed by a "#" on a new line. You may place any comments after this marker; all text following it will be ignored when the file is converted. For example:

```
3
This text is a comment and will be ignored
#1:First section
This is the help text for section 1.
This section will appear in the index as "First section."
#2:Second section
This is the help text for section 2.
The index will show this section as "Second section."
#3:Third & final
This is the help test for section 3.
The index will read "Third & final."
#
This text is also a comment
```

Once you have created the help text required you should convert it to a format suitable for the ENHCON unit by running the HELPCONV utility. The command syntax for this program is

```
HELPCONV input-file output-file
```

where input-file specifies the path and name of the text file to be processed. The output-file parameter is optional and may specify a path, a file name, or both. If no path is specified the help file will be created in the same directory as the input file; the file name defaults to the input file name with the extension changed to "HLP" unless specified to the contrary.

If the text file is converted successfully, HELPCONV will display a count of the number of lines processed and the number of help sections. If the text file is incorrectly formatted in any way, a format error will be displayed, showing the line number at which processing stopped.

A format error can be due to any of the following:

1. The first line does not specify the number of help sections or there are fewer help sections in the file than specified by this number.

2. A section of text does not have a properly formatted header. The "#" symbol must be in the first column and be followed by the section number and a colon, with no intervening blanks.

3. A section is missing or listed out of sequence. Sections must be listed in ascending numerical order.

4. A section of help text is longer than 1,000 lines.

5. A line in the help text has a "#" in the first column. If the help text requires this symbol at the start of a line, precede it with a space.

6. There is no terminating "#" after the last help section.

Once the help conversion utility has been run successfully, the output file can be used by your program.

```
PROCEDURE HelpInitialize (h: HelpConfiguration);
```

The following type declaration appears in the interface section of the unit:

```
HelpConfiguration = RECORD
                      WindowID:           BYTE;
                      HelpFileName:       ConsoleStr;
                      X1, Y1, X2, Y2:     BYTE;
                      NormalAttr,
                      IndexAttr,
                      SelectAttr:         BYTE;
                      Border:             WindowBorder;
                      BorderAttr:         BYTE;
                      HdrText, FtrText:   ConsoleStr;
                      HdrPos,  FtrPos:    WindowJustify;
                      HdrAttr, FtrAttr:   BYTE;
                      GeneralKey,
                      ContextKey,
                      LastHelpKey,
                      MoveWindowKey:      CHAR;
                      Flags:              BYTE;
                    END;
```

Several of the fields are the same as those in a record of type WindowDefinition. Each field is described in detail below. A variable of type HelpConfiguration should be used to set the fields to the required values, then HelpInitialize should be called to set up the help unit.

The number passed in WindowID is used as the window number for the "pop up" help window. This number must be in the range 1 through 255, and your application program should not attempt to access this window directly (with DefineWindow, OpenWindow, and so on).

HelpFileName specifies the path and name of the file that holds the help text. Note that the string supplied is with reference to the current directory at the time of the call to HelpInitialize; if the help file is not in the current directory, HelpFileName should also explicitly specify the correct drive and path.

X1, Y1, X2, and Y2 operate in the same way as for a window definition and specify the top-left and bottom-right corners of the required help window. A help window always has a border, so the available area for help text will be two columns and two rows less than the total area of the window.

NormalAttr specifies the text attribute that will be used to display regular help text. IndexAttr and SelectAttr control the way in which the index will appear: The first specifies the text attribute for unselected index words and the second specifies the attribute to be used to highlight a particular entry.

Border and BorderAttr are also identical to a regular window definition and specify the characters and attribute to be used when displaying the help window's border.

The unit always defines the help window as having a border, as this is the usual requirement for a "pop up" window. It is possible to prevent a border from being displayed, however, should this be required (such as when using a help window that occupies the full screen). Setting all eight border characters to a space (32 decimal, 20 hex.) or setting BorderAttr to zero (MonoNone) will accomplish this. The area that would be occupied by the border cannot be used, so a standard 80-by-25 display would have an area of 78 columns by 23 rows available for the help text.

A help window can have a header and footer, and these fields are identical in operation to the fields of the same name in a regular window definition. You can disable a header or footer by setting HdrText or FtrText (or both)

to a null string. The header and footer of a help window are also affected by some other options.

Four fields (GeneralKey, ContextKey, LastHelpKey, and MoveWindowKey) define which keys will be used to control the help system. They are described in the section titled Using help.

Three flags are used by the help system, and these are defined as constants:

```
HFlagTitle      =  $04;
HFlagPageInd    =  $02;
HFlagPageText   =  $01;
```

These values should be added together to set the required flags.

Setting HFlagTitle causes the name of the currently displayed help section to be appended to the header text. If, for example, the current section is named "File editing" and the string in HdrText is 'Help: ' then the header would appear as "Help: File editing". When the index is on display the string 'Index' is used in place of the section name. If HdrText is a null string then the header will consist of just the section name.

Setting HFlagPageInd causes up and down page indicators to be appended to the footer text. These indicators take the form of two arrows (character codes 24 and 25 decimal, 18 and 19 hex.) which show whether there is more text above or below the current point. If the flag HFlagPageText is also set, then the arrows are replaced with "PgUp," "PgDn," or "PgUp/PgDn" as appropriate. If FtrText is a null string, then the footer will consist of just the page up/down indicators.

Errors in the help system are handled differently to those caused by a windows routine (see Help error handling, below). Possible error codes from HelpInitialize are 01 (Invalid co-ordinates), 02 (Invalid border co-ordinates), 08 (Illegal attempt to use window zero), 09 (Window already defined), 10 (Out of memory), 11 (Cannot access help file), 12 (Help system already initialized), 13 (Help file not found), 14 (Format error in help file), 15 (Error in index entry).

