



WinBasic

Zimmer Informatik, Düsseldorf, Germany
1991



About this manual

This manual is designed to be as small as possible; programming languages are learnt by their use, not from books. Nevertheless, this manual should contain most of the reference information you need to write *WinBasic* programs. If you need to know something and can't find it in the manual, the answer is to experiment: write a small program to answer your question.

Acknowledgement

I am very thankful to Michael R. Fitzpatrick, Elyria, Ohio, who received a german copy of *WinBasic* and encouraged me to produce a version in english. He also launched this version on the american shareware market. Without his help you would not hold *Winbasic* in your hands.

Notational conventions

In this manual the following notational conventions are used:

Menu commands:

Menu names are shown together with a command name. For example: you can use the **File Save** command to save the current program to disk.

Bold type

Bold type indicates words that have special significance in *WinBasic*, for example menu names and commands within menus. For instance, **File** is the name of a menu; **File Save** is the name of a command.

Italics

Italics indicate text that you would replace with a particular word or other group of characters. For example, when the manual suggests that you type "*filename*", we do not mean you to type "*filename*"; rather, you should replace *filename* with the name of a particular file.

Computer font

Type that appears on the screen and words you need to type yourself appears in a typeface resembling the screen or printer type of many systems.

<Key>

To make the use of key presses clearer, sometimes a word or words and not a symbol is used; where this is so, the words are surrounded by angled brackets, for example <Enter> or <Ctrl F2>.

Different standard Windows fonts are used for type setting; Arial, Times New Roman, Courier New and Wingdings. If one of these fonts is not installed on your machine, parts of this manual might look a little bit odd.

INSTALLATION

WinBasic is not copy protected, does not come on a multitude of floppy disks and does not require an elaborate installation procedure. Essentially, all you need to do is copy the files from the distribution floppy into a directory onto your hard disk, then tell your machine where you have put them.

Create a subdirectory

Run the File Manager and use the **File Create Directory** command, and type:

```
\winbasic
```

into the ensuing dialog box.

Copy the files

Insert the distribution disk into drive A:. Use the **File Copy** command. In the ensuing dialog box, type <Shift Tab> (to move the focus to the first edit line), and type:

```
a:
```

followed by <Tab> (to move the focus to the second edit line). Type:

```
\winbasic
```

into this, and press <Enter>.

Modify AUTOEXEC.BAT

You now have to tell your computer how to find the files you have just copied. Use the **File Run** command and type:

```
notepad \autoexec.bat
```

into the ensuing dialog box. This runs the NotePad editor on the AUTOEXEC.BAT file in your computer's root directory. Scan through the file, and find the line that starts with:

```
set path= or path=
```

Notice that this is followed by a list of directories, separated by semicolons. Add the following to the end of the line:

```
;\winbasic
```

Save the modifications using the **File Save** command.

Modify WIN.INI

Use the **File Open** command of NotePad to open a file in your Windows subdirectory called WIN.INI. Page down through this file, until you see a section headed

```
[Extensions]
```

At the end of a list of similar looking lines, add the following:

```
bas=winbasic.exe ^.bas  
cod=runwinb.exe ^.cod
```

This tells Windows to automatically run *WinBasic* or its runtime module if you click onto a filename with the extensions .BAS or .COD.

Save the modifications using the **File Save** command and then exit NotePad with the **File Exit** command. Terminate your Windows session and re-boot your computer to bring into effect the changes you have just made.

Modify the Program Manager

In order to start *WinBasic* by simply doubleclicking its item in the Program Manager you have to install it there. Activate the program group into which you want to insert *WinBasic*. Choose **File New** from the Program Manager's menu, then choose to install a new program in the ensuing dialog box. In the next dialog box type into the first three edit fields:

1. WinBasic
2. winbasic
3. \winbasic

Close the box by "OK" and you will find *WinBasic*'s icon in the active program group.



THE WINBASIC ENVIRONMENT

This section describes how to use the *WinBasic* environment; it covers:

- ◆ Starting *WinBasic*;
- ◆ Editing and running *WinBasic* programs;
- ◆ Using the runtime module.

Starting WinBasic

The heart of *WinBasic* is the program WINBASIC.EXE. You can start *WinBasic* by doubleclicking this filename with the mouse in the File Manager of Windows or by doubleclicking its icon in the Program Manager or by typing

```
win winbasic
```

from the DOS prompt.

WinBasic can also be started by doubleclicking on a file with the

extension .BAS, since the installation procedure added the following line to the [Extensions] section of your WIN.INI file:

```
bas=winbasic.exe ^.bas
```

The selected .BAS file will immediately be loaded into the editor. Also, you can run *WinBasic* by specifying a .BAS file from the DOS-Prompt by typing, for example:

```
win hello.bas
```

After starting *WinBasic* the window of the editor appears on the screen and the window of the *WinBasic* program HELLO.BAS is shown as an icon on the lower part of the screen (the icon will usually be blank; details of how to design your own icons are given in the section on the BITMAP.EXE utility).

The *WinBasic* program can be started by the **Run Start** command. When a program is started, its iconic window is opened to full size. This window returns to an icon automatically when the program is terminated with a STOP statement or via the **Close** command in it's system menu.

The editor

Your interface to *WinBasic* is mainly through the editor. It is used both to write programs and give commands to the *WinBasic* compiler.

The window of the editor has three parts:

- ◆ The menu bar;

- ◆ The workspace, which displays the program text;
- ◆ The status line at the lower border.

The status line is used to display the default directory and error messages from the compiler.

Menus

The menu bar contains six menu titles. The first four contain commands related to the creation and modification of *WinBasic* programs, while the fifth, **Run**, contains commands that run the programs you write. The last menu, **Help**, is used to provide a quick reference to *WinBasic* statements.

File

The **File** menu contains commands that load and save programs.

New

This erases the current program from the editor. If the program has been modified, you are first asked if you wish to save the changes.

Open...

This loads an existing program into the editor. If there is a modified program already in the editor, you are given the chance to save it first.

Save

This saves the current program into the file from which it was loaded. If the program was created from scratch, **Save** works the same as **Save As**.

Save As...

This allows the saving of the current program into a file with any name.

Exit

This ends the session with *WinBasic*.

About...

This displays the version number of the software and a copyright notice.

Edit

The **Edit** menu contains commands that operate on the text of the current program. With the exception of **Undo** and **Paste** they work with the selected text.

Undo

This revokes the latest change, if possible.

Cut

This copies the selected text into the clipboard and deletes it from

the program.

Copy

This copies the selected text into the clipboard. The text itself remains unchanged.

Paste

This inserts the text in the clipboard into the program at the current position of the text cursor.

Clear

This deletes the selected text. The text is not copied into the clipboard.

Select All

This selects the whole program for editing.

Search

Find...

This allows you to find a specific piece of text within the program. A dialog box is displayed, into which you type the piece of text.

The **Ignore case** check box allows case to be ignored.

The **Replace with** check box allows the text searched for to be replaced by another piece of text.

The search starts from the current cursor position, and wraps around from the end of text to the beginning.

Find Next

This repeats the action of the last **Find** command.

Options

Insert Mode

simply toggles the editor between insert and overwrite mode.

Autosave

toggles the automatic backup facility. If this option is enabled (ticked) then the current program will be saved to a file with the extension .BKP every 500 keystrokes.

Debug

This directs the compiler to generate debugging information for the program. A program run with this option enabled checks array indexes for overflow and allows the execution of TRACE statements (refer to the reference section for more details).

Because the debugging information makes the program bigger and slightly slower, you should set this option off as soon as it is no longer needed.

Insert mode and **Autosave** are on by default; **Debug** is off.

Run

This menu contains commands which are concerned with the compilation and execution of a program.

Start

This is a combination of **Compile** and **Run**.

Compile

This compiles the current program. If the program contains errors, the first is displayed in the status line at the lower border of the editor window. The text cursor is set to the position within the program where the error occurred.

Run

This executes a program that compiled without error. The program's window is displayed in normal size and the editor window goes into the background. If the executing program has no **Exit** command in its menu, it can be stopped by the **Close** command in its system menu.

After execution of the program, the program's window returns to an icon and the editor window returns to the foreground.

Error

This shows the next error of an incorrectly compiled program. The text cursor is placed at the occurrence of the error in the program. An easier method of displaying the next error is to simply click the left mouse button while the mouse cursor is in

the status line. After displaying the last error the status line shows the current subdirectory, with the next **Error** command (or mouse click) it starts from the beginning again. Up to twenty errors are shown per compilation.

Code

This generates a compiled version of the program and stores it in a file with extension .COD. This file can be executed by the runtime module RUNWINB.EXE.

Help

This command displays a small amount of help for *WinBasic* statements. You select a statement or function of *WinBasic* within the program and choose **Help** in the menu. A window is displayed that shows you the syntax and the parameters with their possible values.

The runtime module

WinBasic is supplied with a runtime module called RUNWINB.EXE. This executes a *WinBasic* program which has been produced by the **Run Code** command. This produces a file with the extension .COD, which contains a compact, fast-executing version of the program. The runtime module allows *WinBasic* programs to be run without access to the compiler or the source code. RUNWINB.EXE may be freely distributed together with .COD files to let other people use the software you develop with *WinBasic*. To run RUNWINB.EXE they also need

all the files matching *Z*.DLL* which are supplied with *WinBasic*; these also may be distributed freely.

A compiled *WinBasic* program can be executed when its filename is doubleclicked with the mouse from the File Manager. Alternatively, the execution can be initiated at the start of Windows by typing

```
win hello.cod
```

from the DOS prompt.

In either case, the [Extensions] section of the WIN.INI file must contain the entry

```
cod=runwinb.exe ^.cod
```

which it does, if you followed the proposed installation procedure.



WINBASIC

This section is not a complete introduction to programming in BASIC. Rather, its purpose is to show the specialities of *WinBasic*. If you have problems writing BASIC programs, you should consult other, more appropriate, sources.