INDEX LAYOUT

It is not necessary to specify how the index is to be laid out; the unit does this automatically when HelpInitialize is called. Index layout is determined by the size of the help window and the length of the longest section name, and ENHCON will fit as many columns of index entries across the window as is possible. The help window must be at least wide enough to hold the longest section name.

USING HELP

After a call to HelpInitialize has been completed successfully, the help system is ready for use. ENHCON monitors keyboard input for the help key, or keys, by linking into the standard ReadKey function. This means that all console input should be by way of ReadKey if on-line help is to be available throughout the program. On-line help will not be available during any input with Read or ReadLn.

All the editing routines in the ENHCON unit use ReadKey and will, therefore, allow the help system to be activated during an edit.

GENERAL HELP

The key specified in the GeneralKey field of the help configuration will be trapped by ReadKey and cause the help index to be displayed. The first index entry is highlighted and the cursor keys can then be used to select the required topic from the index. If the window is not large enough to hold the section name for every topic, the index is automatically split into pages. The following keys also have an effect while the index is displayed:

Tab	Move forward to next index entry
Shift-Tab	Move back to previous index entry
PgDn	Move to next index page
PgUp	Move to previous index page
Enter	Select topic
Esc	Leave help system

When a specific topic has been selected with the Enter key, the index is cleared and the requested text is displayed. Once again, the text is automatically split into pages if it is too long for the help window and the PgUp, PgDn, Home, and End keys allow movement around the section's text. Pressing Escape causes an immediate exit

from the help system; pressing the general-help key results in a return to the index so that another topic may be selected.

When the help system is left by pressing Escape, the help window is closed and the original screen restored. ReadKey waits for another key to be pressed and returns this key to the calling program (unless help is requested again, of course). In this way the use of the help system is completely transparent to the application program.

CONTEXT-SENSITIVE HELP

Many large programs now use a context-sensitive help system, where requesting help will bring up the help section appropriate to the current area, or context, of the program. A code assigned to the ContextKey field of the help configuration record specifies the key that will activate context-sensitive help. If context-sensitive help is not required, the value NUL should be assigned to disable it.

There is a global variable called HelpContext (of type byte) which determines the current context. Whenever context-sensitive help is requested, the value in HelpContext is used to jump straight to a particular help section. If HelpContext is set to 5, pressing the context-sensitive help key will cause the help text for the fifth section to be immediately displayed, bypassing the index.

To implement this scheme effectively requires that a value be assigned to HelpContext whenever the main program enters an area which must have a different help section. The value assigned will be used as the help section for context-sensitive help until such time as a new value is assigned. Setting HelpContext to zero causes the context-sensitive help key to display the index, just as if general help had been requested. The index is also displayed if HelpContext does not hold a valid section number at the time help is requested (e.g. if HelpContext is 20 and the help text only has 10 sections).

Depending upon the program, it may not be desirable to have two separate help keys - one for general help and one for context-sensitive help. If GeneralKey and ContextKey are set to the same value, pressing the help key will display the context-sensitive help as usual. If the key is pressed a second time (i.e. while the help text is visible), the help window will switch to the index.

It is also possible to disable general help by assigning NUL to GeneralKey. Under these conditions it will not be possible to access the index unless context-sensitive help is

called for when HelpContext is set to zero. This option could be used if it is unlikely (or undesirable) that the user will want to "browse" through the help text.

```
PROCEDURE PushHelpContext (NewContext: BYTE);
PROCEDURE PopHelpContext;
```

In many applications, a subroutine that requires on-line help may be called from several different places in the main program. This requires the assignment of a new value to HelpContext upon entry to the subroutine, but the original value must be restored when control passes back to the main program.

The PushHelpContext and PopHelpContext procedures provide a simple way to achieve this by way of a help context stack. A call to PushHelpContext causes the current value of HelpContext to be saved on the stack; the value passed as parameter NewContext is then assigned to HelpContext. Calling PopHelpContext causes the previously stored value to be retrieved from the stack.

In this way, a commonly used subroutine need only call PushHelpContext with its appropriate context number upon entry and make a call to PopHelpContext before returning.

The stack has a maximum storage capacity of 127 numbers, which should be adequate for even the most complex application. PushHelpContext can return an error code of 17 (ConErrHelpStkFull) if the stack is full, and PopHelpContext can return an error of 18 (ConErrHelpStkEmpty) if there are no numbers currently stored on the stack.

Note that calls to either procedure are simply ignored if the help system is not initialized. This prevents calls from halting the application program if it is running without on-line help.

LAST-HELP FACILITY

One other way to activate the help system is provided: If LastHelpKey is assigned a key value, pressing the specified key will cause the last help text read to be re-displayed. It does not matter whether the last help access was from the index or by way of context-sensitive help; whichever section was on display when the help window was removed will be recalled. The user may then move around the text, press Escape to leave the help system, or press the general-help key to view the help index.

Setting LastHelpKey and GeneralKey to the same value will cause the specified key to activate the last-help facility. A second press of the key will then cause a switch to the index. If the last-help facility is not required, LastHelpKey should be assigned a value of NUL.

MOVING THE HELP WINDOW

The initial position of the help window is determined by the co-ordinates passed to HelpInitialize in the configuration record. In some programs it is desirable to allow the window to be moved by the user - so that it does not obscure other text, for example.

If a key code is assigned to the MoveWindowKey field of the configuration record, the specified key can be used to toggle the cursor key mode. Whenever the help system is activated (by pressing the general-help, context-help, or last-help key), the cursor keys allow selection of an index entry. After presssing the move-window key the cursor keys move the help window around the screen, whether the current help window is displaying the index or the text for a specific topic. Pressing the move-window key a second time fixes the window in its new position and returns the cursor keys to normal. If the user leaves the help system, the new position for the window is retained so that further calls to the help system will use the user's choice of window position.

There is one "special" value that may be assigned to MoveWindowKey. Assigning the constant HMoveScroll (character code 255 decimal, FF hex.) causes the scroll lock status to be used to determine the cursor-key mode. When scroll lock is on, the cursor keys move the help window; when scroll lock is off they allow movement around the index as usual.

Finally, if MoveWindowKey is assigned NUL it will not be possible for the help window to be moved from its initial position.

PROCEDURE HelpReset;

Most programs will initialize the help system once and then leave it in place. A call to HelpReset will allow the help system to be removed, should this be necessary. Possible reasons would be to reclaim memory space or to switch to a different help file (an advanced user's text in place of novice text, for example). After resetting the help system, no help will be available until

HelpInitialize is called to install a new help system.

One possible use of this technique is to separate help into distinct modules. An accounts program, for example, could have three main modules: Accounts receivable, accounts payable, and general ledger. Three help files would be provided on the disk, one for each module, and the entry code to each module would initialize the help system with the appropriate file. Requesting the help index while in accounts receivable would then only show help sections relevant to that module; help relating to accounts payable and the general ledger would not clutter the index.

This method also allows each module to have up to 255 help sections (rather than there being 255 sections for the whole program), so long as the help text can be neatly divided into two or more separate indexes.

HELP SYSTEM ERROR HANDLING

By default, the ENHCON unit performs its own internal error checking on the help system and will abort a program with an appropriate message if an error occurs. For example:

```
ENHCON unit run-time error 11
Cannot access help file in HelpInitialize
```

The interface section of the unit defines constants which represent the possible errors; these are listed above in the Windows error handling section.

The description of HelpInitialize, above, indicates possible results from an attempt to initialize the help system. Once the help system is set up, an error may be generated during a request for help. Such errors are limited to three types: ConErrHelpRead, ConErrHelpInvalid, and ConErrHeap.

ConErrHelpRead is generated when the help file has been corrupted. It usually indicates that a record in the file is missing or of an incorrect format. ConErrHelpInvalid indicates that the unit's record of the help system is corrupt. This error is likely if your program attempts to manipulate the help window directly (with OpenWindow, HideWindow, etc.). ConErrHeap is generated when a request for heap storage fails, indicating that the system is out of free memory.

In addition to providing internal error checking, the help system also allows a program to install its own error

handler by way of the global procedural variable HelpError. The applicable declarations in the interface are as follows:

```
TYPE
  HelpErrorProc  =  PROCEDURE (HErr: BYTE);

VAR
  HelpError:  HelpErrorProc;
```

HelpError is initialized to the ENHCON's own error handler and if this is all that is required there is no need to make any assignment to HelpError.

A different error handler can be installed by writing a procedure which takes a single byte-sized parameter and assigning this procedure to HelpError. Example:

```
{$F+}
PROCEDURE MyHelpError (ErrCode: BYTE);

  BEGIN
  .
  .
  END;
{$F-}

BEGIN  { Main code }
  HelpError := MyHelpError;
  .
  .
  .
```

Note that the error-handling routine must be compiled as a far subroutine; this is a requirement of Turbo Pascal's procedural variables.

When an error occurs in the help system a call is made to HelpReset to reset the help system. Control is then passed to your own error hander, which may take any action considered necessary for the particular program. The automatic reset of the help system enables many errors to be handled in a simple manner. If, for example, the user has removed the diskette containing the help file, the error "Cannot access help file" is generated. The error routine could open a window on the screen, request the user to re-insert the appropriate diskette, and then call HelpInitialize to re-load the help text.

There is one exception to the automatic reset-on-error process: If HelpInitialize is called when the help system is already initialized, no reset action will take place. When your own help-error handler is called, the help system will be reset for any error except ConErrHelpInit.

RESTRICTIONS AND TEXTMODE

There are two restrictions which should be observed when using the ENHCON unit. First, the unit makes extensive use of Turbo Pascal's Window procedure to set the size of windows on the screen. You should refrain from calling Window (in the CRT unit) directly and control all windowing on the screen using the appropriate routines in ENHCON. Second, extensive use is made of the heap to allocate window definitions, saved screen areas, and so on. The allocation routines used for this are New, Dispose, GetMem, and FreeMem, and in accordance with Borland's recommendations your program should not try to allocate and de-allocate heap storage by using Mark and Release.

By default, Turbo Pascal generates a run-time error when a program requests heap space which is not available. It is possible, however, to disable this feature by implementing your own heap-error handler (consult your reference manual for details). If your own program's error routine causes calls to New or GetMem which fail to return a nil pointer, then the ENHCON unit generates its own error message (ConErrHeap, code 10 hex.). This allows your application program to handle out-of-memory errors in any way you wish, while ensuring that ENHCON will always respond appropriately.

Finally, you should be careful when making a call to the TextMode procedure. Changing to a different text display mode causes ENHCON to dispose of all window records and to reset the help system. If you must change modes within your program, be sure to re-define any windows you might want for the new mode and re-initialize the help system.