Program statements

Programs consist of a sequence of statements. Several statements can occur on one line of program text, separated by a colon. However, there are some exceptions:

- ◆ If statements follow a single-line IF statement such as:

```
If a% < 0 Then a% = -a% : e% = f% : Gosub Thing(f%)
```

then they are executed only if the IF expression is true;

- ◆ No statements may follow a label;
- ◆ No statements may follow the "structured" statements.

There are two types of statements; executable and non-executable. Executable statements are processed at the execution of the program and lead to a change in the data or the logical state of the program. Non-executable statements define or declare code or data, for example DIM, RECTYPE or DECLARE.

Programs can be structured using the following pairs of statements:

- ◆ Subroutine ... Endsub
- ◆ For ... Next/Endfor
- ◆ If ... Endif
- ◆ While ... Wend
- ◆ Repeat ... Until

These build a block between the beginning (Subroutine, For, If, While, Repeat) and the ending statement (Endsub, Next/Endfor, Endif, Wend, Until). These statements often are called "structured statements".

No other statements may follow these structured statements on the same line. The lines between the pair of structured statements may contain one or more lines of other statements.

The words in the text of a program can be one of two types; reserved words and those defined by the programmer. Reserved words are those which constitute the language, like IF, SUBROUTINE, GOTO, including the intrinsic functions like SIN(x), TRIM\$(a\$). The programmer defines variables and constants, which must not have the names of reserved words.

Words in the program text are not case dependent; the variable ABC\$ may be written Abc\$, abc\$ and so on.

An important part of modern programming languages are subroutines (or procedures) which are implemented in *WinBasic* through the pair of statements SUBROUTINE ... ENDSUB. Subroutines are closed parts within the program, which can be used at different places of the program, but are written only once. Subroutines are placed in *WinBasic* at the beginning of a program, because they have to be declared before they can be called. The main program begins after the ENDSUB statements of the last subroutine. Note that this ordering can be overridden by the DECLARE statement, so that a subroutine can call another that has yet to be defined.

Program structure

Introduction

Traditional BASIC programs are essentially executed from the first to the last statement, apart from gotos, loops and so on. All actions of the user are controlled by the program; for example a user can choose an item from a menu only when the program chooses to display it, data can only be entered when prompted for, and so on.

(The user is the person who uses a program when it is run. The user is to be distinguished from the programmer, who creates a program with *WinBasic*.)

Windows inverts this. The user can select an item from a menu at any time; they can use the mouse or keyboard at any time; they can change between applications at any time. A program running in the Windows environment is very different to one running in

an MS-DOS environment, since the program becomes subservient to the user.

The main program

From the point of view of Windows the main routine of a program created with *WinBasic* (or any other language) is a service subroutine, which is called by Windows whenever it seems appropriate. This is the case when the user clicks on a command in the menu or presses a key on the keyboard or the mouse. The main routine of the *WinBasic* program is then responsible for carrying out the action specified by the user.

Messages

When the main program should react to the user it is sent a "message" by Windows. This message is a code which tells the program what activity has taken place. A *WinBasic* main program has special labels - the Windows labels - for all messages it wants to respond to. The main program will start execution at these labels after the corresponding message was sent.

For example, here is a *WinBasic* main program which responds to keystrokes from keyboard and clicks from the mouse:

```
Rem *** Main program ***

Print "Test mouse and key messages"

_MOUSE:
  Mouse Button%, x%, y%
  MoveTo x%, y%
  Print "Button "; Button%
```

```
_KEY:
    Print inkey$()
```

This small program consists of an initialization part (`Print "Test ..."`) and two windows labels which are entry points for messages from Windows:

- ◆ `_MOUSE` is the entry point when a key on the mouse is pressed.
- ◆ `_KEY` is the entry point when the user presses a key on the keyboard. This must be a printable character or one of the function keys F1 to F9 or Shift F1 to Shift F9.

All Windows labels are prefixed with an underscore to distinguish them from ordinary BASIC labels.

At the start of the program above, the portion before the first Windows label is executed as per a normal BASIC program. When the first Windows label is encountered, execution of the program suspends.

If the user presses a key on the keyboard, the message `KEY` is generated by Windows. The program now continues execution at the Windows label `_KEY` and runs under the control of the following statements until another Windows label (not an ordinary BASIC label) or the physical end of the program is reached. Then the program is suspended again and waits for new messages.

If one of the mouse buttons is pressed, the message `MOUSE` is generated by Windows and the program continues execution at

the Windows label `_MOUSE`. The statements after this label obtain information about the pressed button together with the coordinates of the mouse when the button was pressed and displays this information by printing the number of the pressed button at these coordinates. Because the Windows label `_KEY` is the next statement, the program is suspended again, waiting for further messages.

The program can be terminated by choosing the **Close** command from its system menu.