APPENDIX A.
UNIT INTERFACES

This appendix summarizes the interface section for each unit in the library.

```
UNIT STRINGS;

{$L SUCASE}
{$L SUTRIM}
{$L SUPAD}
{$L SUTRUNC}
{$L SUCNVRT}
{$L SUMISC}

{$V-}

INTERFACE

TYPE
    FormatConfigRec =   RECORD
                            Fill,
                            Currency,
                            Overflow,
                            FracSep:    CHAR;
                        END;


CONST
    UCaseLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    LCaseLetters = 'abcdefghijklmnopqrstuvwxyz';
    Letters = UCaseLetters+LCaseLetters;
    DecDigits = '0123456789';
    HexDigits = '0123456789ABCDEF';
    OctDigits = '01234567';
    BinDigits = '01';

    { Format symbol record }
    FormatConfig: FormatConfigRec =
                    (Fill: '*';
                     Currency: '$';
                     Overflow: '?';
                     FracSep: '-');
```

```
FUNCTION LoCase(ch: CHAR): CHAR;
FUNCTION UpperCase(s: STRING): STRING;
FUNCTION LowerCase(s: STRING): STRING;
FUNCTION DuplChar(ch: CHAR; count: BYTE): STRING;
FUNCTION DuplStr(s: STRING; count: BYTE): STRING;
FUNCTION TrimL(s: STRING): STRING;
FUNCTION TrimR(s: STRING): STRING;
FUNCTION PadL(s: STRING; width: BYTE): STRING;
FUNCTION PadR(s: STRING; width: BYTE): STRING;
FUNCTION TruncL(s: STRING; width: BYTE): STRING;
FUNCTION TruncR(s: STRING; width: BYTE): STRING;
FUNCTION JustL(s: STRING; width: BYTE): STRING;
FUNCTION JustR(s: STRING; width: BYTE): STRING;
FUNCTION JustC(s: STRING; width: BYTE): STRING;
FUNCTION Precede(s,target: STRING): STRING;
FUNCTION Follow(s,target: STRING): STRING;
FUNCTION Break(VAR s: STRING; d: STRING): STRING;
FUNCTION Span(VAR s: STRING; d: STRING): STRING;
FUNCTION Replace(s,srch,repl: STRING): STRING;
FUNCTION Remove(s,srch: STRING): STRING;
FUNCTION StripBit7(s: STRING): STRING;
FUNCTION FileSpecDefault(s,path,name,extn: STRING): STRING;
FUNCTION HexStr(n: WORD; count: BYTE): STRING;
FUNCTION OctStr(n: WORD; count: BYTE): STRING;
FUNCTION BinStr(n: WORD; count: BYTE): STRING;
FUNCTION Format(n: REAL; form: STRING): STRING;
```

```
UNIT MATH;

{$N-}

INTERFACE

TYPE
    FLOAT = REAL;
```

```
UNIT MATH87;

{$N+}

INTERFACE

TYPE
    FLOAT = EXTENDED;
```