Program termination

Your program may also need to "clean-up" before it terminates. This can be accomplished by placing an **Exit** command in a menu, and responding to that command by calling a clean-up routine before terminating. For example:

```
Dim Menu$(2)

Subroutine ProgEnd()
...
Endsub

Rem *** Main program ***

Menu$(1) = "DemoMenu E&xit Hello"
Menu$(2) = ""
Menu Menu$

_1001:
  Gosub ProgEnd()
  Stop

_1002:
  Print "Hello"
```

A menu is created in the initialization part of the main program. The first item of the string array `Menu$` is set to the title and contents of the first (and, in this case, only) menu. **DemoMenu** is the title of the menu, and **Exit** and **Hello** are commands within it. Note the use of the ampersand to prefix the keyboard shortcut character. The second item in `Menu$` is set to an empty string to signify the end of the menu definition. The following statement, `Menu Menu$`, generates a menu from the string array `Menu$`.

The program then suspends and waits for messages from Windows. The program responds to only two messages: the codes 1001 and 1002, which correspond to the first and second commands in the menu. The program resumes execution after receiving these messages at Windows labels `_1001` and `_1002`, respectively.

Messages from menus

The codes of menu related messages are computed when the messages are generated. The first menu in the menu bar has codes beginning with 1000, the second beginning with 2000 and so on. The commands within a menu are numbered beginning with these codes, the menu titles (which are shown on the menu bar) getting the codes 1000, 2000 etc. Therefore, in the above example, the menu title **DemoMenu** has the message 1000, the **Exit** command within that menu has the message 1001, and the **Hello** command has the message 1002. Note that in this case the message 1000 is not received by the program because it is responded internally by Windows in rolling down the menu.

Overall program structure

Structurally, a *WinBasic* program consists of four parts, which must be in the following order:

1. Global variable declarations;
2. Subroutine definitions;
3. Global variable initializations;
4. Windows labels.

Other examples of programming menus are found below.

Dialog boxes

A very important element of Windows programs are dialog boxes. These are the common form of data entry and communication between program and user.

A dialog box is "driven" by a subroutine that responds to Windows messages in a similar manner to the *WinBasic* main program.

For example:

```
Dim Menu$(3)

Subroutine DialogDemo()
  Dialog 10, 10, 120, 40, 0, 0, "Dialog Demo"
  Dialog 4, 12, 30, 14, -1, 5, "Prompt:"
  Dialog 40, 10, 40, 14, 10, 18, "ABC"
  Dialog 90, 10, 20, 12, 11, 13, "OK"
  Dialog
```

```

_1:                                'Enter
_11:
_DlgItem 10, Entry$
Print Entry$
Dialog @

2:                                'ESC
Dialog @
Endsub

Rem *** Main program ***

Menu$(1) = "Dialog"
Menu$(2) = "E&xit"
Menu$(3) = ""
Menu Menu$

_1000:
_Gosub DialogDemo()
_2000:
_Stop

```

The statement `Menu Menu$` generates a menu which consists of two single commands on the menu bar. In other words, the commands **Dialog** and **Exit** are not menu titles but act like commands within menus; they send messages 1000 and 2000 when clicked by the user.

When the command **Dialog** is clicked, the program calls the subroutine `DialogDemo()`. This subroutine is responsible for the creation, control and termination of the dialog box.

If a program has more than one dialog box, each is driven by a subroutine of its own.

The dialog box is created by the statements of the form:

Dialog x%, y%, dx%, dy%, id%, type%, text\$

Each statement of this form creates one object in a dialog box.

The first four parameters are values for the starting point and size of the items of the dialog box.

id% is the individual identification of an item and must be unique within a dialog box (but may be used in different dialog boxes).

type% defines the type of the item. In the example the types used are:

type% = 0 generates a frame around the dialog box;

type% = 5 generates a left-justified piece of text.

type% = 18 creates an edit control for text entry;

type% = 13 creates a push button;

The full list of types is given in the reference section.

The sequence of DIALOG statements is completed by the simple statement

Dialog

which indicates the end of the dialog creation process and displays the dialog box. The subroutine is suspended here and waits for messages from Windows. Messages are sent when the user clicks on one of the items in the dialog box or presses the

<Enter> or <Esc> keys.

When an item in the dialog box is clicked, the identification *id%* from the dialog definition is sent as a message to the dialog box subroutine. The <Enter> and <Esc> keys generate the messages 1 and 2, respectively.

When the user clicks the OK push button the message 11 is sent by Windows to the subroutine, because the OK push button is given an id of 11 in the example above.

The statement

```
Dialog @
```

terminates the dialog box, removing it from the screen. The subroutine returns immediately. Therefore, the statement `Dialog @` works like a RETURN statement.

The statement

```
DlgItem 10, Entry$
```

obtains the data associated with *id%* 10 in the dialog box and copies it into the variable *Entry\$*. In this case, the above statement will get the string entered into the dialog's edit control.

More examples of programming dialog boxes can be found in the *Examples* section.

The Timer

There is one more Windows label, which is related to the timer.