This section applies to both MATH and MATH87:

```
FUNCTION  FahrToCent(FahrTemp: FLOAT): FLOAT;
FUNCTION  CentToFahr(CentTemp: FLOAT): FLOAT;
FUNCTION  KelvToCent(KelvTemp: FLOAT): FLOAT;
FUNCTION  CentToKelv(CentTemp: FLOAT): FLOAT;
PROCEDURE InchToFtIn(Inches: FLOAT; VAR ft,ins: FLOAT);
FUNCTION  FtInToInch(ft,ins: FLOAT): FLOAT;
FUNCTION  InchToYard(Inches: FLOAT): FLOAT;
FUNCTION  YardToInch(Yards: FLOAT): FLOAT;
FUNCTION  InchToMile(Inches: FLOAT): FLOAT;
FUNCTION  MileToInch(Miles: FLOAT): FLOAT;
FUNCTION  InchToNautMile(Inches: FLOAT): FLOAT;
FUNCTION  NautMileToInch(NautMiles: FLOAT): FLOAT;
FUNCTION  InchToMeter(Inches: FLOAT): FLOAT;
FUNCTION  MeterToInch(Meters: FLOAT): FLOAT;
FUNCTION  SqInchToSqFeet(SqInches: FLOAT): FLOAT;
FUNCTION  SqFeetToSqInch(SqFeet: FLOAT): FLOAT;
FUNCTION  SqInchToSqYard(SqInches: FLOAT): FLOAT;
FUNCTION  SqYardToSqInch(SqYards: FLOAT): FLOAT;
FUNCTION  SqInchToSqMile(SqInches: FLOAT): FLOAT;
FUNCTION  SqMileToSqInch(SqMiles: FLOAT): FLOAT;
FUNCTION  SqInchToAcre(SqInches: FLOAT): FLOAT;
FUNCTION  AcreToSqInch(Acres: FLOAT): FLOAT;
FUNCTION  SqInchToSqMeter(SqInches: FLOAT): FLOAT;
FUNCTION  SqMeterToSqInch(SqMeters: FLOAT): FLOAT;
FUNCTION  CuInchToCuFeet(CuInches: FLOAT): FLOAT;
FUNCTION  CuFeetToCuInch(CuFeet: FLOAT): FLOAT;
FUNCTION  CuInchToCuYard(CuInches: FLOAT): FLOAT;
FUNCTION  CuYardToCuInch(CuYards: FLOAT): FLOAT;
FUNCTION  CuInchToCuMeter(CuInches: FLOAT): FLOAT;
FUNCTION  CuMeterToCuInch(CuMeters: FLOAT): FLOAT;
FUNCTION  FluidOzToPint(FluidOz: FLOAT): FLOAT;
FUNCTION  PintToFluidOz(Pints: FLOAT): FLOAT;
FUNCTION  FluidOzToImpPint(FluidOz: FLOAT): FLOAT;
FUNCTION  ImpPintToFluidOz(ImpPints: FLOAT): FLOAT;
FUNCTION  FluidOzToGals(FluidOz: FLOAT): FLOAT;
FUNCTION  GalsToFluidOz(Gals: FLOAT): FLOAT;
FUNCTION  FluidOzToImpGals(FluidOz: FLOAT): FLOAT;
FUNCTION  ImpGalsToFluidOz(ImpGals: FLOAT): FLOAT;
FUNCTION  FluidOzToCuMeter(FluidOz: FLOAT): FLOAT;
FUNCTION  CuMeterToFluidOz(CuMeters: FLOAT): FLOAT;
PROCEDURE OunceToLbOz(Ounces: FLOAT; VAR lb,oz: FLOAT);
FUNCTION  LbOzToOunce(lb,oz: FLOAT): FLOAT;
FUNCTION  OunceToTon(Ounces: FLOAT): FLOAT;
FUNCTION  TonToOunce(Tons: FLOAT): FLOAT;
FUNCTION  OunceToLongTon(Ounces: FLOAT): FLOAT;
FUNCTION  LongTonToOunce(LongTons: FLOAT): FLOAT;
FUNCTION  OunceToGram(Ounces: FLOAT): FLOAT;
FUNCTION  GramToOunce(Grams: FLOAT): FLOAT;
```

```
UNIT TIME;

{$V-}

{$L DATE}
{$L TIME}


INTERFACE

USES
    DOS;

TYPE
    DateString =    STRING[9];
    TimeString =    STRING[13];

    DateRec =       RECORD
                        M,D:    BYTE;
                        Y:      WORD;
                    END;

    TimeRec =       RECORD
                        H,M,S:  BYTE;
                    END;



CONST
    DateFormNumeric  = 0;       TimeFormNormal       = 0;
    DateFormAlpha    = 1;       TimeFormNormalSec    = 1;
    DateFormMDY      = 2;       TimeFormShort        = 2;
    DateFormDMY      = 3;       TimeFormShortSec     = 3;
    DateFormLower    = 4;       TimeFormMilitary     = 4;
    DateFormZeroFill = 8;       TimeFormMilitarySec  = 5;
                                TimeFormMilitaryHHMM = 6;
    FullDateFormMDY  = 0;
    FullDateFormDMY  = 1;


    TimeFormat:         BYTE        = TimeFormNormal;
    DateFormat:         BYTE        = DateFormNumeric;
    FullDateFormat:     BYTE        = FullDateFormMDY;
    TimeDelimiter:      CHAR        = ':';
    DateDelimiter:      CHAR        = '/';
    TimeParseDelims:    TimeString  = ':., '+#9;
    DateParseDelims:    DateString  = '/-., '+#9;
    TimeParseNow:       BOOLEAN     = FALSE;
    DateParseToDay:     BOOLEAN     = FALSE;
    DateParseCurYear:   BOOLEAN     = FALSE;
    DateParseCent21:    BYTE        = 0;
```

```
PROCEDURE CombineDateTime(VAR DtTm: DateTime;
                          Dt: DateRec; Tm: TimeRec);
PROCEDURE SplitDateTime(DtTm: DateTime;
                    VAR Dt: DateRec; VAR Tm: TimeRec);
PROCEDURE GetToDay(VAR Dt: DateRec);
PROCEDURE GetTimeNow(VAR Tm: TimeRec);
PROCEDURE GetDateTime(VAR DtTm: DateTime);
FUNCTION  DateValid(Dt: DateRec): BOOLEAN;
FUNCTION  TimeValid(Tm: TimeRec): BOOLEAN;
FUNCTION  DateTimeValid(DtTm: DateTime): BOOLEAN;
PROCEDURE WordToDate(w: WORD; VAR Dt: DateRec);
FUNCTION  DateToWord(Dt: DateRec): WORD;
FUNCTION  LeapYear(Y: WORD): BOOLEAN;
FUNCTION  TimeAP(Tm: TimeRec): TimeString;
PROCEDURE AdjustDate(VAR Dt: DateRec; n: INTEGER);
PROCEDURE AdjustTime(VAR Tm: TimeRec; n: LongInt);
PROCEDURE AdjustDateTime(VAR DtTm: DateTime; n: LongInt);
PROCEDURE SetLastDay(VAR Dt: DateRec);
FUNCTION  DayOfWeek(w: WORD): BYTE;
FUNCTION  DayOfWeekStr(d: BYTE): DateString;
FUNCTION  MonthStr(M: BYTE): DateString;
FUNCTION  DayOfMonthStr(D: BYTE): DateString;
FUNCTION  DateStr(Dt: DateRec): DateString;
FUNCTION  FullDateStr(Dt: DateRec): STRING;
FUNCTION  TimeStr(Tm: TimeRec): TimeString;
FUNCTION  DateParse(s: STRING; VAR Dt: DateRec): BOOLEAN;
FUNCTION  TimeParse(s: STRING; VAR Tm: TimeRec): BOOLEAN;
```

```
UNIT STDERR;

{$L STDERR}

INTERFACE

PROCEDURE WriteStdErr(s: STRING);
```

```
UNIT CRTCLERR;

INTERFACE

TYPE
    ErrorString =   STRING[20];


PROCEDURE CriticalErrorDOS;
PROCEDURE CriticalErrorTP;
PROCEDURE CriticalErrorOwn(ErrAddr: POINTER);
FUNCTION  CriticalErrorMsg(n: BYTE): ErrorString;
```

```
UNIT ENHCON;

{$V-}

{$L CRTVDU}
{$L CRTKB}

INTERFACE

USES
    DOS,CRT,STRINGS,TIME;


TYPE
    CharSet     =   SET OF CHAR;
    ConsoleStr  =   STRING[80];

    SignalErrorProc  =  PROCEDURE(width: BYTE);

    EditFormatRec =
        RECORD
            Attribute:      BYTE;
            StartChar,
            EndChar:        CHAR;
            MarkerAttr:     BYTE;
            AllowChars,
            ExitKeys:       CharSet;
            EditKey,
            RestoreKey,
            AbortKey:       CHAR;
            NumFormat:      STRING[12];
            SignalError:    SignalErrorProc;
            Flags:          WORD;
        END;


    WindowStatus    = (Undefined,Closed,Hidden,Open,Active);
    WindowBorder    = ARRAY[1..8] OF CHAR;
    WindowJustify   = (WJustLeft,WJustCenter,WJustRight);
    WindowMovement  = (WMoveLeft,WMoveRight,
                       WMoveUp,WMoveDown);
```

```
    WindowDefinition =
        RECORD
            X1, Y1, X2, Y2:     BYTE;
            DefaultAttr:        BYTE;
            DefaultCrsrHide:    BOOLEAN;
            DefaultCrsrSize:    WORD;
            Border:             WindowBorder;
            BorderAttr:         BYTE;
            HdrText, FtrText:   ConsoleStr;
            HdrAttr, FtrAttr:   BYTE;
            HdrPos,  FtrPos:    WindowJustify;
            Flags:              BYTE;
        END;

    HelpConfiguration =
        RECORD
            WindowID:           BYTE;
            HelpFileName:       ConsoleStr;
            X1, Y1, X2, Y2:     BYTE;
            NormalAttr,
            IndexAttr,
            SelectAttr:         BYTE;
            Border:             WindowBorder;
            BorderAttr:         BYTE;
            HdrText, FtrText:   ConsoleStr;
            HdrPos,  FtrPos:    WindowJustify;
            HdrAttr, FtrAttr:   BYTE;
            GeneralKey,
            ContextKey,
            LastHelpKey,
            MoveWindowKey:      CHAR;
            Flags:              BYTE;
        END;

    HelpErrorProc   = PROCEDURE(HErr: BYTE);

CONST
                                    WFlagClrOpen    = $01;
    EdFlagTrimL     = $0001;        WFlagClrClose   = $02;
    EdFlagTrimR     = $0002;        WFlagClrHide    = $04;
    EdFlagPadL      = $0004;        WFlagRestore    = $08;
    EdFlagPadR      = $0008;        WFlagShowBrdr   = $10;
    EdFlagUpper     = $0010;        WFlagWriteBrdr  = $20;

    EdFlagFlushKB   = $0100;        HFlagPageText   = $01;
    EdFlagInsert    = $0200;        HFlagPageInd    = $02;
    EdFlagForceIns  = $0400;        HFlagTitle      = $04;
    EdFlagInsStat   = $0800;
    EdFlagFirstClr  = $1000;        WCrsrDefault    = $FF00;
    EdFlagEdKeyExit = $2000;        WCrsrLine       = $FE00;
    EdFlagHideCrsr  = $4000;        WCrsrBlock      = $FD00;

                                    HMoveScroll     = #$FF;
```

```
    WBorder1:       WindowBorder  =
        (#218,#196,#191,#179,#217,#196,#192,#179);
    WBorder2:       WindowBorder  =
        (#201,#205,#187,#186,#188,#205,#200,#186);
    WBorderV1H2:    WindowBorder  =
        (#213,#205,#184,#179,#190,#205,#212,#179);
    WBorderH1V2:    WindowBorder  =
        (#214,#196,#183,#186,#189,#196,#211,#186);

    MonoNone        = $00;
    MonoUnderline   = $01;
    MonoNormal      = $07;
    MonoIntenseUL   = $09;
    MonoIntense     = $0F;
    MonoReverse     = $70;


{ ASCII control codes }

    NUL = #$00;     { Null }
    SOH = #$01;     { Start Of Header }
    STX = #$02;     { Start of Text }
    ETX = #$03;     { End of Text }
    EOT = #$04;     { End Of Transmission }
    ENQ = #$05;     { Enquiry }
    ACK = #$06;     { Acknowledge }
    BEL = #$07;     { Bell }
    BS  = #$08;     { Backspace }
    HT  = #$09;     { Horizontal Tab }
    LF  = #$0A;     { Line Feed }
    VT  = #$0B;     { Vertical Tab }
    FF  = #$0C;     { Form Feed }
    CR  = #$0D;     { Carriage Return }
    SO  = #$0E;     { Shift Out }
    SI  = #$0F;     { Shift In }
    DLE = #$10;     { Data Link Escape }
    DC1 = #$11;     { Device Control 1 }
    DC2 = #$12;     { Device Control 2 }
    DC3 = #$13;     { Device Control 3 }
    DC4 = #$14;     { Device Control 4 }
    NAK = #$15;     { Negative Acknowledge }
    SYN = #$16;     { Synchronous idle }
    ETB = #$17;     { End Transmission Block }
    CAN = #$18;     { Cancel }
    EM  = #$19;     { End of Medium }
    SUB = #$1A;     { Substitute }
    ESC = #$1B;     { Escape }
    FS  = #$1C;     { File Separator }
    GS  = #$1D;     { Group Separator }
    RS  = #$1E;     { Record Separator }
    US  = #$1F;     { Unit Separator }
    DEL = #$7F;     { Delete }
```

```