The Windows timer facility can be activated with the `TIMER` statement (see reference). After the defined time has elapsed, the message `_TIMER` is sent which can be trapped by the corresponding Windows label:

```
_TIMER:
    'Process timer message here
```



Variables

Types

Data is stored in variables. *WinBasic* supports the data types integer (16 bit), real (32 bit), long (32 bit integer), double (64 bit real) and string (character string of variable length, up to 255).

Variables have names of up to 16 characters. The type of the variable is fixed by the last character, after the following convention:

%	Integer	<i>e.g.</i>	i%
&	Long	<i>e.g.</i>	l&
#	Double	<i>e.g.</i>	d1#
\$	String	<i>e.g.</i>	str\$

In the absence of one of these identifying characters, the variable

is assumed to be of type real.

Variable names may not contain a period because the period has a special meaning within records (see below).

The range and the precision of the data types is the same as for constants of each type; see the section on *Constants* for more details.

Scope

Variables are global to the entire program unless they are declared LOCAL or STATIC within a subroutine. LOCAL and STATIC variables within a subroutine are distinct from global variables of the same name.

Arrays and records

WinBasic supports arrays and records. Arrays are collections of variables of the same type that are addressed by a common name; a numeric index is used to access individual members within each array. Arrays are created by the DIM (short for DIMension) statement.

Records are collections of different variable types that are addressed by a common name; individual members are addressed by a variable name. The two names are linked by a period character.

Records are generated in two steps. First, the structure of the record is defined using the RECTYPE statement. Then, instances of the record are declared using the RECORD statement. Record

names cannot have a type identifying character (*i.e.* no %, &, # or \$).

Constants

In the same way that there are different data types, there are also different types of data constants.

Integer Constants

Integer constants are comprised of the digits 0 to 9 and the signs + and -. The values of integer constants must lie between -32768 and +32767, this being the range which can be represented with 16 bits. If integer constants have values outside this range, the internal value generated by *WinBasic* is undefined. The plus sign is optional for positive constants.

Long integer Constants

Long integer constants must lie in the range from -2147483648 to +2147483647. They have an "L" suffix to mark them as "long", for example -434345456L or 34L.

Real Constants

Real constants consist of numeric characters before and after a decimal point or - in scientific notation - of a mantissa and an exponent, separated by an "E". The valid range is approximately from 3.4E-38 to 3.4+38 for both positive and negative values. The precision is seven digits for the mantissa.

Double Constants

In double constants the "E" of the real constants is changed to "D". The valid range is approximately from $1.7D-308$ to $1.7D+308$, the precision 14 digits.

Because reals and doubles have a limited precision, they cannot represent every floating point value exactly. Practically, there may be difficulties when comparing numbers. For example:

```
If a = 1.0/3.0 Then Break
```

Because $1/3$ or $1.0/3.0$ can only be represented by an infinite row of digits, they cannot be matched by a real or double with 7 or 14 digits. The result of the comparison is unpredictable and depends on the way the variable `a` was computed. Reals and doubles should not be used as counter variables in FOR loops.

String Constants

String constants are composed of any printable characters enclosed by a pair of quotation marks `"`. The maximum length of a string constant is 255 characters.

Arithmetic expressions

Variables and constants are combined with arithmetic operators to form arithmetic expressions. The values of these expressions can be assigned to variables.

Arithmetic operators are +, -, *, / for the four basic forms of computing. The common rules for precedence and association are valid, which can be overridden by the use of parentheses.

Type conversion

If there are numeric items of different data types in one arithmetic expression - including the variable which stores the result - an automatic type conversion is made. Strings, arrays and records are not compatible with any other type, so an automatic type conversion is not possible. Of course the items in an array or record may be used in expressions, because they are simple data types. The automatic type conversion is always done into the type with the wider range of values. For example in the expression

```
a% * b1
```

which is formed by the integer *a%* and the real *b1*, *a%* is converted into a real before the multiplication is done. If the value of an arithmetic expression is stored in a variable through the assignment operator =, as in

```
result% = a% * b1
```

then the expression is always converted into the type of the storing variable, in this case *result%*. If this type has a lower range of values than the expression, information may be lost (in this example the digits after the decimal point are lost, because *result%* is an integer storing a real expression) and a completely wrong result may be received when the value of the expression lies outside the range of the type of the storing variable. For example, if *a% * b2* is 40000.0, *result%* cannot store it correctly

because integers can only take values from -32768 to +32767.

If constants are directly assigned to variables, it is a good idea to match the types of the constants with the types of the variables, as in

```
a = 1.0
a# = 1.0D0
```

If you write

```
a = 1
a# = 1.0
```

instead, an automatic type conversion is done, which costs extra space for program code and slows down execution.

Records

Only items within records may be part of arithmetic expressions. The exception is the assignment, which may be used thus:

```
Rectype Atype A1, A2
...
A1 = A2
```

Functions

A function may be substituted for a variable in any arithmetic expression. These functions may be the built-in functions of *WinBasic* like VAL() or VAL%(), or subroutines programmed with *WinBasic* which return a value with a RETURN statement.