PoundSign = #$9C;

StandardChars = [#32..#126];

KeyIns   = #$80;
KeyDel   = #$81;
KeyUp    = #$82;
KeyDown  = #$83;
KeyLeft  = #$84;          KeyCLeft    = #$8A;
KeyRight = #$85;          KeyCRight   = #$8B;
KeyHome  = #$86;          KeyCHome    = #$8C;
KeyEnd   = #$87;          KeyCEnd     = #$8D;
KeyPgUp  = #$88;          KeyCPgUp    = #$8E;
KeyPgDn  = #$89;          KeyCPgDn    = #$8F;

KeyA0    = #$90;          KeyAHyphen  = #$9A;
KeyA1    = #$91;          KeyAEquals  = #$9B;
KeyA2    = #$92;
KeyA3    = #$93;
KeyA4    = #$94;          KeySTab     = #$9D;
KeyA5    = #$95;          KeyCPrtSc   = #$9E;
KeyA6    = #$96;
KeyA7    = #$97;
KeyA8    = #$98;
KeyA9    = #$99;

KeyF1    = #$A0;          KeySF1  = #$B0;
KeyF2    = #$A1;          KeySF2  = #$B1;
KeyF3    = #$A2;          KeySF3  = #$B2;
KeyF4    = #$A3;          KeySF4  = #$B3;
KeyF5    = #$A4;          KeySF5  = #$B4;
KeyF6    = #$A5;          KeySF6  = #$B5;
KeyF7    = #$A6;          KeySF7  = #$B6;
KeyF8    = #$A7;          KeySF8  = #$B7;
KeyF9    = #$A8;          KeySF9  = #$B8;
KeyF10   = #$A9;          KeySF10 = #$B9;

KeyCF1   = #$C0;          KeyAF1  = #$D0;
KeyCF2   = #$C1;          KeyAF2  = #$D1;
KeyCF3   = #$C2;          KeyAF3  = #$D2;
KeyCF4   = #$C3;          KeyAF4  = #$D3;
KeyCF5   = #$C4;          KeyAF5  = #$D4;
KeyCF6   = #$C5;          KeyAF6  = #$D5;
KeyCF7   = #$C6;          KeyAF7  = #$D6;
KeyCF8   = #$C7;          KeyAF8  = #$D7;
KeyCF9   = #$C8;          KeyAF9  = #$D8;
KeyCF10  = #$C9;          KeyAF10 = #$D9;
```

```
    KeyAA    = #$E1;          KeyAP  = #$F0;
    KeyAB    = #$E2;          KeyAQ  = #$F1;
    KeyAC    = #$E3;          KeyAR  = #$F2;
    KeyAD    = #$E4;          KeyAS  = #$F3;
    KeyAE    = #$E5;          KeyAT  = #$F4;
    KeyAF    = #$E6;          KeyAU  = #$F5;
    KeyAG    = #$E7;          KeyAV  = #$F6;
    KeyAH    = #$E8;          KeyAW  = #$F7;
    KeyAI    = #$E9;          KeyAX  = #$F8;
    KeyAJ    = #$EA;          KeyAY  = #$F9;
    KeyAK    = #$EB;          KeyAZ  = #$FA;
    KeyAL    = #$EC;
    KeyAM    = #$ED;
    KeyAN    = #$EE;
    KeyAO    = #$EF;


{ Error codes }

    ConErrXY            = $01;
    ConErrBorderXY      = $02;
    ConErrMove          = $03;
    ConErrOpen          = $04;
    ConErrClosed        = $05;
    ConErrHidden        = $06;
    ConErrNotHidden     = $07;
    ConErrZero          = $08;
    ConErrDefined       = $09;
    ConErrUndefined     = $0A;
    ConErrReturn        = $0B;
    ConErrHeap          = $10;
    ConErrHelpRead      = $11;
    ConErrHelpInit      = $12;
    ConErrNoHelpFile    = $13;
    ConErrHelpFormat    = $14;
    ConErrHelpIndex     = $15;
    ConErrHelpInvalid   = $16;
    ConErrHelpStkFull   = $17;
    ConErrHelpStkEmpty  = $18;


    InsKeyEnable:       BOOLEAN = FALSE;
    CursorInsert:       BOOLEAN = FALSE;
    WindowCheck:        BOOLEAN = TRUE;
    EnhConHaltError:    WORD    = 0;
    HelpContext:        BYTE    = 0;

VAR
    HelpError:  HelpErrorProc;
```

```
FUNCTION  ColorDisplay: BOOLEAN;
FUNCTION  MaxCursorSize: BYTE;
PROCEDURE SetCursor(size: WORD);
FUNCTION  GetCursor: WORD;
PROCEDURE HideCursor(hide: BOOLEAN);
FUNCTION  CursorHidden: BOOLEAN;
PROCEDURE OrigCursor;
PROCEDURE LineCursor;
PROCEDURE BlockCursor;
FUNCTION  GetDisplayPage: BYTE;
FUNCTION  GetDisplayBase: WORD;
PROCEDURE GetMaxXY(VAR x,y: BYTE);
PROCEDURE FlushKB;
FUNCTION  CapsLock: BOOLEAN;
FUNCTION  NumLock: BOOLEAN;
FUNCTION  ScrollLock: BOOLEAN;
FUNCTION  InsertLock: BOOLEAN;
PROCEDURE ForceInsert(Ins: BOOLEAN);
PROCEDURE StdSignalError(width: BYTE);
FUNCTION  EditString(form: EditFormatRec;
                     VAR s: STRING; width: BYTE): CHAR;
FUNCTION  EditInt(form: EditFormatRec;
                     VAR i: LongInt; min,max: LongInt): CHAR;
FUNCTION  EditReal(form: EditFormatRec;
                     VAR r: REAL; min,max: REAL): CHAR;
FUNCTION  EditDate(form: EditFormatRec;
                     VAR Dt: DateRec): CHAR;
FUNCTION  EditTime(form: EditFormatRec;
                     VAR Tm: TimeRec): CHAR;
FUNCTION  WindowResult: BYTE;
FUNCTION  ConErrorMsg(ErrNum: BYTE): ConsoleStr;
PROCEDURE GetWindowDef(WindowID: BYTE;
                       VAR d: WindowDefinition);
FUNCTION  WindowStat(WindowID: BYTE): WindowStatus;
FUNCTION  CurrentWindow: BYTE;
PROCEDURE DefineWindow(WindowID: BYTE; d: WindowDefinition);
PROCEDURE PurgeWindow(WindowID: BYTE);
PROCEDURE OpenWindow(WindowID: BYTE);
PROCEDURE SelectWindow(WindowID: BYTE);
PROCEDURE CloseWindow(WindowID: BYTE);
PROCEDURE HideWindow(WindowID: BYTE);
PROCEDURE ShowWindow(WindowID: BYTE);
PROCEDURE RelocateWindow(WindowID: BYTE; X,Y: BYTE);
PROCEDURE MoveWindow(WindowID: BYTE;
                     Direction: WindowMovement);
PROCEDURE WriteWindow(s: ConsoleStr);
PROCEDURE HelpReset;
PROCEDURE PushHelpContext(NewContext: BYTE);
PROCEDURE PopHelpContext;
PROCEDURE HelpInitialize(h: HelpConfiguration);
PROCEDURE TextMode(Mode: WORD);
FUNCTION  ReadKey: CHAR;
```