String expressions

String constants and variables can form string expressions. These expressions are not compatible with other data types. Variables and constants of other types cannot be automatically converted.

Record and array items of type string can be used.

The only valid operator in string expressions is the concatenation operator "+", which does not mean *add* as in arithmetic expressions but combines two or more strings into one, for example:

```
Result$ = "abc" + a$ + STR$(1.23)
```

Here, STR\$() is an intrinsic function of type string, which allows the use of the real constant 1.23 in a string expression by explicit type conversion.

Logical expressions

Basic logical expressions are formed by constants, variables and the comparison operators:

=	equal
<	less
<=	less or equal
>	greater
>=	greater or equal
<>	not equal

For example:

```
If a% > 0 Then ...  
While sum% = 0
```

The comparison operator '=' is not to be confused with the assignment operator '='. They share the same token, but have a different meaning. In the second example above, the variable *sum%* is not assigned a value; rather, a comparison is done to determine if its value is equal with that of the constant 0.

The value of a logical expression is -1 if the comparison is TRUE, or 0 if the comparison is FALSE.

Because logical expressions have a numeric result, they can be used in arithmetic expressions, for example

```
a% = b% > 1
```

If *b%* is greater than 1, the logical expression *b% > 1* is TRUE and has the value -1, so *a%* is set to -1. Otherwise the expression is FALSE, so *a%* is 0.

Vice versa, every arithmetic expression can be used as a logical expression, because every value not equal to 0 is interpreted as TRUE. The loop

```
i% = 10  
While i%  
    ...  
    i% = i% - 1  
Wend
```

is executed until i% is counted down to 0 and thus interpreted as FALSE by the WHILE statement.

Logical expressions can form compounded logical expressions when combined with the operators AND, OR and NOT, for example:

```
If i% > 0 AND sum > 100.0 Then ...  
If NOT s$ = "123" Then ...
```

If there is more than one compound operator in an expression, precedence rules are valid in a manner similar to that for arithmetic expressions. An AND operation is done before an OR operation in the same manner as multiplication is done before addition in arithmetic expressions. This precedence can be overridden using parentheses.

File IO

WinBasic supports the common input/output (IO) functions of BASIC. However, the use of random access files is enhanced by the implementation of records.

Two kinds of files are supported; text files and random access files.

Random access files have a fixed record length. Only complete records can be read from and written to a file. A record number is used to identify which record is to be read from or written to the file; the record number corresponds to the position of the record within the file. New records can only be appended to the end of a

file or overwrite the position of an existing record; it is not possible to create a file with random record numbers. However, records may be read from a file in any order. Random access files are read from and written to by the PUT and GET statements.

Text files are written and read sequentially, one character at a time, from the beginning of the file to the end. Text files are written to by the PRINT, LPRINT and WRITE statements, and read by the INPUT and LINE INPUT statements.

Both kinds of files are opened and/or created with the OPEN statement and eventually closed with the CLOSE statement.

Character sets

MS-DOS and Windows work with two different character sets: MS-DOS with the "IBM Extended Character Set" and Windows with the "ANSI Character Set". The difference between the two sets becomes obtrusive when using foreign characters.

Two functions are provided to convert character strings between these two standards, WINTODOS\$() and DOSTOWIN\$(). Details of these two functions can be found in the reference section of this manual.



EXAMPLE PROGRAMS

The source code for the following illustrative examples is supplied on the distribution disk.

HELLO.BAS

The following trivial program simply prints a message in the program's window:

```
Print "Hello World"
```

MESSAGE.BAS

This example illustrates the creation of menus and processing of messages from both these menus and from Windows.

```
1  Dim Menu$(4)
2
3  Menu$(1)="Menu1 MenuItem_1001 MenuItem_1002"
4  Menu$(2)="&Menu2"
5  Menu$(3)="Menu3 MenuItem_3001 MenuItem_3002"
6  Menu$(4)=""
7
8  Menu Menu$
9
10 _1001:
11 _1002:
12   a$=chr$(getmessage()-1000+48)
13   MessageBox "Item "+a$, "Menu1", 0, answer%
14 _2000:
15   MessageBox "direct working", "Menu2", 0,
answer%
16
17 _3001:
18 _3002:
```

```

19     a$=chr$(getmessage$()-3000+48)
20     MsgBox "Item "+a$, "Menu3", 0, answer%
21 _KEY:
22     a$=inkey$()
23     Print a$, asc%(a$)
24 _MOUSE:
25     Mouse Button%, x%, y%
26     MoveTo x%, y%
27     Print "Button ";Button%

```

Note that the program lines are preceded by line numbers to assist the following explanation; they do not appear in the file.

Lines 1 to 8 define and create a menu. To do this, a string array is created with as many items as menu items on the menu bar plus one. In this example, four items in the string array are needed. The array elements are instantiated with strings which contain the commands for the menus. The first word is the title of the menu displayed on the menu bar. The remaining words are the commands within the menu, which send messages to the main program when clicked by the user. If a string contains only one word like the one in line 4, no popup menu is generated; instead, the menu title creates a message when clicked by the user. The last item of the string array is instantiated with an empty string, "", to indicate the end of the menu definition. The statement on line 8 creates and displays the menu in the program's window.

The program is now waiting for messages. Commands generated by the user (from the menus, keyboard or mouse) are served by the appropriate Windows labels.

When the menu is created, a code is computed for every command in the menu. All commands in the first menu are assigned codes beginning with 1000, those in the second menu

are assigned codes beginning with 2000 and so on. Within a menu the commands are numbered sequentially from the code of its title; refer to the example for an illustration.

The commands in the menus send their codes as messages when they are selected by the user. The Windows labels at which a suspended program restarts execution are comprised of the code of the awaited message prefixed by an underscore character. This underscore distinguishes Windows labels from normal BASIC labels. Each Windows label is followed by statements that define the program's response to the message. When the next Windows label is reached, the program is suspended again awaiting the next message. When the physical end of the program is reached (line 27 in the above example) the program is suspended too, not terminated. The program can only be terminated by the STOP or END statement or by the user clicking **Close** from the program's system menu.

The Windows labels `_1001` and `_1002` are grouped together. These messages are served by the same program statements. When messages are served in groups in this way it may be necessary to distinguish the exact message within the group of program statements. The built-in function `GETMESSAGE%()` is useful in this context, as it returns the code of the last message received.

The Windows labels `_KEY` and `_MOUSE` are entry points for messages created by key presses on the keyboard and clicks on the mouse. The message `KEY` (code -200 from `GETMESSAGE%()`) is only created for printable characters and the function keys F1 to F9 and Shift-F1 to Shift-F9. The pressed key can be determined with the built-in function `INKEY$()`.

The message MOUSE (code -300) is sent when the user presses a button on the mouse. Information about the pressed button and the position of the mouse within the program's window can be obtained with the MOUSE statement.

DIALOG.BAS

This program illustrates the implementation of dialog boxes.

```
1  Dim Menu$(3)
2
3  Subroutine SetFont()
4      Dialog 60, 20, 160, 120, 0, 0, "Set Font"
5      Dialog 10, 10, 80, 100, -1, 10, "Font"
6      Dialog 20, 20, 60, 12, 11, 12, "Times"
7      Dialog 20, 32, 60, 12, 12, 12, "Helvetica"
8      Dialog 20, 44, 60, 12, 13, 12, "Swiss"
9      Dialog 20, 56, 60, 12, 14, 12, "Script"
10     Dialog 20, 68, 60, 12, 15, 12, "Roman"
11     Dialog 20, 80, 60, 12, 16, 12, "System"
12     Dialog 20, 92, 60, 12, 17, 12, "Courier"
13     Dialog 100, 16, 30, 10, -1, 5, "Height:"
14     Dialog 140, 14, 12, 12, 21, 18, "48"
15     Dialog 100, 30, 30, 10, -1, 5, "Width:"
16     Dialog 140, 28, 12, 12, 22, 18, "30"
17     Dialog 100, 44, 50, 10, 31, 1, "Bold"
18     Dialog 100, 56, 50, 10, 32, 1, "Italic"
19     Dialog 100, 68, 50, 10, 33, 1, "Under"
20     Dialog 100, 82, 50, 12, 3, 13, "OK"
21     Dialog 100, 98, 50, 12, 4, 11, "Cancel"
22     Dialog
23 _INIT:
24     Dlgitem 16, 11, 17, 6, chr$(1)
25     bold%=0
26     under%=0
27     italic%=0
```



```

28 _1: 'Enter
29 _3: 'OK-Button
30 _ Dlgitem 21, itemtext$ : h% = val%(itemtext$)
31 _ Dlgitem 22, itemtext$ : w% = val%(itemtext$)
32 _ Font h%, w%, bold%, italic%+under%*2, FF%
33 _ Cls
34 _ Print "Your Font"
35 _ Dialog @
36 _2: 'Escape
37 _4:
38 _ Dialog @
39 _11:
40 _12:
41 _13:
42 _14:
43 _15:
44 _16:
45 _17:
46 _ item% = GetMessage%()
47 Dlgitem item%, 11, 17, 6, chr$(1)
48 FF%=item%-10
49 _31:
50 _ bold% = 1-bold%
51 _ Dlgitem 31, 0, 0, bold%*4+2, chr$(1)
52 _32:
53 _ italic% = 1-italic%
54 _ Dlgitem 32, 0, 0, kursiv%*4+2, chr$(1)
55 _33:
56 _ under% = 1-under%
57 _ Dlgitem 33, 0, 0, under%*4+2, chr$(1)
58 Endsub
59
60 REM *** Main program ***
61
62 Menu$(1)="E&xit"
63 Menu$(2)="&Font"
64 Menu$(3)=""
65
66 Menu Menu$
67

```

```

68  _1000:
69      Stop
70  _2000:
71      Gosub SetFont ()

```

Note that the program lines are preceded by line numbers to assist the following explanation.