APPENDIX B.
CODE DEPENDENCIES

TPU DEPENDENCIES

Library unit:	Uses units:
STRINGS	DOS
MATH/MATH87	No dependencies
TIME	DOS, STRINGS
STDERR	No dependencies
CRTCLERR	DOS
ENHCON	DOS, CRT, STRINGS, TIME

ASSEMBLER MODULE DEPENDENCIES

Library unit:	Requires OBJ modules:
STRINGS	SUCASE, SUTRIM, SUPAD, SUTRUNC, SUCNVRT, SUMISC
MATH/MATH87	No assembler modules
TIME	DATE, TIME
STDERR	STDERR
CRTCLERR	No assembler modules
ENHCON	CRTVDU, CRTKB

OPERATING SYSTEM & FIRMWARE DEPENDENCIES

ENHCON unit requires compatibility at BIOS services level. All other units require only compatibility at operating-system level for DOS 2.0 or greater.

HELP CONVERSION UTILITY DEPENDENCY

The following units are required in order to recompile HELPCONV:

DOS, STDERR, CRTCLERR, STRINGS

APPENDIX C.
REVISION HISTORY

VERSION 1.0, August 1991

The name of this library package became Turbo Pascal Library when the code was released to the public domain in August 1991.

VERSION 1.1, December 1991

The most noticeable change is the integration of ENHCRT, WINDOWS, and HELP into ENHCON, putting all console-related routines together into one unit. Programs written with version 1.0 of Turbo Pascal Library can now have references to the original three units replaced with a single "USES ENHCON;" declaration.

The error codes for the old HELP unit have been changed, to enable them to be integrated with the windows-related errors. All WErr and HErr codes should be replaced with the appropriate ConErr code. The function WindowErrorMsg no longer exists; you should now use the ConErrorMsg function instead. The two variables WindowHaltError and HelpHaltError have been replaced with the single variable EnhConHaltError.

After using the help system in version 1.0, ReadKey would return the value of the help key to the calling program. This is no longer the case; ReadKey waits for another key to be pressed before returning.

A bug which prevented the help system from being activated during an edit routine has been fixed.

A problem may have been encountered with version 1.0 if your program changed text modes. The WINDOWS unit would dispose of all window records, thus corrupting the HELP unit's record of the on-line help system. A call to TextMode now automatically resets the help system.

The DATE unit has been modified to allow numeric dates to be forced to American or British format, regardless of what the system configuration may be. The constants to be assigned to DateFormat have been altered to accommodate this change.

The TPU files that were supplied with version 1.0 were compiled under Turbo Pascal 5.5. To avoid potential confusion for people using different versions of the compiler, the TPU files are not now supplied. A batch file

is now provided which will compile all six units.

You may have also encountered a problem with spacing in the source code of version 1.0. This was due to the use of tabs which can be interpreted differently by different editors. All source code now uses ASCII spaces in place of tabs to avoid this problem.

VERSION 2.0, May 1993

This major revision of the library has introduced several additions and changes to units and seen a revision of the disk distribution arrangement and example support files. Details of specific changes will be found in the appropriate section of this manual.

The Format function in the STRINGS unit has been extended to permit fractions to be displayed in vulgar form as well as decimals. Several other options have also been added: left-justification, padding with a definable character, and absolute value display. The method for using a floating currency symbol has also been amended slightly, but you will not need to change your code unless it previously used the British pounds-sterling sign. This change has also corrected a bug in earlier versions that prevented a zero-field from being blanked with the "B" option if the British symbol was used.

The new units MATH and MATH87 have been added to provide measurement conversion routines.

The DATE and TIME units have been integrated into a single unit and several new subroutines have been added. These changes allow easier mixing of this package's routines and Turbo Pascal's library routines for handling combined dates and times. The TimeStr function supports two new formats, TimeFormShort and TimeFormShortSec. The constants relating to all time formats have been revised, so you should check any of your code that uses the old TimeStr function. Date and time parsing have also been extended to allow null input to return the current date or time. DateParse will also now permit date entries to default to the current year and to allow two-digit entry of 21st century dates. These options are controlled by global unit variables. The default character assigned to TimeDelimiter for use by TimeStr is now a colon if your system is set to American configuration and a period if it is set to British configuration.

A help-context stack has been added to the ENHCON unit to simplify the use of context-sensitive help with application programs that have many subroutines. The procedures

PushHelpContext and PopHelpContext have been implemented to support this feature and two new error codes have been added to the ConErr list. Existing code using version 1.1 of the library will work without change, although you may wish to take advantage of the new stack when updating your code.

A bug has also been discovered in ENHCON that prevented the ConErrHelpInit error code from being reported properly if a call was made to HelpInitialize when the help system was already initialized. This has been corrected.

VERSION 2.1, June 1993

A compatibility problem with the new Turbo Pascal 7.0 compiler has been discovered. Earlier versions of the library will falsely generate an "Out of memory" error when a null string is passed to the WriteWindow procedure in ENHCON, due to a change in the GETMEM routine from earlier compilers. Since WriteWindow is used by the windowing and help routines, you may have also experienced the same problem with some other operations when using Turbo Pascal 7.0. Version 2.1 of the library has corrected this problem.

APPENDIX D.
DISTRIBUTION POLICY

This package is released for free distribution and use by individuals or businesses. My reasons for not charging a registration fee are fairly simple: I am not in a position to provide full support for any of the software I write. If you feel that you should make some sort of financial contribution for the software you use, then I suggest a donation of a few dollars to a charity.

Although there is no "official" support for this package, you are welcome to write with any ideas or suggestions. If you have any problems that you think are caused by a bug in the software, please feel free to write me and I will try to help (letters to England from the United States require 50 cents postage for the first half ounce). If you just want to pass on a short comment or let me know that you find the software useful, a picture-postcard from your home-town would be most welcome (40 cents postage).

Please pass on copies of this package to anyone who may find it useful.