Dialog boxes are driven by a related subroutine. The example program consists of a main program beginning at line 60, which generates the menu and services the messages created by this menu, and a subroutine to drive the dialog box. This subroutine has three duties:

- ◆ Creation of the dialog box;
- ◆ Responding to messages created by the dialog box;
- ◆ Termination of the dialog box.

The dialog box in the example generates and displays a list of fonts, as well as allowing input of the attributes and size of a font. The dialog then displays an instance of the specified font.

The dialog box is created with multiple statements of the form:

```
Dialog x%, y%, dx%, dy%, id%, type%, text$
```

each of which defines one item to be placed in the dialog box.

x% and *y%* are the coordinates of the upper left corner of the item; *dx%* and *dy%* specify the size of the item.

id% is used to identify each item within the dialog box. It must be

unique within the dialog box because it is used as a code for messages.

type% determines the type of the item: constant text, edit field, push button and so on.

text\$ is used to give the item a title.

A more thorough discussion of the parameters of the DIALOG statement can be found in the reference section.

The first DIALOG statement with type 0 generates a frame for the dialog box. Setting *text\$* to "Set Font" displays a title for the dialog box. *id%* has no meaning in this case, because the frame does not send messages. *x%* and *y%* define the upper left corner of the dialog box with in the program's window.

The following DIALOG statements generate the items in the dialog box. In each case *x%* and *y%* determine the left upper corner of the item within the dialog box.

The first item of type 10 is a group box, *i. e.* a line frame with *text\$* as title. *id%* has no meaning because the group box cannot be clicked by the user and therefore cannot send a message. It is set to -1.

The group box bounds the following seven radio buttons, which are generated by type 12. The radio buttons have successive id's 11 to 17.

Then there are two pairs of constant left justified text (type 5) and left justified edit fields (type 18). The texts have -1 as their id's, because they do not send messages.

Three check boxes are generated by type 1 to determine the font properties bold, italic and underscore. Check boxes are used for yes/no decisions.

Next are two push buttons. Type 13 specifies a default push button "OK" and type 11 a push button "Cancel". The default push button has a thicker frame than a normal push button. The dialog box should be programmed in such a manner that clicking the default push button gives the same result as pressing <Enter> on the keyboard (which generates a message of code 1 in dialog boxes).

The statement `DIALOG` without any parameters in line 22 ends the definition and displays the dialog. Key presses are directed into the first edit field or default push button in the sequence of the defining statements. This item has the so-called "input focus". The input focus can be given to other items by the user by pressing the tabulator keys or clicking the mouse on another item. The item who has the input focus has a dotted frame around it, or, in the case of an edit field, it contains a text cursor.

If the user chooses one of the items by clicking with the mouse or pressing <Space>, the item sends its identifier as a message to the subroutine which controls the dialog box.

The subroutine has appropriate Windows labels to serve as entry points to the code that services the messages. The Windows labels consist of the identifier code of the message, prefixed with an underscore.

The message `INIT` is sent by Windows immediately before the dialog box is displayed in the program's window. This allows

initialization of the dialog box to take place. In the above example, lines 24 to 27 set radio button 16 (in the group of radio buttons 11 to 17) on, and resets flag variables for the check boxes. Refer to the entry for DLGITEM in the reference section for more details.

The messages 11 to 17 generated by the radio buttons are served by a group of Windows labels. The built-in function GETMESSAGE%() is used to determine which button was clicked. The id's of the radio buttons are chosen in such a way that the code for their respective font family (required by the FONT statement) is obtained simply by subtracting 10.

If one of the three check boxes 31 to 33 is clicked, the related flag variable is toggled between 0 and 1 and depending on the result of this the appropriate check mark is either set or reset.

Messages from the edit fields are not served; any messages these controls generate should be left to the internal routines. Only when the messages 1 (Enter) or 3 (push button "OK") are received are the edit fields read and values computed from their contents.

The input parameters are combined and passed to the FONT statement (line32). Subsequently, some sample text is displayed in the created font.

The dialog box is terminated by the statement:

```
Dialog @
```

This causes the subroutine to return immediately, and therefore functions like a RETURN statement.

On reception of the messages 2 (Esc key) or 4 (push button "Cancel") the dialog box is terminated without further action.



The BITMAP.EXE Utility

BITMAP.EXE is a small program which copies graphic data from the clipboard into a file. It is used for creating icons for *WinBasic* programs in the following manner:

1. Start the Windows Paint program, set the size to 32 x 32 pixels and make a small drawing
2. Select the drawing with the mouse and copy it to the clipboard using the **Edit Copy** command.
3. Run the program BITMAP.EXE. This copies the contents of the clipboard into the file ICON.ICO.
4. Rename the file ICON.ICO to the same name as your *WinBasic* program. For example, if your program is called HELLO.BAS, you should rename ICON.ICO to HELLO.ICO.

Note that BITMAP.EXE does not open a window.

